

# **Класове в TS. Прости класове и видове членове на класа. Особености на конструкторите. Наследяване и полиморфизъм. Типови параметри (generics). Duck Typing.**

---

*Роман Литвинов (6MI8000023)  
Юли Чавдаров (5MI0700101)  
Фати Келчева (2MI0700156)  
Майкъл Зарков (45655)  
Людмила Пулова (1MI0700175)*

# Съдържание

---

- Класове в TS. Прости класове и видове членове на класа.
- Особености на конструкторите.
- Наследяване и полиморфизъм.
- Типови параметри (generics).
- Duck Typing.

# Класове в TS.

## Прости класове и видове членове на класа.

1. През 2015 г. ECMAScript 6 въведе нов синтаксис в JavaScript за създаване на класове, които вътрешно използват **прототипните** функции на езика.
2. Както и с другите възможности на езика JavaScript, TypeScript добавя анотации на типове и друг синтаксис, който позволява да изразявате връзки между класове и други типове.
3. TypeScript напълно поддържа този синтаксис и добавя допълнителни функции върху него, като видимост на членовете, абстрактни класове, обобщени (generic) класове, методи със стрелкови функции, модификатори за достъп и някои други.
4. Най-простият пример на клас Base може да се види отясно.



```
class Base {  
  private _x: number = 0;  
  constructor (x: number) {  
    this.x = x //setter called  
  }  
  
  set x(value: number) {  
    if(value < 0)  
      throw Error("Less than zero")  
    this._x = value  
  }  
  
  get x(): number {  
    return this._x;  
  }  
}
```

# ДО ES2015 и после...



JS

```
function Animal(name) {  
  const privateData = new WeakMap();  
  privateData.set(this, { name });  
  
  this.speak = function() {  
    console.log(privateData.get(this).name + ' makes a sound.');  };  
}  
  
const dog = new Animal('Dog');  
dog.speak(); //Dog makes a sound.  
console.log(dog.name); //undefined
```



TS

```
class Animal {  
  private name: string;  
  
  constructor(name: string) {  
    this.name = name;  
  }  
  
  speak(): void {  
    console.log(this.name + ' makes a sound.');  }  
}  
  
const dog = new Animal('Dog');  
dog.speak();  
console.log(dog.name) // Error
```

# Пример

```
export class Channel { Show usages
  private static count: number = 0;
  public readonly id: number;
  public name: string;
  public description: string;
  protected _members: string[];
  private readonly _createdAt: Date;

  constructor(name: string, creator: string, description: string = "") { Show usages
    this.id = Channel.count + 1;
    this.name = name;
    this.description = description;
    this._members = [creator];
    this._createdAt = new Date();
    Channel.count++;
  }

  set members(members: string[]) { no usages
    this._members = [...members]
  }

  get members(): string[] { no usages
    return [...this._members];
  }

  get createdAt(): Date { Show usages
    return new Date(this._createdAt);
  }
}
```

Статична член дана

Член-дани

Конструктор

Сетер

Гетери

Методи

Статични методи

```
addMember(username: string): void { Show usages
  if (!this._members.includes(username)) {
    this._members.push(username);
  } else {
    throw new Error(`${username} is already a member of the channel.`);
  }
}

removeMember(username: string): void { Show usages
  const index: number = this._members.indexOf(username);
  if (index !== -1) {
    this._members.splice(index, 1);
  } else {
    throw new Error(`${username} is not a member of the ${this.name} channel.`);
  }
}

showChannelInfo(): void { Show usages
  console.log(`
ID: ${this.id} Channel: ${this.name}\n Description: ${this.description}\n
Created At: ${this.createdAt}\n Members: ${this._members.join(", ")}\n`);
}

showChannelInfoArrow : () => void = (): void => { Show usages
  console.log(`
ID: ${this.id} Channel: ${this.name}\n Description: ${this.description}\n
Created At: ${this.createdAt}\n Members: ${this._members.join(", ")}\n`);
}

static createChannel(name: string, creator: string, description: string = ""): Channel {
  return new Channel(name, creator, description);
}

static getCount(): number { Show usages
  return Channel.count;
}
```


# Демонстрация

```
function demo(): void { Show usages
  try {
    const channels: Channel[] = [
      Channel.createChannel("General", "admin", "Main discussion channel"),
      new Channel("WEB-2025", "admin", "Materials here")
    ]
    channels.forEach(channel : Channel => {
      channel.addMember("user1");
      channel.addMember("user2");
    })

    //channels[0].removeMember("user1");
    //channels[1]["_members"].push("admin1") // This is a private property,
    //                                     // but we can access it and change using bracket notation
    // const members = ["user3", "user4"]
    // channels[0].members = members;

    Representation.log(channels[1].showChannelInfo); // will not work
    Representation.log(channels[0].showChannelInfoArrow); // will work
    Representation.log(channels[1].showChannelInfo.bind(channels[1])); // will work
    console.log(` Total channels created: ${Channel.getCount()}`)

    channels[0].removeMember("user1"); // throws an Error
  } catch (e) {
    console.log(` Error: ${(e as Error).message}`);
  }
}
```



```
rlitvinov@MacBook-Pro-Roman presentation-demo % npm run start
```

```
> demo@1.0.0 start
> tsc && node dist/index.js
```

```
ID: undefined Channel: undefined
Description: undefined
Created At: undefined
Members: undefined
```


```
ID: 1 Channel: General
Description: Main discussion channel
Created At: Sun Mar 30 2025 02:46:02 GMT+0200 (Eastern European Standard Time)
Members: admin, user1, user2
```

```
ID: 2 Channel: WEB-2025
Description: Materials here
Created At: Sun Mar 30 2025 02:46:02 GMT+0200 (Eastern European Standard Time)
Members: admin, user1, user2
```

```
Total channels created: 2
```

# Какво е конструктор?

- Конструкторът е специален метод в класовете, който се използва за инициализиране на новосъздадените обекти. В повечето обектно-ориентирани езици, включително TypeScript и Java, той се изпълнява автоматично при създаването на инстанция на класа.



```
1 class Person {
2     name: string;
3     age: number;
4
5     constructor(name: string, age: number) {
6         this.name = name;
7         this.age = age;
8     }
9
10    introduce(): void {
11        console.log(`Здравейте, казвам се ${this.name} и съм на ${this.age} години.`);
12    }
13 }
14
15 const person1 = new Person("Иван", 30);
16 person1.introduce(); // Здравейте, казвам се Иван и съм на 30 години.
```

# Създаване на свойства директно от конструктора..

- TypeScript предлага съкратен синтаксис, който позволява автоматично деклариране и инициализация на член-променливи директно в конструктора. Това не е възможно в JavaScript.

TypeScript :

```
class Car {  
    constructor(public brand: string, private model: string, protected year: number) {}  
}  
  
let myCar = new Car("BMW", "X5", 2005);  
console.log(myCar.brand); // OK: BMW  
// console.log(myCar.model); Грешка: model е private  
// console.log(myCar.year); Грешка: year е protected
```



Java:

```
1  class Car {  
2      private String brand;  
3      private String model;  
4      private int year;  
5  
6      public Car(String brand, String model, int year) {  
7          this.brand = brand;  
8          this.model = model;  
9          this.year = year;  
10     }  
11 }
```



# Конструктори с generics

- TypeScript позволява използването на Generics в конструктори.

```
class Box<T> {  
  constructor(public value: T) {}  
  getValue(): T {  
    return this.value;  
  }  
}  
  
let numberBox = new Box<number>(123);  
console.log(numberBox.getValue()); // 123
```

Това не е възможно в JavaScript защото няма поддръжка на Generics.

# Няма поддръжка на Overloading



- В TypeScript няма constructor overloading както в Java или C#. Вместо това, може да се използват union types или if проверки вътре в конструктора.

TypeScript:

```
class User {
  name: string;
  age: number | undefined;

  constructor(name: string);
  constructor(name: string, age: number);
  constructor(name: string, age?: number) {
    this.name = name;
    if (age !== undefined) {
      this.age = age;
    }
  }
}

let user1 = new User("Ivan");
let user2 = new User("Georgi", 30);
console.log(user1); // User { name: 'Ivan' }
console.log(user2); // User { name: 'Georgi', age: 30 }
```

Java:

```
public class User {
  String name;
  int age;

  // Конструктор без възраст
  public User(String name) {
    this.name = name;
    this.age = 0; // Ако няма възраст, слагаме 0 по подразбиране
  }

  // Конструктор с възраст
  public User(String name, int age) {
    this.name = name;
    this.age = age;
  }

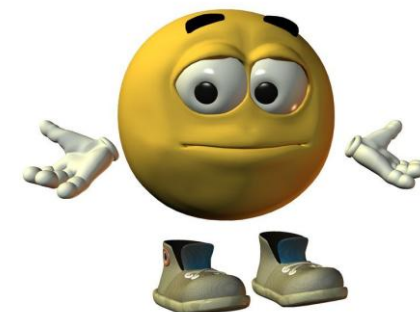
  public void printInfo() {
    System.out.println("Name: " + name + ", Age: " + age);
  }

  public static void main(String[] args) {
    User user1 = new User("Ivan");
    User user2 = new User("Georgi", 30);

    user1.printInfo(); // Извежда: Name: Ivan, Age: 0
    user2.printInfo(); // Извежда: Name: Georgi, Age: 30
  }
}
```

# Изискване за извикване на `super()` в наследените класове

- Ако наследяваме клас, и конструкторът на родителския клас има параметри, трябва задължително да извикаме `super()` в конструктора на наследника, преди да използваме `this`.



```
class Animal {  
    constructor(public name: string) {}  
}  
  
class Dog extends Animal {  
    constructor(name: string, public breed: string) {  
        super(name); // трябва да извикаш super() преди да използваш `this`  
        console.log(`${this.name} е порода ${this.breed}`);  
    }  
}  
  
const dog = new Dog("Buddy", "Golden Retriever");
```

# Type (assertion) Casting в конструкторите.

```
class User {
  age: number;

  constructor(age: string | number) {
    this.age = age as number; // Type assertion
  }

  getAge(): number {
    return this.age;
  }
}

const user1 = new User(25);
const user2 = new User("30"); // Компиляторът не дава грешка, но може да доведе до runtime грешка!

console.log(user1.getAge()); // 25
console.log(user2.getAge()); // "30", но компилаторът мисли, че е число
```

- TypeScript Type Casting е само на ниво компилатор т.е компилаторът просто интерпретира дадена стойност като друг тип, но реално не я преобразува в паметта. Вместо това, TypeScript използва Type Assertions (като `as` и `<type>`), които казват на компилатора да третира стойността по определен начин, без да я променят реално.

# НАСЛЕДЯВАНЕ В TYPESCRIPT

```
1 class Animal {  
2   name: string;  
3  
4   constructor(name: string) {  
5     this.name = name;  
6   }  
7  
8   move(distance: number = 0) {  
9     console.log(`${this.name} moved ${distance} meters.`);  
10  }  
11 }  
12  
13 class Dog extends Animal {  
14   bark() {  
15     console.log("Woof!");  
16   }  
17 }  
18  
19 const dog = new Dog("Buddy");  
20 dog.bark();           // Woof!  
21 dog.move(10);         // Buddy moved 10 meters.
```

# Презаписване на методи (Method Overriding)

---

```
1 class Bird extends Animal {  
2   move(distance: number = 5) {  
3     console.log(`${this.name} flew ${distance} meters.`);  
4   }  
5 }  
6  
7 const bird = new Bird("Pigeon");  
8 bird.move(); // Pigeon flew 5 meters.
```

# ПОЛИМОРФИЗЪМ В TYPESCRIPT

Полиморфизъм = един и същ тип,  
различно поведение.

```
1 class Animal {
2   makeSound() {
3     console.log("Some sound...");
4   }
5 }
6
7 class Cat extends Animal {
8   makeSound() {
9     console.log("Meow");
10  }
11 }
12
13 class Cow extends Animal {
14   makeSound() {
15     console.log("Moo");
16   }
17 }
18
19 function playSound(animal: Animal) {
20   animal.makeSound();
21 }
22
23 playSound(new Cat()); // Meow
24 playSound(new Cow()); // Moo
```

# Полиморфизъм с интерфейси:

```
1 interface Shape {  
2   area(): number;  
3 }  
4  
5 class Circle implements Shape {  
6   constructor(private radius: number) {}  
7   area(): number {  
8     return Math.PI * this.radius ** 2;  
9   }  
10 }  
11  
12 class Square implements Shape {  
13   constructor(private side: number) {}  
14   area(): number {  
15     return this.side ** 2;  
16   }  
17 }  
18  
19 const shapes: Shape[] = [new Circle(3), new Square(4)];  
20  
21 shapes.forEach(shape => console.log(shape.area()));
```

Характеристика	<b>extends</b>	<b>implements</b>
Наследява от....	Клас	Интерфейс
Получава поведение?	Да – методи, свойства	Не – само проверка на структура
Колко може да има?	Само един	Много (разделени със запетая)
Може ли да се комбинират?	Да – extends + implements	Да – например: class C extends A implements B, C {

## Разлика: extends vs implements

# Abstract класове и методи

```
1 abstract class Shape {  
2     abstract area(): number;  
3  
4     describe() {  
5         console.log("I am a shape.");  
6     }  
7 }
```

```
1 class Rectangle extends Shape {  
2     constructor(private w: number, private h: number) {  
3         super();  
4     }  
5  
6     area(): number {  
7         return this.w * this.h;  
8     }  
9 }
```

# Видимост: public, protected, private

<u>Модификатор</u>	<u>Видимост</u>
public	Навсякъде – по подразбиране
protected	В класа и неговите наследници
private	Само в класа, дори наследниците нямат достъп

```
1 class Example {  
2     public a = 1;  
3     protected b = 2;  
4     private c = 3;  
5  
6     log() {  
7         console.log(this.a, this.b, this.c);  
8     }  
9 }  
10  
11 class SubExample extends Example {  
12     logSub() {  
13         console.log(this.a); // OK  
14         console.log(this.b); // OK  
15         // console.log(this.c); // private  
16     }  
17 }
```

# Какво са generics?

---

Можем да си мислим че:

**Generic** типови псевдоними, интерфейси, функции и класове са преизползваеми шаблони, които могат да работят с различни типове данни.



# С какво ни помагат?

---

- Преизползване
- Намаляване повтарянето на код
- Безопасност на типовете (*type safety*)

```
function head(arr: any): any { return arr[0]; }  
head(42);
```



# С какво ни помагат?

---

- Преизползване
- Намаляване повтарянето на код
- Безопасност на типовете (*type safety*)

```
function head(arr: any): any { return arr[0]; }
```

```
head(42); // undefined
```

```
function head<T>(arr: T[]): T { return arr[0]; }
```

```
head(42); // Compile time error.
```



# Generic типови псевдоними

---

```
type UndefOr<T> = undefined | T;
```

# Generic типови псевдоними

---

```
type UndefOr<T> = undefined | T;
```

```
let undefOrNum: UndefOr<number> = 96;
```

```
let undefOrString: UndefOr<string> = undefined;
```

```
let undefOrArr: UndefOr<number[]> = [1, 2, 3];
```

# Примерни типове

---

```
type OneOrMany<T> = T | T[];
```

```
typeNullOrOneOrMany<T> = null | T | T[];
```

```
type Triple<T> = [T, T, T];
```

```
type NonEmptyArray<T> = [T, ...T[]];
```

```
type Zipped<T, H> = [T, H][];
```

# Примерни инстанции

---

```
let a: OneOrMany<Date> = new Date();
```

```
let b:NullOrOneOrMany<string> = null;
```

```
let c: Triple<number> = [1, 2];    // Compile time error!
```

```
let d: Triple<number> = [1, 2, 3]; // OK
```

```
let e: Zipped<number, string> = [ [1, "foo"], [2, "bar"] ];
```

# Generic класове

---

Най-често ги използваме за контейнери за различни типове данни.



# Пример стек

---

```
class Stack<T> {  
    private s: T[] = [];  
  
    push(element: T): void { this.s.push(element); }  
  
    pop(): UndefinedOr<T> { return this.s.pop(); }  
  
    isEmpty(): boolean { return !this.s.length; }  
}
```

# Пример двоично наредено дърво

---

```
class myNode<T> {  
    data: T;  
  
    leftChild: NullOr<myNode<T>>;  
    rightChild: NullOr<myNode<T>>;  
  
    constructor(data: T) {  
        this.data = data;  
        this.leftChild = null;  
        this.rightChild = null;  
    }  
}
```

# Пример двоично наредено дърво

---

```
class BST<T> {  
    private root: myNode<T>;  
  
    constructor(data: T) { this.root = new myNode(data); }  
  
    add(data: T): boolean    { /* ... */ }  
  
    find(data: T): NullOr<T> { /* ... */ }  
  
    returnByLevels(): T[][] { /* ... */ }  
}
```

# Пример двоично наредено дърво

---

```
class BST<T> {  
    private root: myNode<T>;  
  
    constructor(data: T) { this.root = new myNode(data); }  
  
    add(data: T): boolean    { /* ... */ }  
  
    find(data: T): NullOr<T> { /* ... */ }  
  
    returnByLevels(): T[][] { /* ... */ }  
}
```

Обаче това няма да работи!

# Ограничения на типове (type constraints)

---

Можем да поставяме ограничения върху  
типовите параметри.

```
T extends { le(t: T): boolean }
```



# Пример двоично наредено дърво (коригиран)

---

```
class BST<T extends { le(t: T): boolean }> {  
    private root: myNode<T>;  
  
    constructor(data: T) { this.root = new myNode(data); }  
  
    add(data: T): boolean    { /* ... */ }  
  
    find(data: T): NullOr<T> { /* ... */ }  
  
    returnByLevels(): T[][] { /* ... */ }  
}
```

# Пример ограничения

---

```
interface TrustMeBro {  
    engineer: boolean;  
    name: string;  
}
```

```
class Mechanic<T extends TrustMeBro> {  
  
    mechanic: T;  
  
    // ...  
  
    qualified(): boolean {  
        return this.mechanic.engineer;  
    }  
  
    // ...  
}
```

# Статични функции

---

Статичните функции в класовете **не могат** да използват типови параметри, защото конкретна статична функция за конкретен *generic* клас е една и съща независимо от типовете параметри.

```
class MyClass<T> {  
    static foo(t: T): T { // Compile time error!  
        return t;  
    }  
}
```

# Вградени generic типове/класове

---

`Array<T>` същото като `T[]`

`ReadOnlyArray<T>` същото като `readonly T[]`

`ReadOnly<T>`

`Set<T>`

`Partial<T>`

`Required<T>`

# Duck Typing

---

```
if (x === "🦆") {  
  |   console.log("It's a duck!");  
}
```

# Какво е Duck Typing?

---

- "If it walks like a duck, swims like a duck, and quacks like a duck – then it probably is a duck.,,
- Важното е какво може обектът, не какъв е по произход.



Номинално Типизиране	Duck Typing	Structural Typing
Java	Python	TypeScript
C++	Ruby	Go
C#	JavaScript	OCaml

# Сравняване с номинално типизиране

```
1 interface Animal {
2     makeSound(): void;
3 }
4
5 class Dog {
6     makeSound() {
7         console.log("Woof!");
8     }
9 }
10
11 class Cat {
12     makeSound() {
13         console.log("Meow!");
14     }
15 }
16
17 function makeItSpeak(animal: Animal) {
18     animal.makeSound();
19 }
20
21 const dog = new Dog();
22 const cat = new Cat();
23
24 makeItSpeak(dog); // OK
25 makeItSpeak(cat); // OK
```

Съвместимостта се основава на структурата

Проверка : По структура – дали има нужните методи

```
1 interface Animal {
2     void makeSound();
3 }
4
5 class Dog implements Animal {
6     public void makeSound() {
7         System.out.println("Woof!");
8     }
9 }
10
11 class Cat {
12     public void makeSound() {
13         System.out.println("Meow!");
14     }
15 }
16
17 public class Main {
18     public static void makeItSpeak(Animal animal) {
19         animal.makeSound();
20     }
21
22     public static void main(String[] args) {
23         Dog dog = new Dog();
24         Cat cat = new Cat();
25
26         makeItSpeak(dog); // OK
27
28         // makeItSpeak(cat); // Error: Cat does not implement Animal
29     }
30 }
```

Съвместимостта на типовете се определя по име и йерархия

Има изрична декларация на връзката

Проверка : По име и връзка към интерфейс

# Duck Typing срещу Structural Typing

```
1 function makeItQuack(duck) {
2   |   duck.quack();
3 }
4
5 const realDuck = {
6   |   quack: function() {
7     |     console.log("Quack!");
8   |   }
9 };
10
11 const person = {
12   |   quack: function() {
13     |     console.log("I'm imitating a duck!");
14   |   }
15 };
16
17 makeItQuack(realDuck); // Извежда: "Quack!"
18 makeItQuack(person);  // Извежда: "I'm imitating a duck!"
```

Проверка на типовете се извършва по време на изпълнение (run-time)

Основава се на наличието на определени методи и свойства в обекта

```
1 interface Quackable {
2   |   quack(): void;
3 }
4
5 class Duck implements Quackable {
6   |   quack() {
7     |     console.log("Quack!");
8   |   }
9 }
10
11 class Person {
12   |   quack() {
13     |     console.log("I'm imitating a duck!");
14   |   }
15 }
16
17 function makeItQuack(duck: Quackable) {
18   |   duck.quack();
19 }
20
21 const realDuck = new Duck();
22 const person = new Person();
23
24 makeItQuack(realDuck); // Извежда: "Quack!"
25 makeItQuack(person);  // Извежда: "I'm imitating a duck!"
26
```

Проверка на типовете се извършва по време на компилация (compile-time)

Основава се на сравнение на структурите на типовете

# TypeScript

```
1  ✓ interface Person {  
2    |   name: string;  
3    |   age: number;  
4    | }  
5  
6    const ivan = { name: "Ivan", age: 21, faculty: "FMI" };  
7  ✓ let student: Person = ivan; // OK: обектът ivan има  
8    |   |   |   |   |   |   |   // всички свойства на Person  
9    console.log(student.name); // "Ivan"  
10
```

Пример 1: Интерфейс и обект

```
1  class Dog {  
2    |   name: string;  
3    |   constructor(name: string) { this.name = name; }  
4    | }  
5  class Person {  
6    |   name: string;  
7    |   constructor(name: string) { this.name = name; }  
8    | }  
9  
10 function greet(entity: { name: string }) {  
11   |   console.log(`Hello, ${entity.name}!`);  
12   | }  
13  
14 const myDog = new Dog("Sharo");  
15 const myPerson = new Person("Ivan");  
16  
17 greet(myDog); // Hello, Sharo!  
18 greet(myPerson); // Hello, Ivan!
```

Пример 2: Функция с параметър по структура (не по клас)

# Защо е полезно?

---

**Гъвкавост:** Работи с различни обекти без нужда от наследяване.

**По-малко шаблони:** Функциите могат да приемат различни типове по структура.

**Лесна интеграция:** Работи директно с JSON или външни данни.

**Комфорт + сигурност:** Комбинация от динамична гъвкавост и статична проверка.

# Пример: Система за известия

```
1 // Интерфейс, дефиниран по структура
2 interface Notifier {
3   | send(message: string): void;
4 }
5
6 // Различни класове, които пасват на интерфейса по структура
7
8 class EmailService {
9   | send(message: string) {
10    |   console.log(`Email: ${message}`);
11   | }
12 }
13
14 class SMSService {
15   | send(message: string) {
16    |   console.log(`SMS: ${message}`);
17   | }
18 }
19
20 class PushNotification {
21   | send(message: string) {
22    |   console.log(`Push Notification: ${message}`);
23   | }
24 }
25
26 // Функция, която приема всеки обект, който "изглежда" като Notifier
27 function notifyAll(notifiers: Notifier[], message: string) {
28   | for (const notifier of notifiers) {
29    |   notifier.send(message);
30   | }
31 }
```

```
32
33 // Създаваме обекти без да ги "вързваме" с интерфейса
34 const email = new EmailService();
35 const sms = new SMSService();
36 const push = new PushNotification();
37
38 notifyAll([email, sms, push], "Welcome to the system!");
39 |
```

# Благодарим за вниманието!

---

*Роман Литвинов (6MI8000023)*

*Юли Чавдаров (5MI0700101)*

*Фати Келчева (2MI0700156)*

*Майкъл Зарков (45655)*

*Людмила Пулова (1MI0700175)*