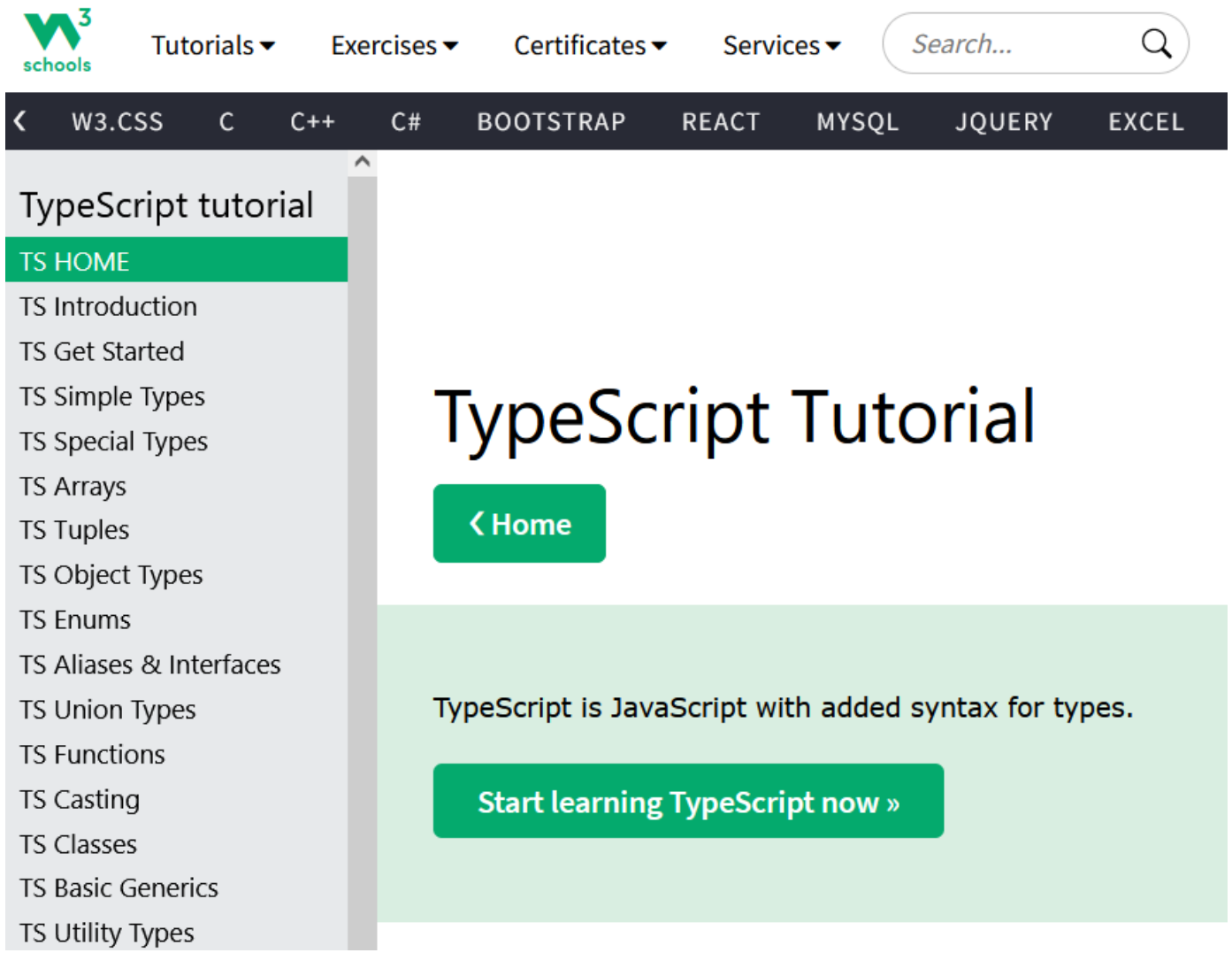
TypeScript

tutorial version 1.8

TypeScript

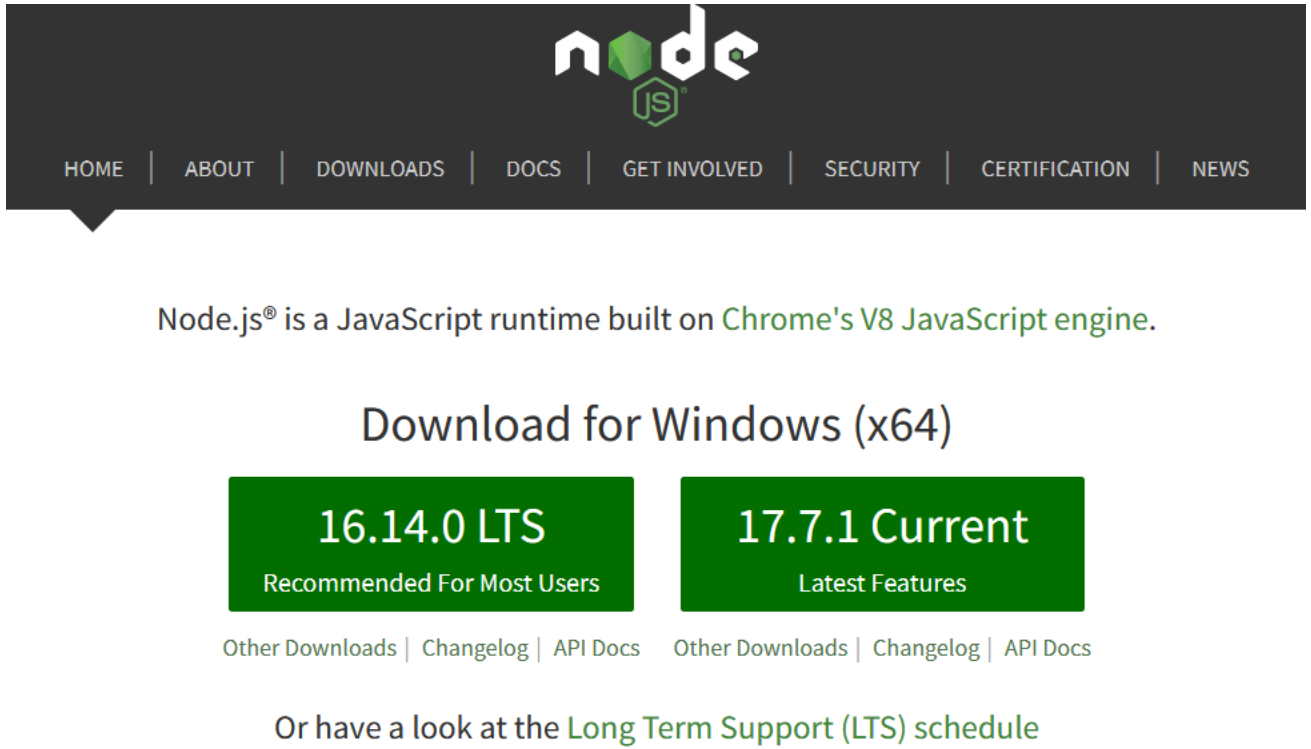
https://www.typescriptlang.org/

TypeScript



https://www.w3schools.com/typescript/

Node.js



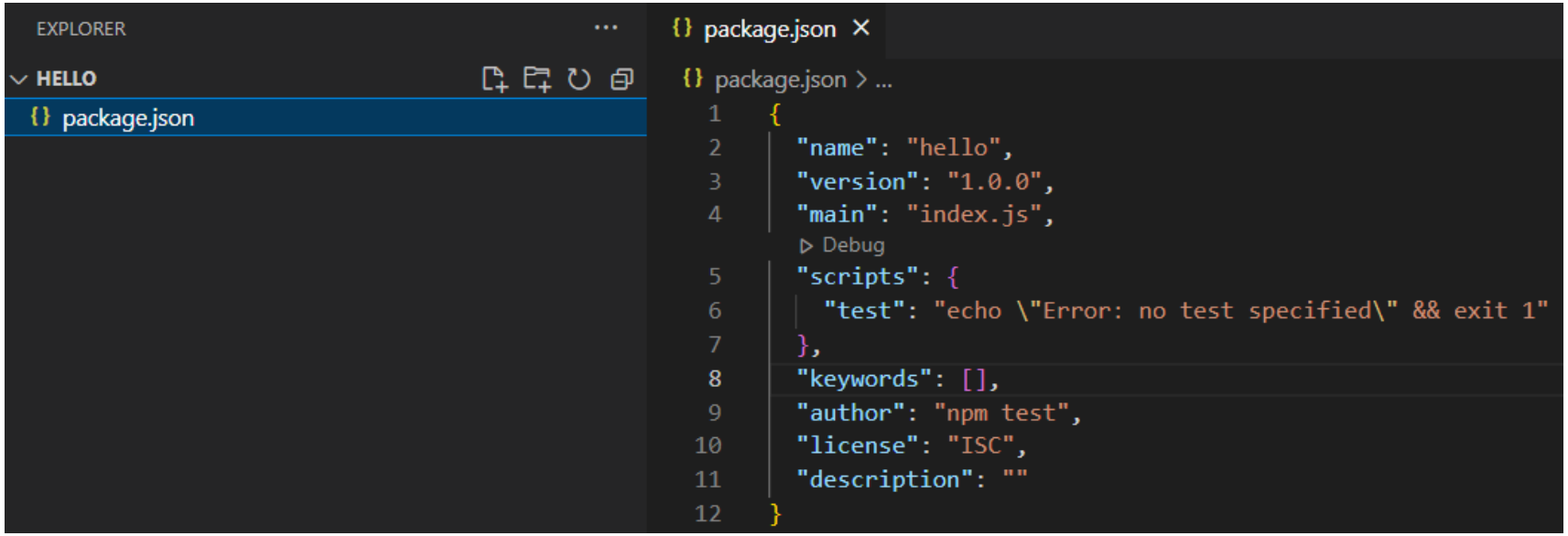
https://nodejs.org/en/

Node.js
nowy projekt

w Visual Studio Code w katalogu projektu (hello) w terminalu wydajemy polecenie inicjalizujące projekt:

```
npm -y init
```

wygenerowany zostaje plik package.json z konfiguracją projektu



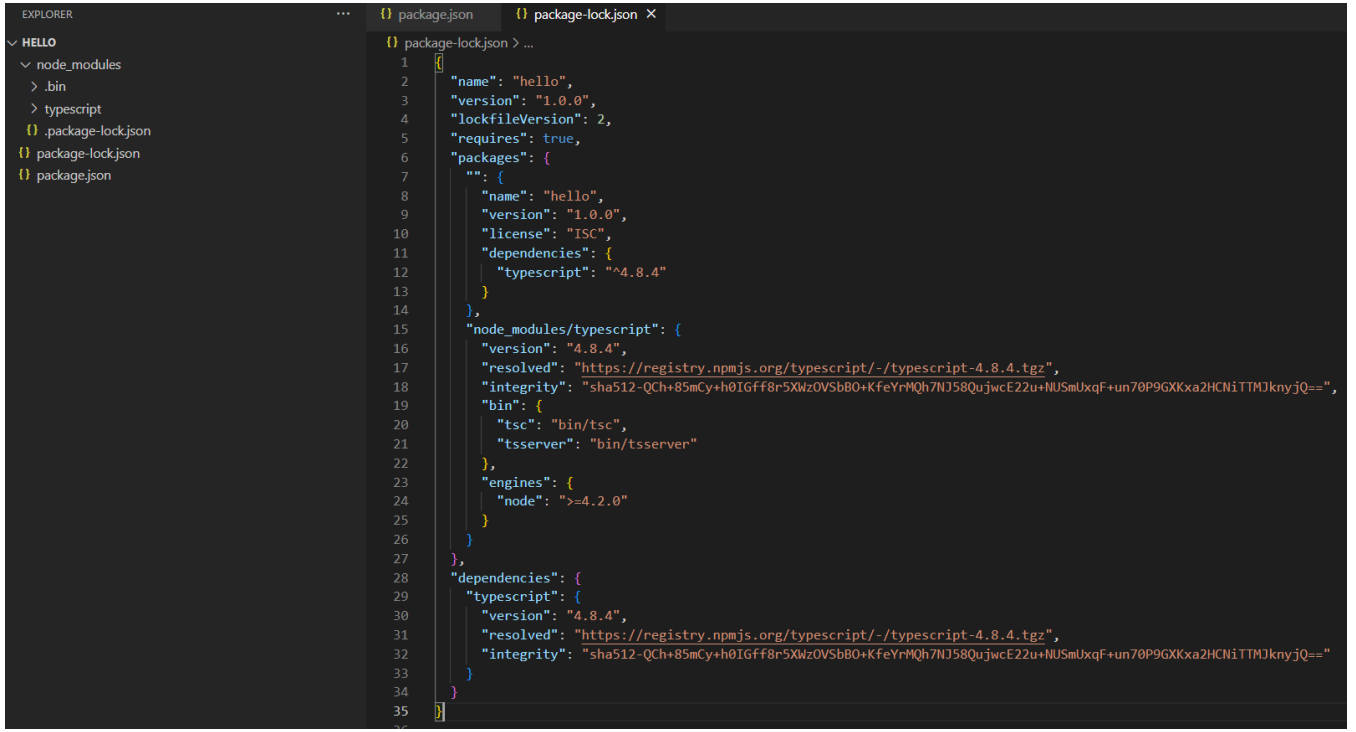
Node.js
instalacja TypeScript

w Visual Studio Code w katalogu projektu (hello) w terminalu wydajemy polecenie instalujące język TypeScript:

```
npm install -g typescript
```

//g: global

wygenerowany zostaje plik package-lock.json z informacjami dotyczącymi języka oraz katalog node_modules z plikami obsługującymi TypeScript
(przy opcji –g pliki mogą być wygenerowane w innym katalogu)



Node.js
wersja TypeScript

sprawdzenie zainstalowanej wersji języka

```
tsc --version
```

```
Version 5.4.3
```

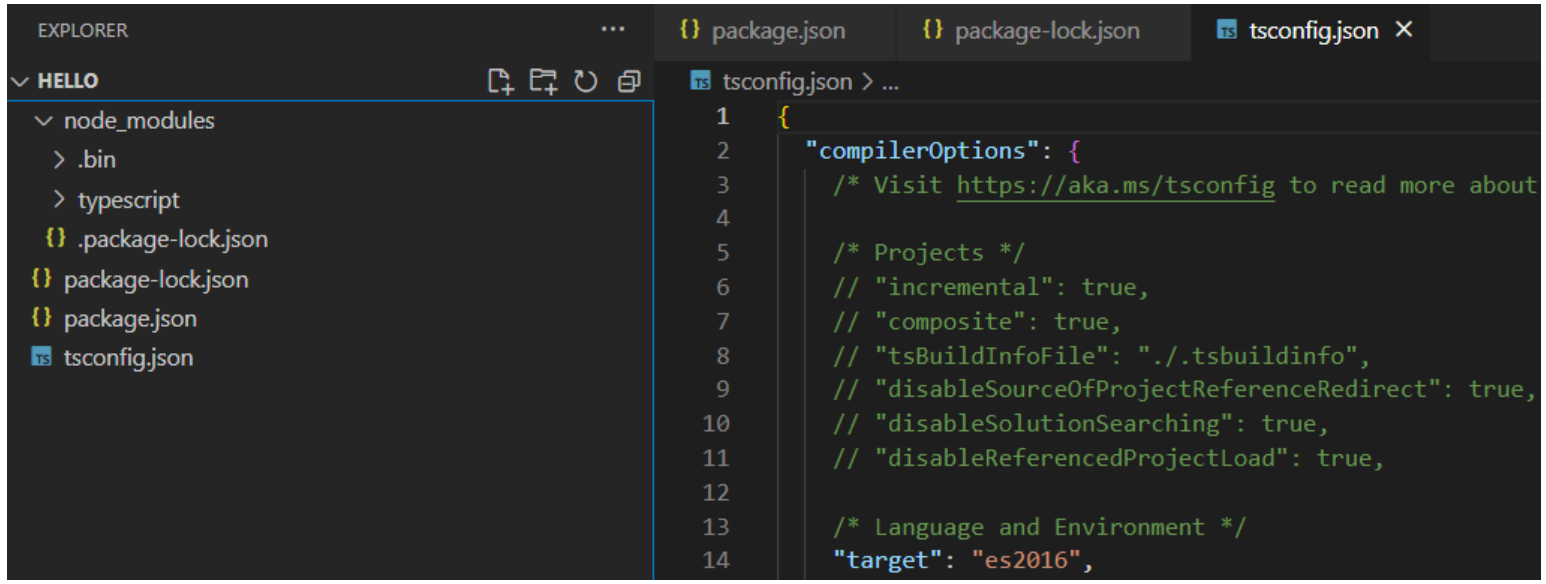
Node.js
inicjalizacja TypeScript

w Visual Studio Code w katalogu projektu (hello) w terminalu
wydajemy polecenie inicjalizujące język TypeScript:

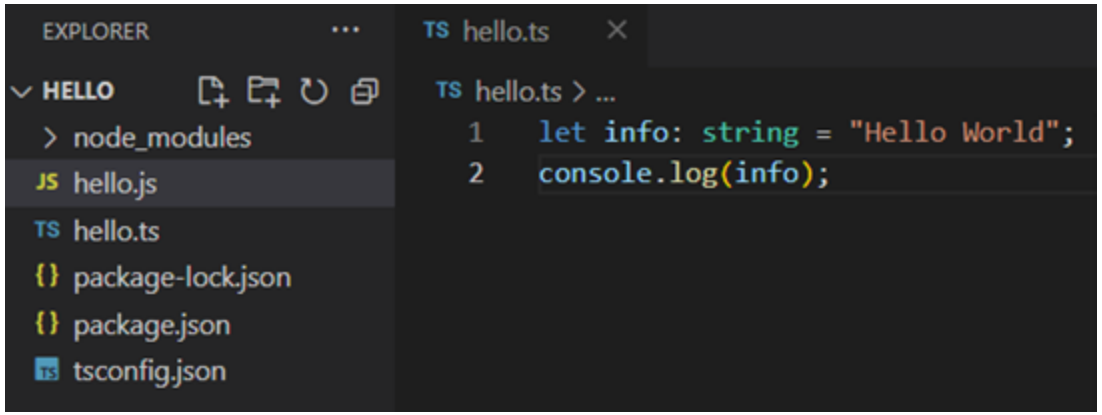
```
hello> npx tsc -init
```

npx: Node Package Execute - uruchomienie modułu
z katalogu node_modules
tsc: typescript compiler

wygenerowany zostaje plik tsconfig.json z konfiguracją kompilatora



Node.js
plik hello.ts



tworzymy plik hello.ts w języku TypeScript

Node.js
TypeScript

aby uruchomić plik ts w środowisku node.js należy go przekonwertować do js

w Visual Studio Code w katalogu projektu (hello) w terminalu kompilujemy plik hello.ts - wydajemy polecenie konwerujące plik hello.ts do języka JavaScript:

```
npx tsc hello.ts
```

wygenerowany zostaje plik w języku JavaScript hello.js który można uruchomić w node.js

plik źródłowy (jest typ string - statyczny mechanizm typowania)

wygenerowny plik
po kompilacji (brak typu string – dynamiczny mechanizm typowania)

Node.js
TypeScript

```
PS D:\hello> node hello.js
Debugger attached.
Hello World
Waiting for the debugger to disconnect...
```

aby uruchomić plik hello.js wpisujemy polecenie:

```
node hello.js
```

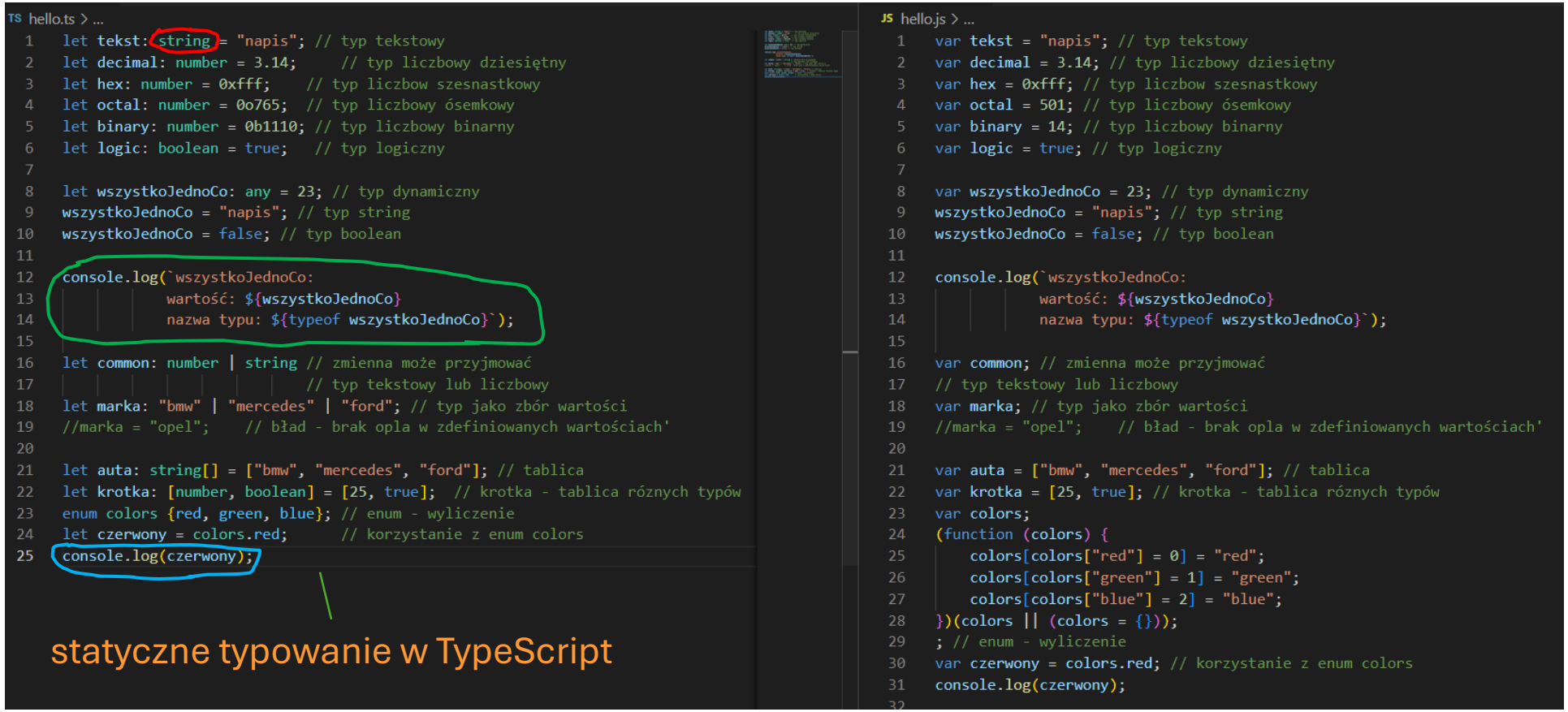
```
npx tsc --watch
```

polecenie uruchmiające tryb nasłuchu – automatyczną kompilacje pliku ts po wykryciu w nim zmiany

TypeScript

hello.ts

hello.js



TS hello.ts > ...

```ts
let tekst: string = "napis"; // typ tekstowy
let decimal: number = 3.14;     // typ liczbowy dziesiętny
let hex: number = 0xfff;    // typ liczbow szesnastkowy
let octal: number = 0o765;  // typ liczbowy ósemkowy
let binary: number = 0b1110; // typ liczbowy binarny
let logic: boolean = true;   // typ logiczny

let wszystkoJednoCo: any = 23; // typ dynamiczny
wszystkoJednoCo = "napis"; // typ string
wszystkoJednoCo = false; // typ boolean

console.log(`wszystkoJednoCo:
            wartość: ${wszystkoJednoCo}
            nazwa typu: ${typeof wszystkoJednoCo}`);

let common: number | string // zmienna może przyjmować
                            // typ tekstowy lub liczbowy
let marka: "bmw" | "mercedes" | "ford"; // typ jako zbór wartości
//marka = "opel";    // bład - brak opla w zdefiniowanych wartościach'

let auta: string[] = ["bmw", "mercedes", "ford"]; // tablica
let krotka: [number, boolean] = [25, true];  // krotka - tablica róznych typów
enum colors {red, green, blue}; // enum - wyliczenie
let czerwony = colors.red;      // korzystanie z enum colors
console.log(czerwony);
```

JS hello.js > ...

```js
var tekst = "napis"; // typ tekstowy
var decimal = 3.14; // typ liczbowy dziesiętny
var hex = 0xfff; // typ liczbow szesnastkowy
var octal = 501; // typ liczbowy ósemkowy
var binary = 14; // typ liczbowy binarny
var logic = true; // typ logiczny

var wszystkoJednoCo = 23; // typ dynamiczny
wszystkoJednoCo = "napis"; // typ string
wszystkoJednoCo = false; // typ boolean

console.log(`wszystkoJednoCo:
            wartość: ${wszystkoJednoCo}
            nazwa typu: ${typeof wszystkoJednoCo}`);

var common; // zmienna może przyjmować
// typ tekstowy lub liczbowy
var marka; // typ jako zbór wartości
//marka = "opel";    // bład - brak opla w zdefiniowanych wartościach'

var auta = ["bmw", "mercedes", "ford"]; // tablica
var krotka = [25, true]; // krotka - tablica róznych typów
var colors;
(function (colors) {
    colors[colors["red"] = 0] = "red";
    colors[colors["green"] = 1] = "green";
    colors[colors["blue"] = 2] = "blue";
})(colors || (colors = {}));
; // enum - wyliczenie
var czerwony = colors.red; // korzystanie z enum colors
console.log(czerwony);
```

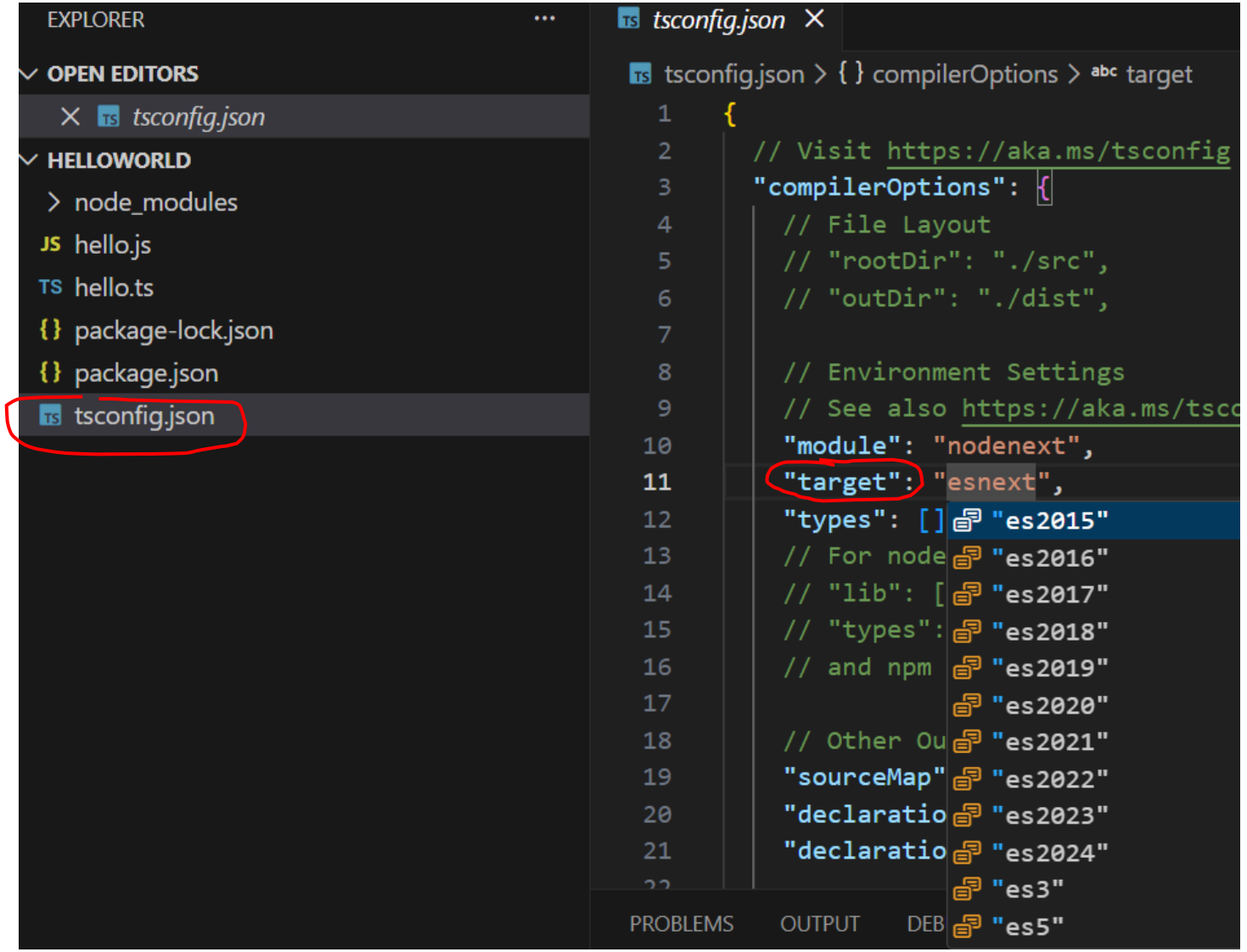
statyczne typowanie w TypeScript

```
PS D:\hello> node hello.js
Debugger attached.
wszystkoJednoCo:
            wartość: false
            nazwa typu: boolean
0
```

uruchomienie hello.js

dynamiczne typowanie w JavaScript
(wygenerowany plik hello.js może się różnić w zależności od wybranej wersji kompilatora)

TypeScript

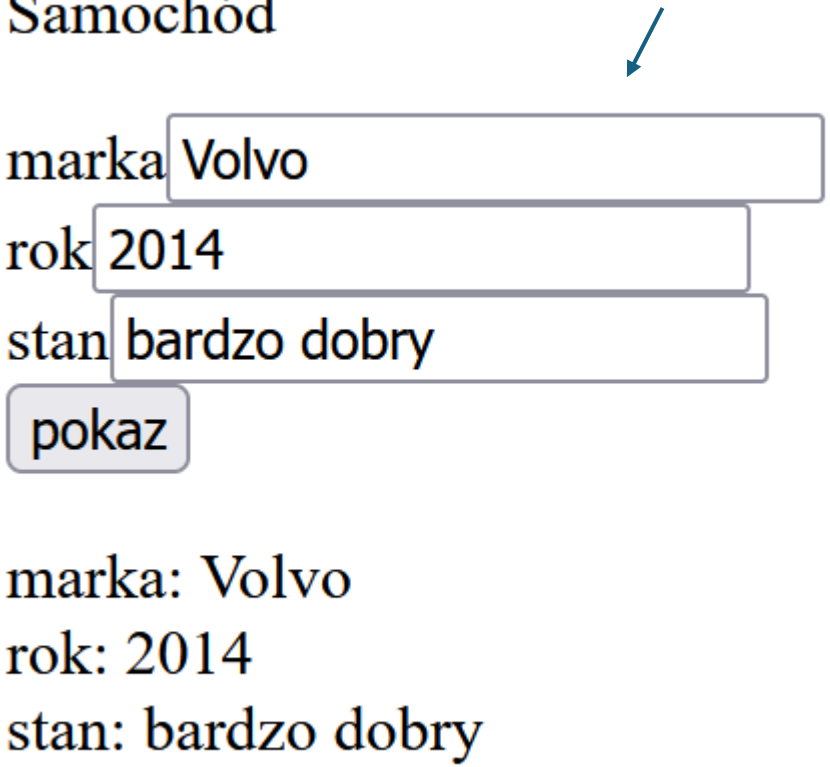


wybór wersji kompilatora TypeScript

TypeScript - obiekty

napisz stronę internetową o podanej funkcjonalności.
Całość ma się składać z plików index.html oraz app.ts (nie piszemy skryptu w js tylko w ts, plik ts przekonwertujemy do js w środowisku node.js)

skrypt w ts pobiera wartości pól do obiektu samochod



skrypt wyświetla informacje z obiektu samochod po naciśnięciu na przycisk „pokaz”

TypeScript – wbudowane typy DOM

W TypeScripcie dla formularzy HTML używa się **wbudowanych typów DOM**, takich jak `HTMLInputElement`.
Najczęściej spotykane typy to:

Najważniejsze typy elementów formularza

- `HTMLInputElement` – `<input>`
- `HTMLTextAreaElement` – `<textarea>`
- `HTMLSelectElement` – `<select>`
- `HTMLOptionElement` – `<option>`
- `HTMLButtonElement` – `<button>`
- `HTMLFormElement` – `<form>`
- `HTMLLabelElement` – `<label>`

Wszystkie one dziedziczą po `HTMLElement`.

TypeScript - obiekty

```html
<!DOCTYPE html>
<html>
<head>
    <meta charset='utf-8'>
    <title>TypeScript</title>
</head>
<body>
    <p>Samochód</p>
    marka<input id="marka" type="text"><br>
    rok<input id="rok" type="text"><br>
    stan<input id="stan" type="text"><br>
    <button id="btn-pokaz">pokaz</button>

    <p id="info"></p>
    <script src='app.js'></script>
</body>
</html>
```

index.html

app.ts

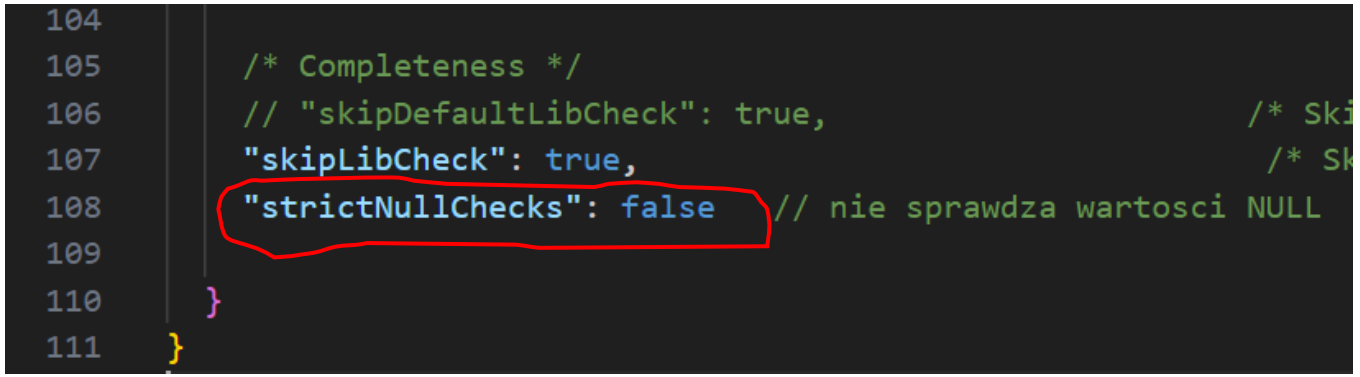
```ts
let info = document.getElementById("info");
let btnPokaz = document.getElementById("btn-pokaz");


btnPokaz.onclick = function(){
    // obiekt samochód
    let samochod: { marka: string, rok: number, stan: string }={
        marka  : (<HTMLInputElement>document.getElementById("marka")).value,
        rok : parseInt((<HTMLInputElement>document.getElementById("rok")).value),
        stan : (<HTMLInputElement>document.getElementById("stan")).value
    }

    info.innerHTML = `marka: ${ samochod.marka}<br>rok: ${samochod.rok}<br>stan: ${samochod.stan}`;
}
```

app.js powstaje po wygenerowaniu z pliku app.ts

TypeScript - obiekty - errors

tsconfig.json – należy wyłączyć w ts sprawdzanie wartości null

'btnPokaz' is possibly 'null'.



Property 'value' does not exist on type 'HTMLElement'

Based on Tomasz Nurkiewiczs answer, the "problem" is that typescript is typesafe. :) So the `document.getElementById()` returns the type `HTMLElement` which does not contain a `value` property. The subtype `HTMLInputElement` does however contain the `value` property.

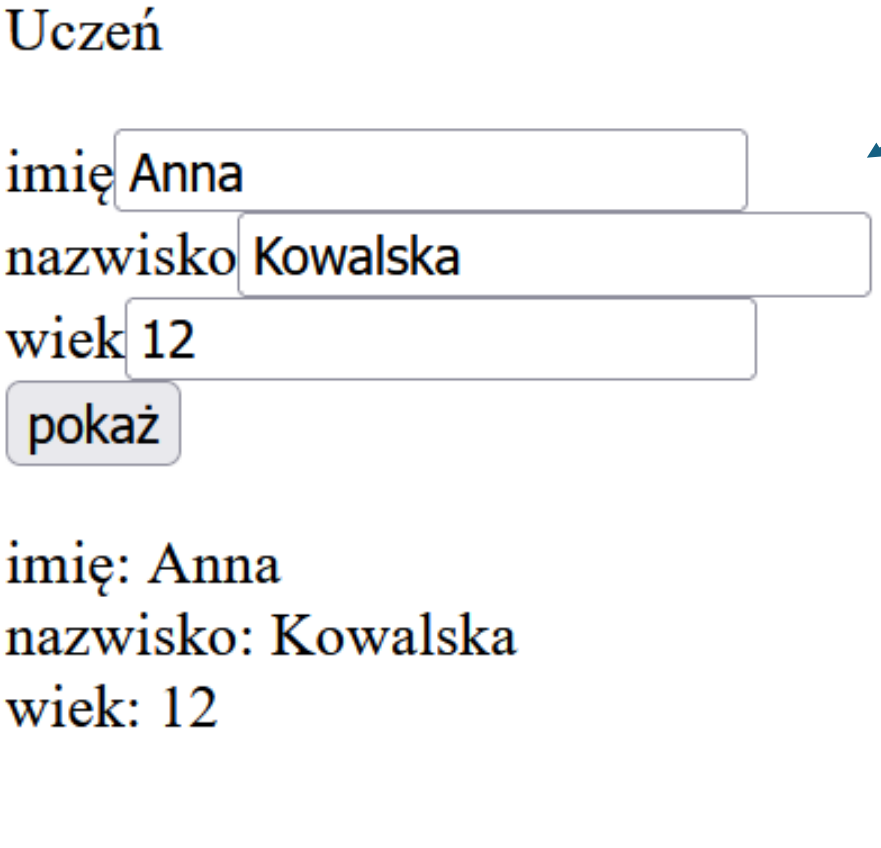
So a solution is to cast the result of `getElementById()` to `HTMLInputElement` like this:

```
var inputValue = (<HTMLInputElement>document.getElementById(elementId)).value;
```

https://stackoverflow.com/questions/12989741/the-property-value-does-not-exist-on-value-of-type-htmlelement

TypeScript - aliasy

inna nazwa typu zmiennej



skrypt w ts pobiera wartości pól do obiektu osoba stworzonego z wykorzystaniem aliasów

skrypt wyświetla informacje z obiektu osoba stworzonego z wykorzystaniem aliasów po naciśnięciu na przycisk „pokaz”

https://www.w3schools.com/typescript/typescript_aliases_and_interfaces.php

TypeScript - aliasy

app.ts

Aliases > app.ts > ...

```ts

let info = document.getElementById("info");
let btnPokaz = document.getElementById("btn-pokaz");

// lokalne aliasy podstawowych typów danych
type WiekUcznia = number;
type ImieUcznia= string;
type NazwiskoUcznia = string;

type Uczen =  {
    imie: ImieUcznia,
    nazwisko: NazwiskoUcznia ,
    wiek: WiekUcznia
}

btnPokaz.onclick = function(){
  // obiekt uczen
  let osoba: Uczen ={
      imie   : (<HTMLInputElement>document.getElementById("imie")).value,
      nazwisko : (<HTMLInputElement>document.getElementById("nazwisko")).value,
      wiek: parseInt((<HTMLInputElement>document.getElementById("wiek")).value)
  }


    info.innerHTML = `imię: ${ osoba.imie}<br>nazwisko: ${osoba.nazwisko}<br>wiek: ${osoba.wiek}`;
}
```

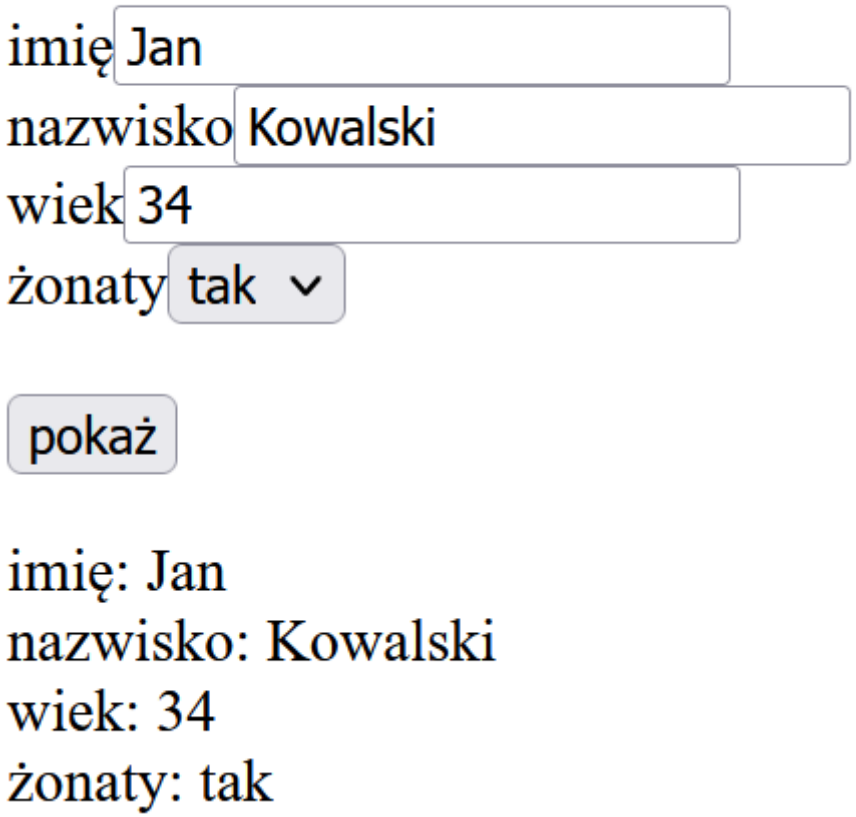
index.html

Aliases > index.html > ...

```html
<!DOCTYPE html>
<html>
<head>
    <meta charset='utf-8'>
    <title>TypeScript</title>
</head>
<body>
    <p>Uczeń</p>
    imię<input id="imie" type="text"><br>
    nazwisko<input id="nazwisko" type="text"><br>
    wiek<input id="wiek" type="text"><br>
    <button id="btn-pokaz">pokaż</button>

    <p id="info"></p>
    <script src='app.js'></script>
</body>
</html>
```

TypeScript - interfejsy

skrypt w ts pobiera wartości pól do obiektu pracownik stworzonego na podstawie interfejsu Pracownik



skrypt wyświetla informacje z obiektu pracownik stworzonego na podstawie interfejsu Pracownik po naciśnięciu na przycisk „pokaz”

https://www.w3schools.com/typescript/typescript_aliases_and_interfaces.php

TypeScript - interfejsy

index.html

Interfaces > index.html > ...

```html
<!DOCTYPE html>
<html>
<head>
    <meta charset='utf-8'>
    <title>TypeScript</title>
</head>
<body>
    <p>Pracownik</p>
    imię<input id="imie" type="text"><br>
    nazwisko<input id="nazwisko" type="text"><br>
    wiek<input id="wiek" type="text"><br>
    żonaty<select id="zonaty"><option>tak</option><option>nie</option></select><br><br>
    <button id="btn-pokaz">pokaż</button>

    <p id="info"></p>
    <script src='app.js'></script>
</body>
</html>
```

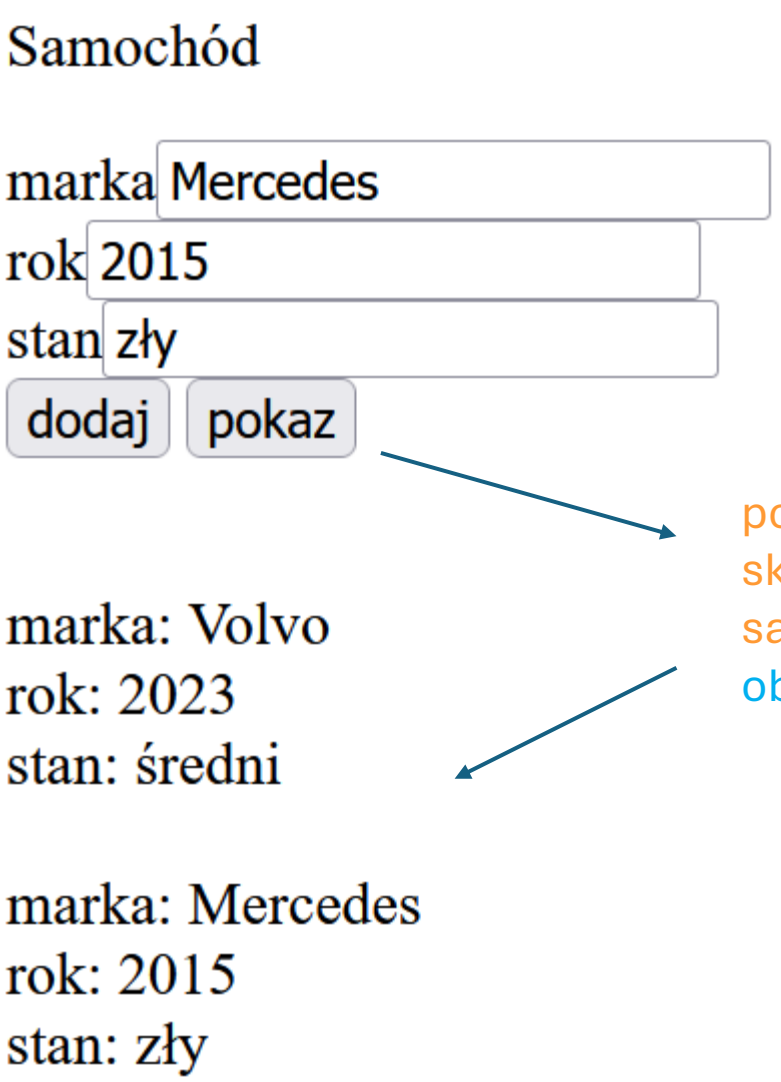
TypeScript - interfejsy

TS app.ts

Interfaces > TS app.ts > ...

```typescript

let info = document.getElementById("info");
let btnPokaz = document.getElementById("btn-pokaz");

// interfejs Pracownik - zamiast tworzyć kolejne obiekty pracowników,
// z których każdy posiada definicję, można utworzyć jeden wspólny interfejs.
interface Pracownik {
    imie: string,
    nazwisko: string,
    wiek: number,
    zonaty: string
  }


btnPokaz.onclick = function(){
  // obiekt pracownik
  let pracownik: Pracownik = {
      imie   : (<HTMLInputElement>document.getElementById("imie")).value,
      nazwisko : (<HTMLInputElement>document.getElementById("nazwisko")).value,
      wiek: parseInt((<HTMLInputElement>document.getElementById("wiek")).value),
      zonaty:(<HTMLInputElement>document.getElementById("zonaty")).value
  }


    info.innerHTML = `imię: ${ pracownik.imie}<br>nazwisko: ${pracownik.nazwisko}
                      <br>wiek: ${pracownik.wiek}<br>żonaty: ${pracownik.zonaty}`;
}
```

TypeScript - klasy



po naciśnięciu na przycisk „dodaj”
skrypt w ts pobiera wartości pól do obiektu klasy Car,
dodaje obiekt do tablicy cars obiektów klasy Car.
oraz wyświetla informacje o dodanym samochodzie

po naciśnięciu na przycisk „pokaz”
skrypt wyświetla informacje o samochodach z tablicy cars obiektów klasy Car

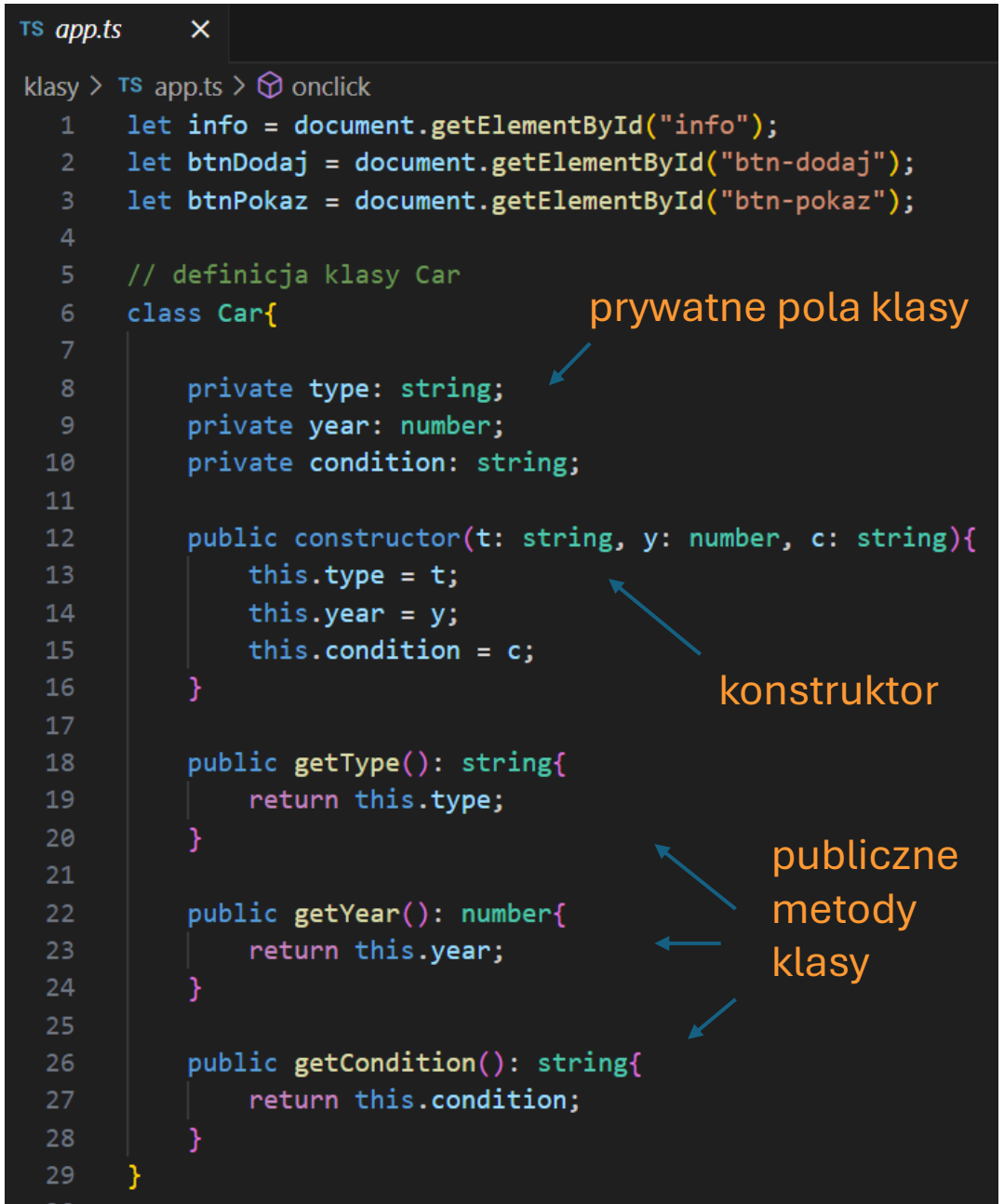
TypeScript - klasy

index.html

klasy > index.html > html > body > script

```
<!DOCTYPE html>
<html>
<head>
    <meta charset='utf-8'>
    <title>TypeScript</title>
</head>
<body>
    <p>Samochód</p>
    marka<input id="marka" type="text"><br>
    rok<input id="rok" type="text"><br>
    stan<input id="stan" type="text"><br>
    <button id="btn-dodaj">dodaj</button>
    <button id="btn-pokaz">pokaz</button>

    <p id="info"></p>
    <script src='app.js'></script>
</body>
</html>
```

TypeScript - klasy



TypeScript - klasy

```
31  // tablica obiektów klasy Car (pusta)
32  const cars: Car[] = [];
33
34
35  // dodanie samochodu do tablicy, wyświetlenie o nim informacji
36  btnDodaj.onclick = function(){
37      let t = (<HTMLInputElement>document.getElementById("marka")).value;
38      let y = parseInt((<HTMLInputElement>document.getElementById("rok")).value);
39      let c = (<HTMLInputElement>document.getElementById("stan")).value;
40
41      // utworzenie obiektu car klasy Car
42      let car = new Car(t,y,c);
43      // wstawienie utworzonego obiektu do tablicy cars
44      cars.push(car);
45      // wyświetlenie informacji o obiekcie
46      info.innerHTML=`marka: ${car.getType()}<br>rok: ${car.getYear()}<br>stan: ${car.getCondition()} `;
47  }
48
49  // wyświtlenie informacji o samochodach znajdujących się w tablicy
50  btnPokaz.onclick = function(){
51      info.innerHTML = "";
52      // pętla po elementach tablicy cars
53      for (const element of cars) {
54          info.innerHTML += `<br>marka: ${element.getType()}<br>rok: ${element.getYear()}
55                            <br>stan: ${element.getCondition()}<br>`;
56      }
57  }
```

TypeScript – klasy
klasa Car alternatywnie

```typescript
// klasa Car
class Car {
    // nie deklarujemy pól klasy, tylko podajemy je w konstruktorze
    public constructor(private type: string, private year: number, private condition: string) {
    }

    public getType(): string {
      return this.type;
    }

    public getYaer(): number{
        return this.year;
    }

    public getCondition(): string {
    return this.condition;
    }
  }
```

alternatywny sposób deklaracji pól klasy – konstruktor je wygeneruje

TypeScript – klasy
klasa Car alternatywnie

app.ts

Classes3 > app.ts > onclick

```ts

let info = document.getElementById("info");
let btnPokaz = document.getElementById("btn-pokaz");
let btnDodaj = document.getElementById("btn-dodaj");

// klasa Car
class Car {
    // konstruktor
    public constructor(private type?: string, private year?: number, private condition?: string) {
    }
    // properties (właściwości)
    get Type(){
      return this.type;
    }

    get Year(){
        return this.year;
    }

    get Condition(){
    return this.condition;
    }

    set Type(value){
        this.type = value;
    }
    set Year(value){
        this.year = value;
    }

    set Condition(value){
        this.condition = value;
    }
  }

```

alternatywny sposób deklaracji pól klasy – konstruktor je wygeneruje, pola są nieobowiązkowe (znak zapytania)

properties (właściwości) – zamiast deklaracji metod geterów i seterów

TypeScript – klasy
klasa Car alternatywnie

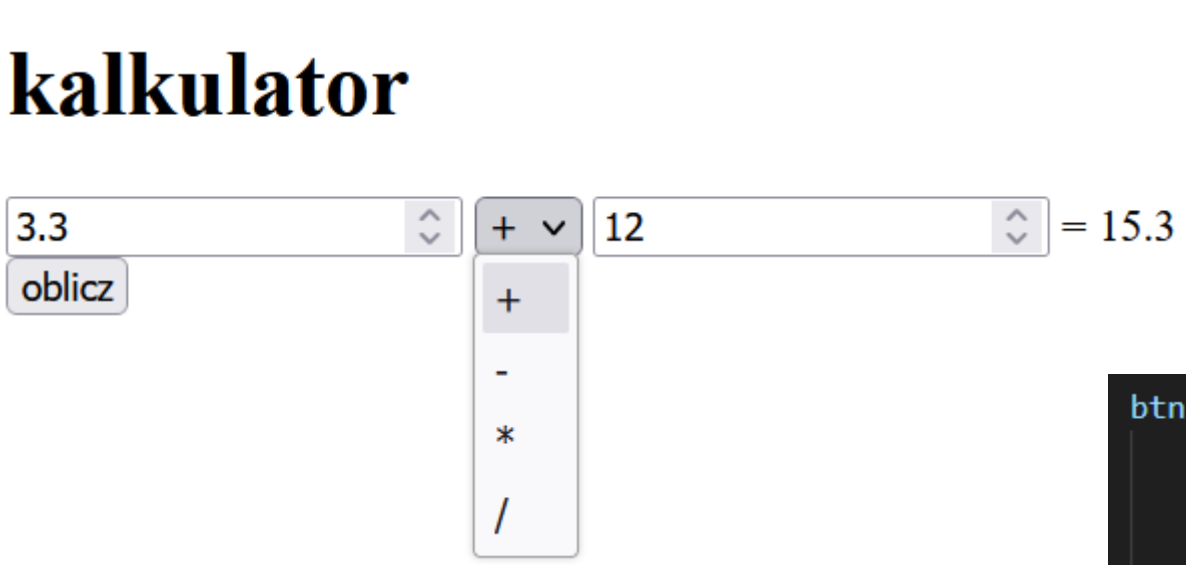
```
35
36  const cars: Car[] = [];
37
38  btnDodaj.onclick = function(){
39      const car = new Car();
40
41      car.Type =  (<HTMLInputElement>document.getElementById("marka")).value;
42      car.Year =  parseInt((<HTMLInputElement>document.getElementById("rok")).value);
43      car.Condition = (<HTMLInputElement>document.getElementById("stan")).value
44
45
46      cars.push(car);
47
48
49      info.innerHTML = `marka: ${car.Type}<br>rok: ${car.Year}<br>stan: ${car.Condition}`;
50  }
51
52  btnPokaz.onclick = function(){
53      info.innerHTML = "";
54      for (const element of cars) {
55          info.innerHTML += `<br>marka: ${element.Type}<br>rok: ${element.Year}<br>stan: ${element.Condition}<br>`;
56      }
57
58  }
```

konstruktor wygeneruje pola klasy (pomimo, że w definicji klasy są one opcjonalne)

z właściwości (properties) korzystamy jak z pól klasy

TypeScript - funkcje

napisz kalkulator korzystając z poniższych funkcji (trzeba je napisać)




```
btn.onclick = function () {
    // pobieramy liczby oraz operator z formularza
    var data = getNumbersAndOperator();
    // liczymy wynik działania
    var result = count(data);
    // wypisujemy rezultat
    printResult(result);
};
```

TypeScript - funkcje

index.html

Functions > index.html > html > body > input#a

```
1   <!DOCTYPE html>
2   <html>
3       <head>
4           <title>kalkulator</title>
5           <meta charset="utf-8">
6           <link rel="stylesheet" href="main.css">
7           <script src="app.js" defer></script>
8       </head>
9       <body>
10      <h1>kalkulator</h1>
11          <input id="a" type="number">
12          <select id="op">
13              <option>+</option>
14              <option>-</option>
15              <option>*</option>
16              <option>/</option>
17          </select>
18          <input id="b" type="number">
19          <label>=</label>
20          <label id="wynik"></label><br>
21          <button id="btn">oblicz</button>
22      </body>
23  </html>
```

TypeScript - funkcje

app.ts

Functions > app.ts > ...

```ts
let btn = document.getElementById("btn");

// interfejs Data (dane)
interface Data {
  a: number,
  b: number,
  op: string
}

btn.onclick = function(){

  // pobieramy liczby oraz operator z formularza
  let data = getNumbersAndOperator();
  // liczymy wynik działania
  let result = count(data);
  // wypisujemy rezultat
  printResult(result);

}

// pobiera dane z formularza
function getNumbersAndOperator() : Data{
  let aa = parseFloat((<HTMLInputElement>document.getElementById("a")).value);
  let bb = parseFloat((<HTMLInputElement>document.getElementById("b")).value);
  let opop = (<HTMLInputElement>document.getElementById("op")).value;
  let data: Data = {
    a : aa,
    b: bb,
    op: opop
  }
  // zwracamy obiekt typu Numbers
  return data;
}
```

TypeScript - funkcje

```
34
35  // funkcja obliczajaca wynik zwraca liczbe lub napis (komunikat przy dzieleniu przez zero)
36  function count(d: Data) : number | string{
37    let result: number | string = 0;
38    switch (d.op){
39      case "+":
40        result = d.a + d.b;
41        break;
42      case "-":
43        result = d.a - d.b;
44        break;
45      case "*":
46        result = d.a * d.b;
47        break;
48      case "/":
49        if (d.b != 0){
50          result = d.a / d.b;
51        }else{
52          result = "nie dzielimy przez zero!"
53        }
54        break;
55    }
56
57    return result;
58  }
59
60  // przy wypisywaniu wyniku należy uwzględnić 2 typy: string oraz number
61  function printResult(res: number | string){
62    if (typeof res == "string"){
63      document.getElementById("wynik").innerHTML = res;
64    }else{
65      document.getElementById("wynik").innerHTML = res.toString();
66    }
67  }
```

TypeScript – funkcje parametry opcjonalne

parametr opcjonalny

```
function orderIce(yourChoice? :string ){
  console.log(yourChoice || "lody z polewą czekoladową")
}

orderIce("lody z polewą truskawkową")
orderIce()
```

```
lody z polewą truskawkową
lody z polewą czekoladową
```

wywołanie funkcji z parametrem opcjonalnym

https://www.w3schools.com/typescript/typescript_functions.php

TypeScript – funkcje named parameters

named parameters

```
function orderCar({type, year, isAfterAccident}: {type:string,  year: number, isAfterAccident: boolean }){
    console.log(`you choose: \ntype: ${type}\nafter accident:${isAfterAccident}\nyear: ${year}`)
}

orderCar({type:"Volvo", year: 2023, isAfterAccident:false});
```

```
you choose:
type: Volvo
after accident:false
year: 2023
```

https://www.w3schools.com/typescript/typescript_functions.php

TypeScript – funkcje
rest parameter

rest parameter

```typescript
function add(a: number, b: number, ...tab: number[]): number{
    let result: number = 0;
    result = a + b;
    tab.forEach((x) => result += x);
    return result;
}

console.log(add(1,2,3,4,5,6,7,8,9,10));
```

rest parameter

a b

```
55
```

rest parameter:

- funkcja może mieć tylko jeden taki parametr
- musi być na końcu wszystkich parametrów
- musi być typu Array (tablica)

https://www.w3schools.com/typescript/typescript_functions.php

implementacja interfejsów, dziedziczenie

app.ts

app.ts > Swimmable > swim

```ts
// interfejs jest kontraktem - gdy klasa go implementuje -
// musi zaimplementowac wszystkie pola i metody interfejsu
interface Flyable {
  fly(): void;
}

interface Swimmable {
  distance: number;
  swim(): void;
}

interface Soundable {
  volume: number;
  makeSound(): void;
}

interface Diveable {
    depth: number;
    dive(): void
}

// klasa bazowa
class Animal {
  private name: string;

  constructor(_name: string) {
    this.name = _name;
  }
   protected introduceYourself(){
    console.log(`nazywam się ${this.name} :)`)
  }
}
```

interfejsy

klasa bazowa

implementacja interfejsów, dziedziczenie

app.ts

app.ts > Swimmable > swim

```ts
// klasa Duck dziedziczy po klasie bazowej Animal wszystkie jej pola i metody
// (można dziedziczyć tylko po jednej klasie)
// implementuje kilka interfejsów
class Duck extends Animal implements Flyable, Swimmable, Soundable, Diveable {
  distance = 1000;
  volume = 30;
  depth = 100;
  fly(): void {
    console.log("Potrafię latać! ");
  }
  swim(): void {
    console.log(`Potrafię przepłynąć ${this.distance} m`);
  }
  makeSound(): void {
      console.log(`Kwa kwa o głośności ${this.volume} dB`);
  }
  dive():void{
      console.log(`umiem nurkować na głebokość ${this.depth} m`);
  }
  introduce(){
    this.introduceYourself()
  }
}

let duck:Duck = new Duck("kaczka");
//duck.introduceYourself(); // błąd
duck.introduce();
duck.fly();
duck.swim();
duck.makeSound();
duck.dive();
```

klasa dziedzicząca po klasie bazowej implementująca interfejsy

```
nazywam się kaczka :)
Potrafię latać!
Potrafię przepłynąć 1000 m
Kwa kwa o głośności 30 dB
umiem nurkować na głebokość 100
```

wzorce projektowe

(ang. design patterns) – uniwersalne, sprawdzone w praktyce rozwiązania często pojawiających się, powtarzalnych problemów projektowych

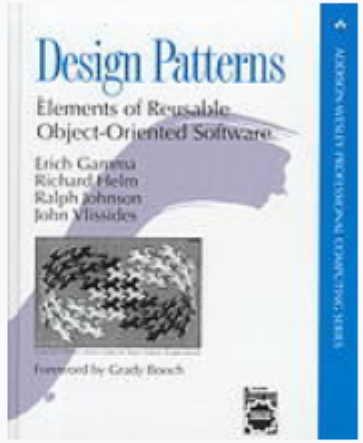
Design Patterns: Elements of Reusable Object-Oriented Software (1994) is a software engineering book describing software design patterns. The book was written by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, with a foreword by Grady Booch. The book is divided into two parts, with the first two chapters exploring the capabilities and pitfalls of object-oriented programming, and the remaining chapters describing 23 classic software design patterns. The book includes examples in C++ and Smalltalk.

It has been influential to the field of software engineering and is regarded as an important source for object-oriented design theory and practice. More than 500,000 copies have been sold in English and in 13 other languages.[1] The authors are often referred to as the *Gang of Four* (GoF).[2][3][4][5]

Development and publication history [edit]

The book started at a birds-of-a-feather session at the 1990 OOPSLA meeting,

Design Patterns: Elements of Reusable Object-Oriented Software



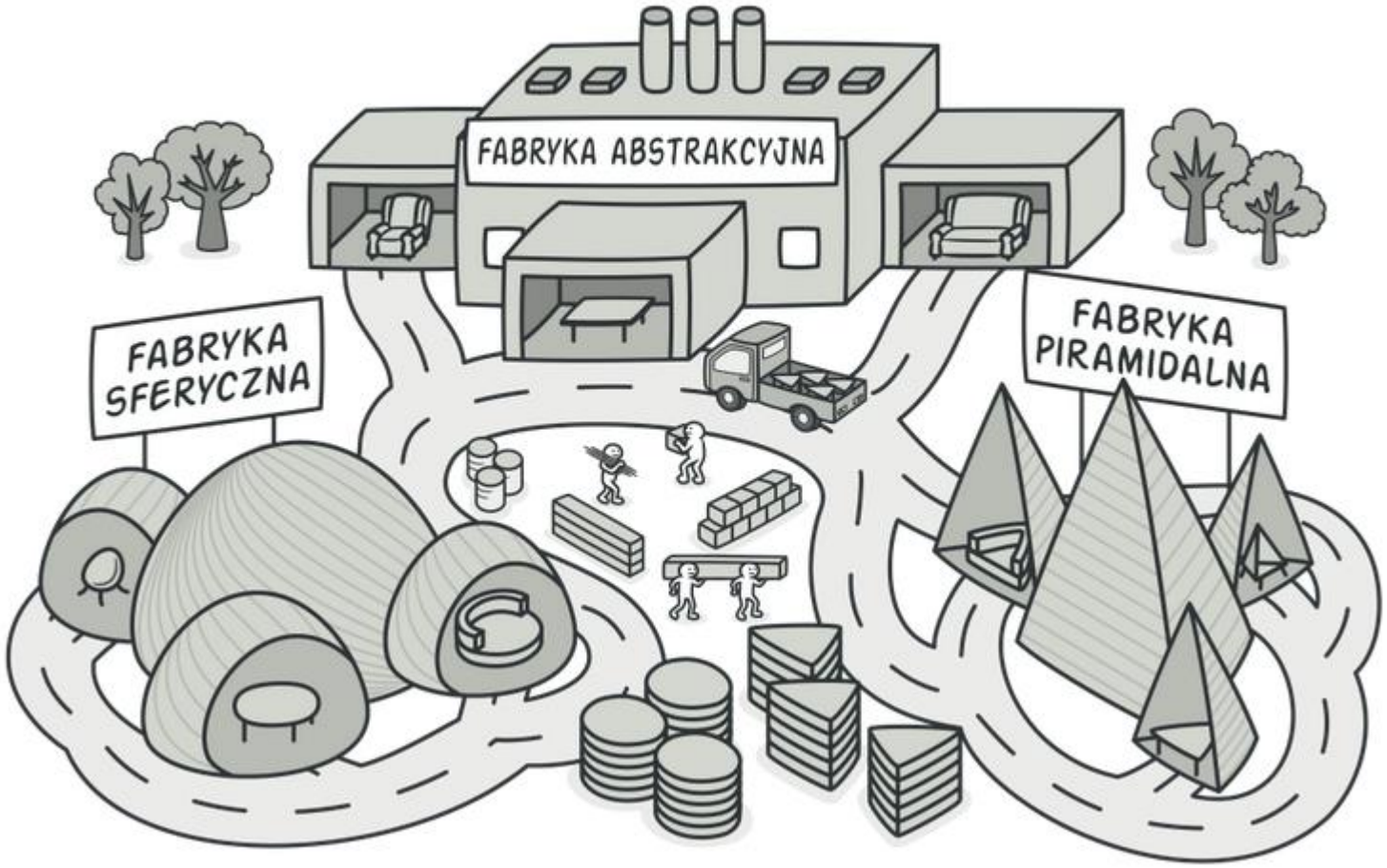
Author	Erich Gamma Richard Helm Ralph Johnson John Vlissides
Subject	Design patterns, software engineering, object-oriented programming

ważna książka

https://en.wikipedia.org/wiki/Design_Patterns
https://pl.wikipedia.org/wiki/Wzorzec_projektowy_(informatyka)

Fabryka abstrakcyjna

Fabryka abstrakcyjna jest kreacyjnym wzorcem projektowym, który pozwala tworzyć rodziny spokrewnionych ze sobą obiektów bez określania ich konkretnych klas.



https://refactoring.guru/pl/design-patterns/abstract-factory

Fabryka abstrakcyjna

app.ts

```typescript
//https://refactoring.guru/design-patterns/abstract-factory

interface Chair {
    name: string;
}

class ArtDecoChair implements Chair {
    name = "Jestem krzesło Art Deco";
}

class ModernChair implements Chair {
    name = "Jestem krzesło Modern";
}

interface FurnitureFactory {
    createChair(): Chair;
}

class Furniture {
    name: string = '';
}

class ArtDecoFurnitureFactory implements FurnitureFactory {
    createChair(): Chair {
        return new ArtDecoChair();
    }
}

class ModernChairFurnitureFactory implements FurnitureFactory {
    createChair(): Chair {
        return new ModernChair();
    }
}

class Client extends Furniture {
    constructor(
        _name: string,
        private furnitureFactory: FurnitureFactory,
    ) {
        super();  // uruchamia konstruktor klasy bazowej Furniture
        this.name = _name;

        console.log(
            `hello ${this.name}, wyprodukowaliśmy dla Ciebie: ${this.furnitureFactory.createChair().name}`
        );
    }
}
const artDecoFurnitureFactory: ArtDecoFurnitureFactory = new ArtDecoFurnitureFactory();
const modernChairFurnitureFactory: ModernChairFurnitureFactory = new ModernChairFurnitureFactory();

const artDecoClient: Client = new Client("artDecoClient", artDecoFurnitureFactory)
const modernChairClient: Client = new Client("modernChairClient", modernChairFurnitureFactory)
```

korzystając z kodu obok zaimplementować resztę mebli

Product families and their variants.

https://refactoring.guru/pl/design-patterns/abstract-factory

TypeScript

polecenia konsoli w Visual Studio Code dotyczące TypeScript w środowisku Node.js

```
cd  katalog_projektu        // przechodzimy do katalogu projektu
npm  –y  init               // inicjalizacja projektu w node.js
npm  install  typescript    // instalacja języka TypeScript w katalogu projektu
npx  tsc  –init             // inicjalizacja TypeScript
npx  tsc  hello.ts          // konwersja pliku hello.ts do języka JavaScript, w wyniku powstaje plik hello.js
node  hello.js              // uruchomienie pliku hello.js w środowisku node.js
npx  tsc  --watch           // włączenie trybu nasłuchu – automatyczną kompilację pliku hello.ts
                            // po wykryciu w nim zmiany
```