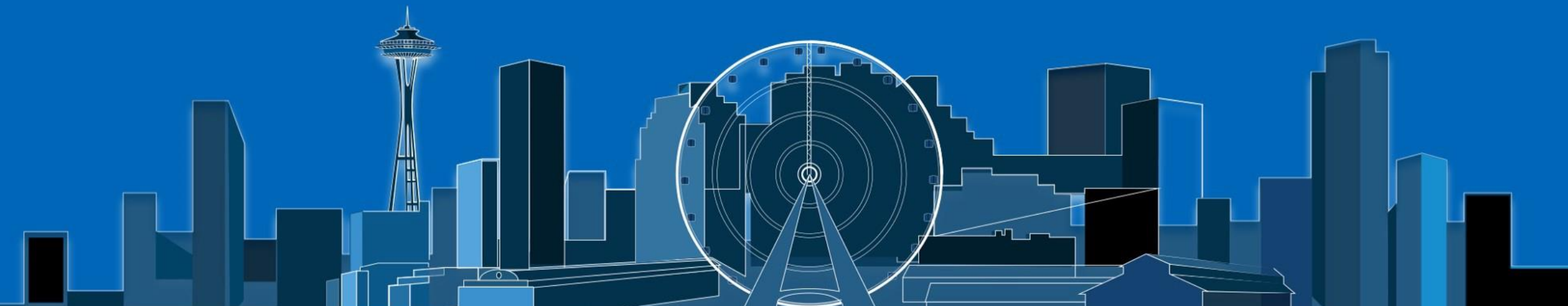
TypeScript



TypeScript

- представлен Microsoft в 2012 году
- увеличивает эффективность и надёжность JavaScript
- взят за основу в некоторых современных фреймворках, таких как Angular.

Основы TypeScript

Переменные и константы

- Переменные

  let hello = "Hello!";

- Константы

  const numLivesForCat = 9;

var vs let/const

var	let/const

`function print(){`
`  if(1) {`
`  var x = 94;`
`  }`
`  console.log(x); // норм`
`}`
	`function print(){`
	`  if(1) {`
	`  let x = 94;`
	`  }`
	`  console.log(x); // ! Ошибка`
	`}`

Доступна в любой части функции, в которой она определена.	Доступна только в рамках блока, в котором она определена

var vs let/const

var	let/const

`function print(){`
`  console.log(x); // undefined,`
`  var x = 76;`
`}`
	`function print(){`
	`  console.log(x); // ! Ошибка`
	`  let x = 76;`
	`}`

Можно использовать в функции перед определением.	Можно использовать только после определения.

var vs let/const

var	let/const

`function print(){`
`  var x = 72;`
`  console.log(x); // 72`
`  var x = 24;  // норм`
`  console.log(x); // 24`
`}`
	`function print(){`
	`  let x = 72;`
	`  console.log(x); // 72`
	`  let x = 24;  // ! Ошибка`
	`  console.log(x);`
	`}`

В одной и той же функции можно несколько раз определить переменную с одним и тем же именем.	В одной и той же функции можно только один раз определить переменную с одним и тем же именем.

Типы данных

- **Boolean**: логическое значение true или false
- **Number**: числовое значение
- **String**: строки
- **Array**: массивы
- **Tuple**: кортежи
- **Enum**: перечисления
- **Any**: произвольный тип
- **Null и undefined**: соответствуют значениям null и undefined в javascript
- **Void**: отсутствие конкретного значения, используется в основном в качестве возвращаемого типа функций
- **Never**: также представляет отсутствие значения и используется в качестве возвращаемого типа функций, которые генерируют или возвращают ошибку

Переменная с типом

```
let x: number = 10;
let hello: string = "hello world";
let isValid: boolean = true;

let x; // тип any
x = 10;
```

Шаблоны строк

```
let firstName: string = "Tom";
let age: number = 28;
let info: string = `Имя ${firstName} Возраст: ${age}`;
console.log(info); // Имя Tom Возраст: 28
```

Массивы

```
let list: number[] = [10, 20, 30];
let colors: string[] = ["red", "green", "blue"];
console.log(list[0]);
console.log(colors[1]);

let names: Array<string> = ["Tom", "Bob", "Alice"];
console.log(names[1]); // Bob
```

Кортежи

```
let userInfo: [string, number];

userInfo = ["Tom", 28];
//userInfo = [28, "Tom"]; // Ошибка

console.log(userInfo[1]); // 28
userInfo[1] = 37;
```

Тип enum

```
enum Season { Winter, Spring, Summer, Autumn };
let current: Season = Season.Summer;
console.log(current);
current = Season.Autumn;
-----------------------------------------------
enum Season { Winter=0, Spring=1, Summer=2, Autumn=3 };
var current: string = Season[2]; // 2 - значение Summer
console.log(current); // Summer
```

Объединения типов

```
let id : number | string;

id = "1345dgg5";
console.log(id); // 1345dgg5

id = 234;
console.log(id); // 234
```

Проверка типа

```
let sum: any;
sum = 1200;
if (typeof sum === "number") {
     let result: number = sum / 12;
     console.log(result);
}
else{
     console.log("invalid operation");
}
```

Псевдонимы типов

```
type stringOrNumberType = number | string;

let sum: stringOrNumberType = 36.6;

if (typeof sum === "number") {
    console.log(sum / 6);
}
```

Преобразования типов

```
let someValue: any = "this is a string";
let strLength: number =
          (<string>someValue).length;

let someValue: any = "this is a string";
let strLength: number =
          (someValue as string).length;
```

Определение функции

```
// определение функции
function add(a: number, b: number): number {
    return a + b;
}

// вызов функции
let result1 = add(1, 2);
console.log(result1);
```

Функция ничего не возвращает

```
function add(a: number, b: number): void {
    console.log(a + b);
}

add(10, 20);
```

Необязательные параметры

```
function getName(firstName: string, lastName?: string) {
      if (lastName)
            return firstName + " " + lastName;
      else
            return firstName;
}

let name1 = getName("Иван", "Кузнецов");
let name2 = getName("Вася");
```

Параметры по умолчанию

```
function getName(
                firstName: string,
                lastName: string="Иванов") {

    return firstName + " " + lastName;
}
let name1 = getName("Иван", "Кузнецов");
let name2 = getName("Вася");
```

Перегрузка функций

```
function add(x: string, y: string): string;
function add(x: number, y: number): number;
function add(x: any, y: any): any {
    return x + y;
}

let result1 = add(5, 4); // 9
let result2 = add("5", "4"); // 54
```

Тип функции

```
function sum (x: number, y: number): number {
    return x + y;
};

let op: (x:number, y:number) => number;

op = sum;
console.log(op(2, 4)); // 6
```

Функции обратного вызова

```
function mathOp(x: number, y: number,
    operation: (a: number, b: number) => number): number{
 let result = operation(x, y);
 return result;
}

let func: (x: number, y: number) => number;

func = function (a: number, b: number): number {
      return a + b;
}
console.log(mathOp(10, 20, func)); // 30
```

Стрелочные функции

- `let sum = (x: number, y: number) => x + y;`
- `let sub = (x, y) => x - y;`
- `let square = x => x * x;`
- `let hello = () => "hello world"`

Стрелочные функции

```
function mathOp(x: number, y: number,
    operation: (a: number, b: number) => number): number{

        let result = operation(x, y);
        return result;
}

console.log(mathOp(10, 20, (x,y)=>x+y)); // 30
console.log(mathOp(10, 20, (x, y) => x * y)); // 200
```

Объектно-ориентированное программирование

Классы

```
class User {
    id: number;
    name: string;
    getInfo(): string {
        return "id:" + this.id + " name:" + this.name;
    }
}

let tom: User = new User();
tom.id = 1;
tom.name = "Tom";
console.log(tom.getInfo());
```

Конструктор

```
class User {
  id: number;
  name: string;

  constructor(userId: number, userName: string) {
     this.id = userId;
     this.name = userName;
  }
}
```

Статические свойства и функции

```
class Operation {
    static PI: number = 3.14;

    static getSquare(radius: number): number {
      return Operation.PI * radius * radius;
    }
}

let result = Operation.getSquare(30);
let result2 = Operation.PI * 30 * 30;
```

Модификаторы доступа

```
class User {
    private name: string;
    protected age: number;

    constructor(name: string, age: number) {
        this.name = name;
        this.age = age;
    }
    public displayInfo(): void {
        console.log("name: " + this.name + "; age: " + this.age);
    }
}
```

Определение свойств через конструктор

```
class User {
    constructor(private name: string,
                private age: number)
    {  }

    public displayInfo(): void {
      console.log("name: " + this.name + "; age: " +
      this.age);
    }
}
```

Методы доступа

```
class User {
    private _name: string;
    public get name(): string {
        return this._name;
    }
    public set name(n: string) {
        this._name = n;
    }
}
let tom = new User();
tom.name = "Tom"; // срабатывает set-метод
console.log(tom.name); // срабатывает get-метод
```

Свойства только для чтения

```
class User {
    readonly id: number;
    name: string;

    constructor(userId: number, userName: string) {
      this.id = userId;
      this.name = userName;
    }
}
```

Наследование.

```
class Employee extends User {
    company: string;
    work(): void {
        console.log(this.name + " работает в компании " +
            this.company);
    }
}

let bill: Employee = new Employee("Bill");
bill.getInfo();
bill.company = "Microsoft";
bill.work();
```

Переопределение конструктора

```
class Employee extends User {
  company: string;

  constructor(userName: string, empCompany: string)
  {
     super(userName);
     this.company = empCompany;
  }
}
```

Переопределение методов

```
class Employee extends User {
    company: string;

    getInfo(): void {
      super.getInfo()
      console.log("Работает в компании: "
            + this.company);
    }
}
```

Абстрактные классы

```
abstract class Figure { abstract getArea(): void; }

class Rectangle extends Figure{
    constructor(public width: number, public height: number){
        super();
    }
    getArea(): void{
        let square = this.width * this.height;
        console.log("area =", square);
    }
}

let someFigure: Figure = new Rectangle(20, 30)
someFigure.getArea();
```

Интерфейсы объектов

```
interface IUser {
    id: number;
    name: string;
}

let employee: IUser = {
    id: 1,
    name: "Tom"
}
```

Параметры методов и функций могут представлять интерфейсы:

```
function getEmployeeInfo(user: IUser): void {
    console.log("id: " + user.id);
    console.log("name: " + user.name)
}
```

И также можно возвращать объекты интерфейса:

```
function buildUser(userId: number, userName: string): Iuser
{
      return { id: userId, name: userName };
}

let newUser = buildUser(2, "Bill");
```

Необязательные свойства и свойства только для чтения

```
interface IUser {
    id: number;
    name: string;
    age?: number;
}

interface Point {
    readonly x: number;
    readonly y: number;
}
```

Определение методов

```
interface IUser {
    id: number;
    name: string;
    getFullName(surname: string): string;
}

let employee: IUser = {
    id: 1,
    name: "Alice",
    getFullName : function (surname: string): string {
          return this.name + " " + surname;
    }
}
```

Интерфейсы классов

```
interface IUser {
    id: number;
    name: string;
    getFullName(surname: string): string;
}

class User implements IUser{
    ...
}

let tom = new User(1, "Tom", 23);
console.log(tom.getFullName("Simpson"));
```

Наследование интерфейсов

```
interface IMovable {
    speed: number;
    move(): void;
}

interface ICar extends IMovable {
    fill(): void;
}
```

Преобразование типов

```
let alice: User = new Employee("Microsoft", "Alice");

// ошибка - в классе User нет свойства company
console.log(alice.company);

// преобразование к типу Employee
let aliceEmployee: Employee = <Employee>alice;
console.log(aliceEmployee.company);

// с помощью as
let aliceEmployee: Employee = alice as Employee;
```

Оператор instanceOf

```
let alice: Employee = new Employee("Microsoft", "Alice");

if (alice instanceof User) {
      console.log("Alice is a User");
}
else {
      console.log("Alice is not a User");
}
```

Обобщения

```
function getId<T>(id: T): T {
    return id;
}

let result1 = getId<number>(5);
```

Обобщенные классы и интерфейсы

```
interface IUser<T> {
	getId(): T;
}

class User<T> implements IUser<T> {

	private _id: T;
	getId(): T {
		return this._id;
	}
}
```

Пространства имен

```
namespace Personnel {
    export interface IUser{
      displayInfo();
    }

    export class Employee {
      constructor(public name: string){
      }
    }
}
```