# TYPESCRIPT

## Установка:

1. Глобально установи компилятор TS командой:
npm i -g typescript
2. Проверь версию:
tsc -v
3. Создай файл конфигурации tsconfig.json:
tsc —init
3. Создай TypeScript-файл (index.ts) и напиши код. Компиляция производится следующей командой:
tsc index или tsc -w

## Основные особенности:

- Статическая типизация;
- Безопасный и устойчивый код;

*TypeScript поддерживает 7 примитивных типов JS: string, number, bigInt, boolean, undefined, null, symbol в виде:*

```
let name: string = 'John'
```

***unknown*** *– для знач. тип, которых неизвестен во время компиляции*
***any*** *– любой тип данных*
***void*** *– отсутствие значения*
***never*** *– для функции, которая не завершится или сгенерирует ошибку*

## UnionTypes

*позволяет объединить несколько типов данных, чтобы переменная могла принимать значения любого из этих типов.*

```
let age: number | string ;
age = 26;
age = '26'
```

## Динамическая типизация

```
let age: any = 100;
age = true;
```

## Объекты

*Объекты в TypeScript должны иметь корректные свойства и значения типов:*

```
let person = {
  name: string;
  isProgrammer: boolean;
}

person.age = 26 // Error: no age prop on person object
person.isProgrammer = 'yes' // Error – should be boolean
```

## Массивы

*Мы должны определить какой тип данных может использоваться:*

```
let ids: number[] = []
ids.push(1)
ids.push('1') // Error
```

*Используйте **UnionTypes** для массивов с различными типами:*

```
let options: (string | number)[];
options = [10, 'up']
options = [10, 'up', true] // Error – only strings or numbers allowed
```

## Функции

*Мы можем типизировать аргументы функции и вернуть определенный тип данных.*

```
function circle(diam: number): string {
  return 'Circle is' + Math.PI * diam;
}
```

*Та же функция – стрелка:*

```
const circle = ( diam : number) : string => {
  return 'Circle is' + Math.PI * diam;
}
```

*Если мы хотим создать функцию, которая не возвращает значения:*

```
let sayHi : (name: string) => void ;
sayHi : (namt: string) => console.log('Hi' + name)
sayHi('Danny') // 'Hi Danny'
```

## Tuples (Кортежи)

*массивы с фиксированным числом элементов и известными типами.*

```
let options: [string, number];
options = ['up', 10]
```

## Aliases (Псевдоним типа)

*Они помогают определить новое имя типа, используются для описания примитивных типов и переиспользуемости:*

```
type Score = number | string
const myScore: Score = 7
```

## Enum (Перечисление)

*позволяет определить набор именованных констант.*

```
enum Color {
  Red,
  Green,
  Blue,
}

let backgroundColor = Color.Red;
```

## TypeGuards (Охранник типа)

*механизм, который позволяет проверять типы значений во время выполнения программы и обеспечивает безопасность типов*

```
function isString(value: any): value is string {
  return typeof value === "string";
}

let value: any = "hello";

if (isString(value)) {
  console.log(value.toUpperCase());
  // Вывод: HELLO
} else {
  console.log("Value is not a string");
}
```

*Typescript поддерживает перезагрузку функций overload – это способ определить функцию с несколькими вариантами типов и реализаций.*

## Interface

*определяет свойства и методы, которые объект должен реализовать.*

```
interface IEmployee {
  empCode: number;
  empName: string;
  getSalary: (number) => number;
  getManagerName(number):
string;
}
```

*Интерфейсы определяются для типов данных object, могу наследоваться и расширяться.*

## types

*С помощью types мы определяем типы данных, а также можем использовать функционал недоступный в интерфейсах: создание unionTypes, aliases и др. возможностей.*

```
type Person = {
  name: string;
  age: number;
  email?: string;
}
```

*Types используются для любых типов данных, их можно объединять ( | ) и наследовать, но нельзя расширять за счет других типов.*

## Omit

*позволяет вам создать тип, передав текущий тип и выбрав ключи, которые нужно пропустить в новом типе.*

```
Omit<Type, Keys>;

interface Todo {
  title: string;
  description: string;
  completed: boolean;
  createdAt: number;
}

type TodoPreview = Omit<Todo, "description">;
```

## Generics (Общие типы)

*механизм, который позволяет создавать компоненты, которые могут работать с различными типами данных, не зависимо от конкретного типа.*

```
function identity<T>(arg: T): T {
  return arg;
}

let result = identity<string>('Hello, world!');
```

*Generics, Index access Types и keyOf*
*Обобщенная функция, принимающая объект и ключ свойства:*

```
function getProperty<T, K extends keyof T>(obj: T, key: K): T[K] {
  return obj[key];
}

const user = {
  name: 'John',
  age: 30,
  email: 'john@example.com'
};

const userName = getProperty(user, 'name');
console.log(userName);

const userAge = getProperty(user, 'age');
console.log(userAge);

const userEmail = getProperty(user, 'email');
console.log(userEmail);
```

*Используя концепции Generics, Index Access Types и keyof, мы можем обобщить функцию getProperty для работы с различными типами объектов и ключами свойств, обеспечивая статическую типизацию и безопасность при работе с объектами*