infoShare ACADEMY

TypeScript

Wprowadzenie do typowania kodu

infoShare Academy

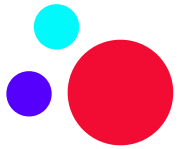
infoShareAcademy.com

HELLO

Jakub Wojtach

Senior **full stack** developer

infoShare ACADEMY



Rozgrzewka i agenda

Zacznijmy ten dzień z przytupem!

Rozgrzewka

JavaScript

Język chiński

Rozgrzewka

Komputer

Tablet

Rozgrzewka

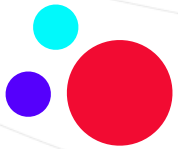
Windows

MacOS

Rozgrzewka

CD

Winyl

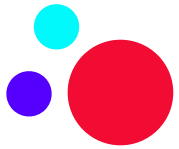


Agenda

- **Podstawy** teorii
- **Podstawy** użycia TypeScript - TypeScript playground
- **Konfiguracja** TypeScript
- **Praktyczne** wykorzystanie typowania statycznego

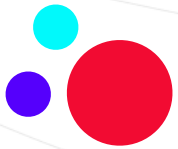
Współpraca

- Zadajemy pytania w dowolnym momencie - kanał **merytoryka**
- Krótkie przerwy (**5 min**) co godzinę
- Długa przerwa (**20 min**) po ostatnim bloku



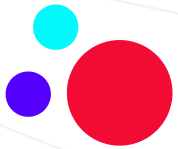
Podstawy teorii

infoShareAcademy.com



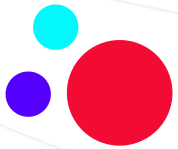
Podstawowa teoria dotycząca typowania

- Typowanie to proces określania typu danych wejściowych, wyjściowych i zmiennych w języku programowania.
- Typowanie jest procesem określania **typu danych**, których używają *zmienne*, *funkcje* i *inne* elementy programu.
- Teoria typowania zawiera podstawy tego, jak systemy typów powinny działać, jakie typy powinny być wspierane i jak powinny być używane.
- TypeScript jest **językiem programowania** tworzonym w modelu *open-source* będący **nadzbiorem** JavaScriptu.
- Główny zespół odpowiadający za jego utrzymanie pracuje w **Microsoft**.
- Można powiedzieć też, że każdy kod napisany w JavaScript możemy skopiować do pliku .ts (rozszerzenie dla plików TypeScript) i wszystko będzie działać poprawnie.



Podstawowa teoria dotycząca typowania

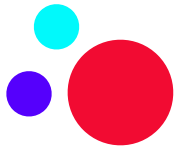
- **Nie** jest to jednak do końca prawda, przynajmniej **nie** przy **domyślnych** ustawieniach kompilatora TypeScript.
- Jeśli zluzujemy nieco różne obostrzenia to dopiero wówczas **KAŻDY** kod JavaScript uruchomi się jako kod TypeScript.
- **TypeScript** jest **kompilowany** do czystego **JavaScriptu**. (wspomniane typy, **najważniejsza cecha TypeScript**, nie występują w trakcie działania programu, czyli podczas *runtime*).
- Jeśli skompilujemy kod z pliku **.ts** do pliku .js i uruchomimy w przeglądarce to **nie będzie miała ona pojęcia**, jakie **typy** gdzie ustawiliśmy, bo one **nie istnieją** w ECMAScript.



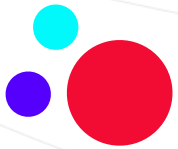
```typescript
class Greeter {

    private name: string;

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

    public greet(): string {
        return `Hello ${this.name}!`;
    }

}


console.log(new Greeter("Pawel").greet());
```

i jest on kompilowany do poniższego kodu JavaScript

```javascript
var Greeter = /** @class */ (function () {
    function Greeter(name) {
        this.name = name;
    }
    Greeter.prototype.greet = function () {
        return "Hello " + this.name + "!";
    };
    return Greeter;
}());
console.log(new Greeter("Pawel").greet());
```

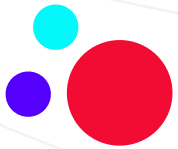


Typowanie dynamiczne vs statyczne



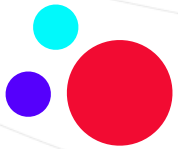
Typowanie dynamiczne vs statyczne

- Typowanie **statyczne** polega na tym, że typy są określane i sprawdzane **przed** uruchomieniem programu, natomiast typowanie **dynamiczne** oznacza, że typy są określane **w czasie** wykonywania programu.



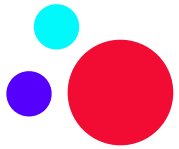
Typowanie dynamiczne vs statyczne **różnice**

- **Określanie typów**: Typowanie dynamiczne określa typy w czasie wykonywania programu, natomiast typowanie statyczne określa typy przed jego uruchomieniem.
- **Wydajność**: Typowanie statyczne jest zwykle szybsze od typowania dynamicznego, ponieważ typy są określone w czasie kompilacji, a nie podczas działania programu.
- **Błędy**: Typowanie statyczne umożliwia wykrycie wielu błędów w czasie kompilacji, co zmniejsza ryzyko wystąpienia błędów podczas działania programu. W przypadku typowania dynamicznego błędy są wykrywane tylko w czasie działania programu.



Typowanie dynamiczne vs statyczne **różnice**

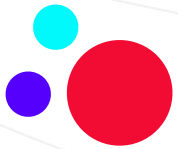
- **Elastyczność:** Typowanie dynamiczne jest bardziej elastyczne niż typowanie statyczne, ponieważ umożliwia tworzenie i przypisywanie wartości zmiennym w czasie działania programu. Typowanie statyczne jest mniej elastyczne, ponieważ typy muszą być określone przed uruchomieniem programu.
- **Dokumentacja:** Typowanie statyczne umożliwia łatwiejsze i szybsze tworzenie dokumentacji kodu, ponieważ typy są jasno określone. W przypadku typowania dynamicznego dokumentacja jest trudniejsza do utworzenia i wymaga dodatkowej pracy.



Typowanie słabe i silne

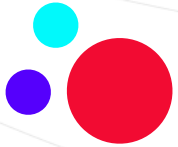
infoShareAcademy.com

infoShare ACADEMY



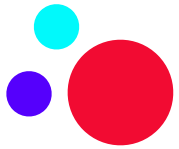
Typowanie słabe i silne

- Typowanie **silne** i **słabe** to terminy odnoszące się do sposobu kontrolowania typów zmiennych w języku programowania TypeScript.
- Typowanie **słabe** pozwala na automatyczną konwersję typów w czasie wykonywania programu, co może prowadzić do błędów i niezamierzonych rezultatów.
- Typowanie **silne** natomiast jest bardziej rygorystyczne i uniemożliwia niejawne koercje typów, co zapewnia większą kontrolę i stabilność w kodzie.
- TypeScript jest językiem programowania z **silnym systemem typów**, co oznacza, że **wymaga** jawnego określenia typów zmiennych.



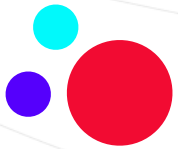
Typowanie słabe i silne

- TypeScript jest także językiem z typami **inferowanymi**, co oznacza, że kompilator **automatycznie** określa typy na podstawie **wartości przypisanych** do **zmiennych**.
- Słabe typowanie w kontekście TypeScript oznacza, że programista **może** używać słabych typów, ale jest to uważane za **niezalecane**.
- TypeScript pozwala także na zastosowanie **duck typing**, co oznacza, że określa typy na podstawie właściwości i metod obiektu, a nie jego nazwy.
- Typowanie nominalne jest kolejną opcją dostępną w TypeScript, pozwala ono na określenie typów na podstawie nazwy i jest podobne do typowania w językach z rodziny C.



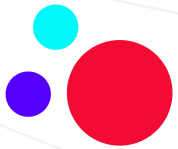
Typowanie strukturalne, nominalne i duck-typing

infoShareAcademy.com



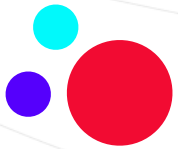
Typowanie strukturalne, nominalne i duck-typing

- Typowanie **strukturalne** w TypeScript oznacza, że typy są porównywane na podstawie ich struktury, a nie nazwy.
- Typowanie **nominalne** w TypeScript oznacza, że typy są porównywane na podstawie ich nazwy.
- **Duck typing** oznacza, że obiekty są typowane na podstawie ich **interfejsu**, a nie ich nazwy lub struktury.

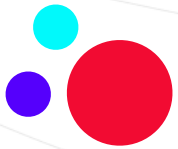


Typowanie strukturalne

- Jest to główne podejście do typowania w TypeScript.
- Pozwala na używanie **obiektów** o różnych nazwach, ale tej samej strukturze.
- Typowanie strukturalne jest bardziej elastyczne niż typowanie nominalne.
- TypeScript wychodzi z założenia, że kompatybilność typów powinna być określona przez ich funkcjonalność, a nie nazwę.

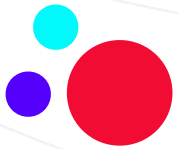


```ts
type Person = {
  name: string;
  DOB: Date;
};

type Employee = {
  name: string;
  DOB: Date;
};

const person: Person = {
  name: 'Buzz Lightyear',
  DOB: new Date(1953, 5, 13)
};

const employee: Employee = person;

```



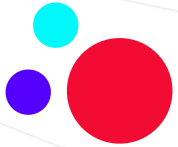
Typowanie nominalne

- Typowanie **nominalne** w TypeScript polega na porównywaniu typów na podstawie ich **nazwy**.
- Typy są uważane za kompatybilne **tylko wtedy**, gdy posiadają **tę samą nazwę**.
- Typowanie **nominalne** jest **bardziej restrykcyjne** niż typowanie **strukturalne**.
- TypeScript zapewnia opcję stosowania typowania nominalnego poprzez użycie operatora "**instanceof**" lub stosowanie **interfejsów**.
- Typowanie nominalne jest **często stosowane** w przypadku, **gdy wymagana jest jasna i ścisła kontrola nad kompatybilnością typów**.



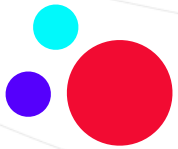
Typowanie nominalne

```
class A {
  name: string
}

class B {
  name: string
}

let a: A

// Error type A is not the same as type B, the names are different
a = new B()
```



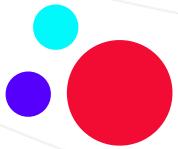
Typowanie duck-type

- Typowanie duck-typing w TypeScript opiera się na koncepcie, że "*jeśli wygląda jak kaczka i kwacze jak kaczka, to jest kaczką*".
- Typy są uważane za kompatybilne, jeśli posiadają **takie** same **właściwości i metody**.
- Typowanie duck-typing jest **bardziej** elastyczne niż typowanie nominalne i strukturalne.

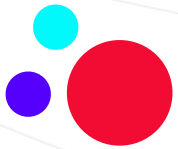


Typowanie duck-type

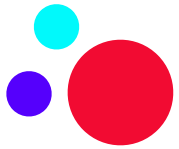
- Pozwala na stosowanie obiektów **bez konieczności** użycia interfejsów czy określania ich typów.
- TypeScript udostępnia opcję stosowania typowania duck-typing, jednak **domyślnie** stosuje **typowanie strukturalne**.
- Typowanie duck-typing jest szczególnie przydatne w przypadku kodu, który jest bardziej **dynamiczny** i wymaga **dużej elastyczności**.



```
class Sparrow {
    sound = "cheep";
}
class Parrot {
    sound = "squawk";
}
class Duck {
    sound = "quack";
    swim(){
        console.log("Going for a dip!");
    }
}
var parrot: Parrot = new Sparrow(); // substitutes
var sparrow: Sparrow = new Parrot(); // substitutes
var parrotTwo: Parrot = new Duck();
//var duck: Duck = new Parrot(); // IDE & compiler error

console.log("Parrot says: "+parrot.sound);
console.log("sparrow says: "+sparrow.sound);
console.log("parrot says: "+parrotTwo.sound);
```



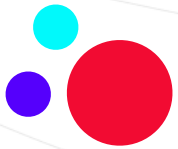
- TypeScript jest językiem programowania o typach statycznych, silnych z pewnymi odstępstwami, strukturalnych, inferowanych, wykorzystującym duck typing i pozwalającym na zastosowanie typowania nominalnego.
- Jego głównym celem jest umożliwienie tworzenia dużych i złożonych aplikacji JavaScript poprzez dodanie systemu typów i narzędzi do wspomagania programowania.



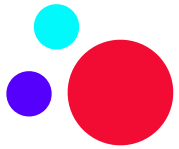
Podstawy użycia Typescript

TypeScript Playground

infoShareAcademy.com



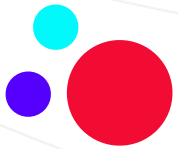
- Typy elementarne
  - Prymitywy**: boolean, bigint, null, number, string, sym bol, undefined**
  - Złożone: **object, Array**
  - Typy specyficzne dla TS: **any, enum, never, Object, unknown, tuple, void**



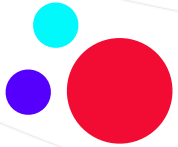
Definiowanie własnych typów

infoShareAcademy.com

infoShare ACADEMY

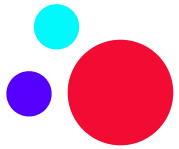


- W TypeScript można definiować własne typy za pomocą słów kluczowych "type" i "interface".
- "Type" pozwala na utworzenie aliasów dla już istniejących typów, np. "type MyString = string".



- **"Interface"** jest bardziej zaawansowanym narzędziem do definiowania nowych typów, które mogą zawierać wiele właściwości i ich typy, np. "interface MyType { prop1: string, prop2: number }".
- **Interfejsy** mogą być implementowane przez **klasy**, co umożliwia określenie wymagań dla ich implementacji.
- Typy utworzone przy użyciu "**type**" i "**interface**" są traktowane jak inne typy i mogą być używane jako **argumenty** funkcji, **typy zwracane** itd.
- Oba narzędzia pozwalają na **lepsze** opisanie **typów danych** w kodzie, co ułatwia jego czytelność i unika błędów.

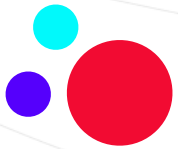
Zadanie



Typowanie funkcji

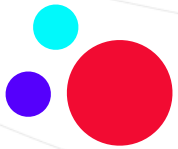
infoShareAcademy.com

infoShare ACADEMY



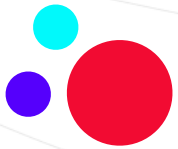
Typowanie funkcji **argumenty**

- W TypeScript, argumenty funkcji mogą być określone jako określonego typu, takiego jak string, number lub jakikolwiek inny typ zdefiniowany przez użytkownika.
- Typowanie argumentów funkcji zapewnia większą jasność i łatwość w debugowaniu, ponieważ wszelkie nieprawidłowe wywołania funkcji z błędnie zdefiniowanymi argumentami są wykrywane przez TypeScript w czasie kompilacji.
- Typowanie argumentów funkcji jest **opcjonalne** w TypeScript, ale **zalecane** w celu uzyskania większej jakości i bezpieczeństwa kodu.



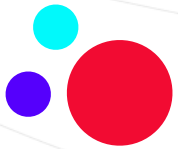
Typowanie funkcji **typ zwracany**

- W TypeScript zwracany przez funkcje typ jest adnotacją typu, która określa typ zwracany przez funkcję.
- Służy do wskazania typu wartości, którą funkcja ma zwrócić.
- Bez adnotacji zwracanego typu funkcja domyślnie zwróci typ „any”.



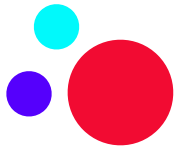
Typowanie funkcji jako **type**

- https://www.typescriptlang.org/docs/handbook/2/functions.html#function-type-expressions

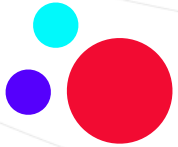


Typowanie funkcji **parametry opcjonalne i domyślne**

- Parametry opcjonalne w TypeScript są oznaczone za pomocą symbolu "?". Dzięki temu funkcja nie wymaga ich przekazywania jako argumentu.
- Możesz ustawić wartość domyślną dla parametru, która zostanie użyta, jeśli nie zostanie przekazany jako argument.
- TypeScript automatycznie interpretuje typy danych dla parametrów domyślnych i opcjonalnych, ale możesz również jawnie określić typ danych.
- Użycie parametrów opcjonalnych i domyślnych pozwala na bardziej elastyczne i przejrzyste definiowanie funkcji, co ułatwia ich użycie w różnych sytuacjach.

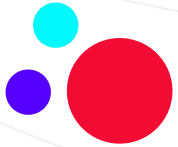


Część wspólna i suma typów



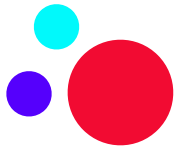
Union

- Union type w TypeScript pozwala na określenie kilku typów, które mogą być przypisane do tej samej zmiennej.
- Union type jest używany, gdy wartość zmiennej może być jednym z kilku różnych typów.
- Union type jest wyrażony przez "|" pomiędzy dwoma lub więcej typów, np. "string | number | boolean".



Intersection

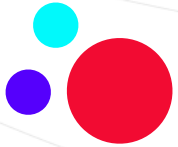
- Intersection type w TypeScript pozwala na połączenie kilku typów w jeden typ, który łączy w sobie właściwości wszystkich typów.
- Intersection type jest używany, gdy wartość zmiennej jest jednocześnie kilku typów.
- Intersection type jest wyrażony przez "&" pomiędzy dwoma lub więcej typów, np. "string & {length: number}".



Inferencja typów

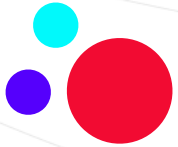
infoShareAcademy.com

infoShare ACADEMY



Inferencja typów

- Na pierwszy rzut oka, mogłoby się wydawać, że TypeScript zmusza nas do pisania znacznie dłuższego kodu przez konieczność dopisywania typów w każdym miejscu – nawet tym najbardziej oczywistym.
- **Nic bardziej mylnego**. W wielu miejscach kompilator (jak i duża część IDE wspierających TS) jest w stanie *„domyślić się”* jaki jest typ zmiennej.
- Przykładowo w poniższym zapisie określanie typu jest zbędne:



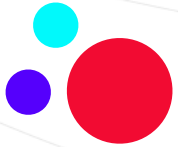
Inferencja typów

```
const title: string = `Wartosc 2 + 2 wynosi 4`;
```

- Jeżeli od razu przypisujemy wartość do zmiennej, w większości przypadków typ powinien zostać rozpoznany sam, tak więc ten zapis jest równoważny:

```
const title = `Wartosc 2 + 2 wynosi 4`;
```

- Kiedy nie określimy typu i nie zostanie on rozpoznany, ustawiany jest typ **any**.

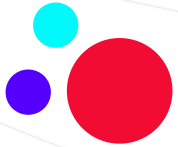


Inferencja typów zwracanych z funkcji

- W przypadku funkcji, inferencja typów zwracanych pozwala na automatyczne określenie typu wartości, która jest zwracana przez funkcję.
- Inferencja typów zwracanych jest szczególnie przydatna w przypadku funkcji, które zwracają różne typy wartości w zależności od wywołania, ponieważ pozwala uniknąć ręcznego określania typu zwracanego.
- Warto zauważyć, że inferencja typów zwracanych nie jest w stanie w pełni zastąpić ręcznego określania typów zwracanych, szczególnie w przypadku skomplikowanych funkcji, które zwracają wartości o złożonych typach. W takich przypadkach nadal zaleca się ręczne określanie typów zwracanych, aby zapewnić jak największą dokładność i jakość kodu.

Inferencja kontekstowa

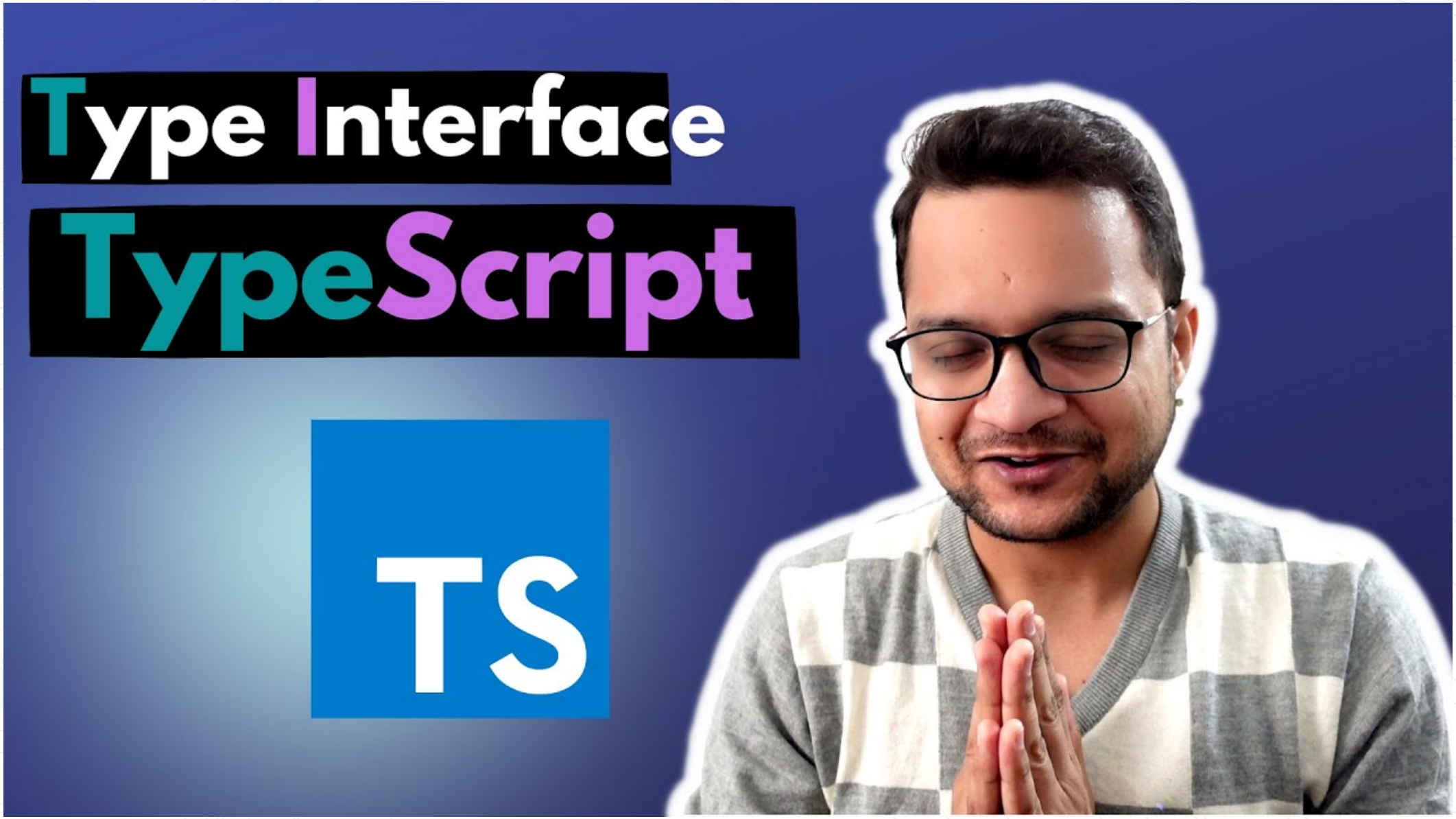
- Inferencja kontekstowa w TypeScript to mechanizm automatycznego określania typów danych na podstawie ich kontekstu użycia.
- Inferencja kontekstowa polega na analizie kodu i jego związku z innymi elementami, takimi jak zmienne, funkcje lub obiekty, aby określić jakiego typu dane powinny być przypisane do danego elementu.
- Inferencja kontekstowa jest bardzo przydatna, ponieważ pozwala na zmniejszenie ilości ręcznie określanych typów, co z kolei prowadzi do bardziej czytelnego i łatwiejszego do utrzymania kodu.
- Inferencja kontekstowa jest szczególnie użyteczna w przypadku kodu, który jest złożony i wymaga wiele ręcznie określanych typów, ponieważ pozwala na automatyczne określanie tych typów na podstawie kontekstu użycia. W rezultacie, kod jest bardziej zoptymalizowany i łatwiejszy do utrzymania.

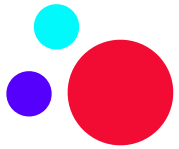


Inferencja ograniczenia

- Inferencja typów w TypeScript ma pewne ograniczenia, które powodują, że nie zawsze jest w stanie prawidłowo określić typ danych.
- W przypadku tablic, błędne inferowanie typu może wystąpić, gdy tablica jest inicjowana jako pusta, a następnie dodawane są do niej elementy różnych typów. W takim przypadku, TypeScript może zgadywać nieprawidłowy typ danych dla całej tablicy.
- Dlatego ważne jest, aby ręcznie określać typy danych w przypadku skomplikowanych przypadków, takich jak tablice zawierające elementy różnych typów, aby zapewnić jak największą dokładność i jakość kodu. W ten sposób, można uniknąć błędnego inferowania typów, które może prowadzić do problemów z kompilacją lub działaniem aplikacji.

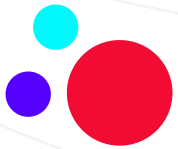
Inferencja typów





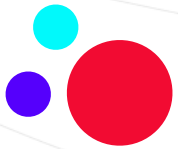
Konfiguracja typescript

infoShareAcademy.com



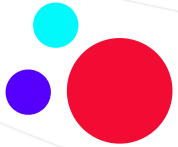
Konfiguracja typescript

- Paczka typescript
- Przygotowanie tsconfig.json
- Tryb strict
- Kompilacja kodu .ts do .js
- Vite



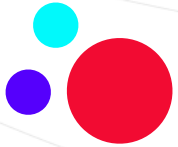
Typesctipt paczka NPM

- Paczka typescript jest głównym narzędziem do tworzenia i kompilacji projektów napisanych w języku TypeScript.
- Można go zainstalować globalnie na komputerze lub jako zależność w projekcie za pomocą narzędzia do zarządzania pakietami npm lub yarn.
- Pakiet TypeScript zawiera narzędzie tsc (TypeScript Compiler), a także inne narzędzia i biblioteki niezbędne do tworzenia i uruchamiania aplikacji w języku TypeScript. Jego instalacja jest konieczna, aby móc korzystać z możliwości języka i narzędzi do jego tworzenia.



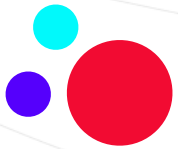
tsconfig.json

- ***tsconfig.json*** jest plikiem konfiguracyjnym dla TypeScript i określa opcje kompilacji dla kodu napisanego w języku TypeScript.
- Można w nim ustawić opcje takie jak target JavaScript, opcje modyfikacji kodu po kompilacji i określić pliki lub foldery, które mają zostać uwzględnione lub pominięte podczas kompilacji.
- Plik tsconfig.json jest **wymagany** w root projektu TypeScript, aby narzędzia takie jak **tsc** (kompilator TypeScript) wiedziały, jak skonfigurować proces kompilacji



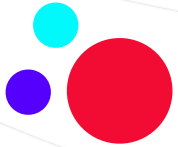
Tryb strict

- Tryb strict w TypeScript jest opcjonalnym trybem kompilacji, który zwiększa poziom bezpieczeństwa i jakości kodu.
- Tryb strict wymusza na programiście stosowanie dobrych praktyk i zapobiega potencjalnym błędom w czasie kompilacji, takim jak niejednoznaczne typy lub nieprzypisane zmienne.
- Aby włączyć tryb strict, należy ustawić opcję "strict": true w pliku tsconfig.json. Włączenie trybu strict może wymagać ręcznej poprawki kodu, ale jest to inwestycja, która zwiększa jakość i niezawodność aplikacji.



tsc czyli kompliacja TS do JS

- tsc (**TypeScript Compiler**) jest oficjalnym narzędziem do kompilacji kodu napisanego w języku TypeScript do kodu JavaScript.
- tsc automatycznie wykrywa plik **tsconfig.json** w root projektu i korzysta z jego ustawień, aby skonfigurować proces kompilacji.
- Użycie tsc pozwala programiście na tworzenie aplikacji w języku TypeScript i uruchamianie ich w środowiskach, które nie obsługują TypeScript bezpośrednio, jednocześnie umożliwiając korzystanie z wszystkich zaawansowanych funkcji języka.



Konfiguracja typescript

🚀 Rozpoczęcie pracy z TS

Zaczynamy od utworzenia pliku `package.json` za pomocą komendy:

```
npm init -y
```

Następnie przygotowujemy strukturę katalogową która powinna zawierać plik `index.html` i katalog `src` który będzie zawierał pliki z rozszerzeniem `.ts`

Możemy na tym etapie zainstalować *TypeScript* w naszym projekcie wpisując:

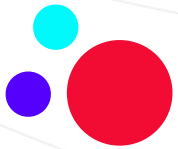
```
npm i typescript
```

Konfiguracja typescript

W projekcie dodajemy plik konfiguracyjny o nazwie `tsconfig.json` którego zawartość może być na początku następująca:

```
{
  "compilerOptions": {
    "rootDir": "./src",
    "outDir": "./dist",
    "target": "ES2015"
  }
}
```

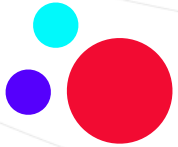
Powyższy kod określa, że pliki źródłowe z rozszerzeniem `.ts` znajdują się w katalogu `src`, a pliki `.js` które powstaną w wyniku kompilacji umieścimy w katalogu `dist`. Klucz `target` pozwala nam określić sposób kompilacji np. jeżeli używamy najnowszych konstrukcji języka JS i chcemy aby wygenerowany wynikowy kod był np. kompatybilny wstecz ze starszymi przeglądarkami to możemy określić standard *ES* który nas interesuje.



Konfiguracja typescript

Pozostało już tylko dodanie skryptu który umożliwi wykorzystanie kompilatora *TypeScripta* (skrypt dodajemy w pliku `package.json` w sekcji `scripts`):

```
"build": "tsc"
```



Konfiguracja Vice i CRA

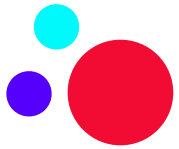
🚀 Użycie TS'a w *Vite* i *CRA*

W przypadku *Vite* projekt wykorzystujący *TypeScript* możemy utworzyć za pomocą:

```
npm init vite@latest . -- --template react-ts
```

Z kolei w przypadku *Create React App* wygląda to następująco:

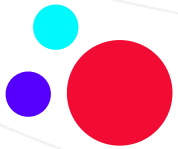
```
npx create-react-app my-app --template typescript
```



Praktyka typescript

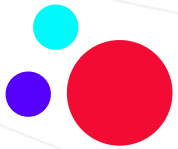
infoShareAcademy.com

infoShare ACADEMY



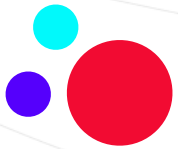
Wprowadzenie typów do aplikacji

```
const cars: unknown[] = [
  {
    model: 'Q7',
    brand: 'Audi',
    year: 2004,
  },
  {
    model: '320',
    brand: 'BMW',
    year: 1992,
  },
  {
    model: '6',
    brand: 'Mazda',
    year: 2018,
  },
];

function carsAfter2000(cars: unknown): unknown {
  return cars.filter((car) => car.year > 2000);
}

const newCars = carsAfter2000(cars);
```



Typowania ciąg dalszy

- https://paste.ofcode.org/5fp3UXTQzV8ECCGRw6254N



Linki dla chętnych

- https://javascript.plainenglish.io/what-is-duck-typing-in-typescript-c537d2ff9b61
- https://typescript-exercises.github.io/
- https://github.com/typescript-cheatsheets/react#types-or-interfaces
- https://create-react-app.dev/docs/adding-typescript/
- https://medium.com/opensanca/migrating-from-js-to-ts-cra-b5f679086c5a

Dziękuję za uwagę

Jakub Wojtach