



# GIS RIA

---

TypeScript

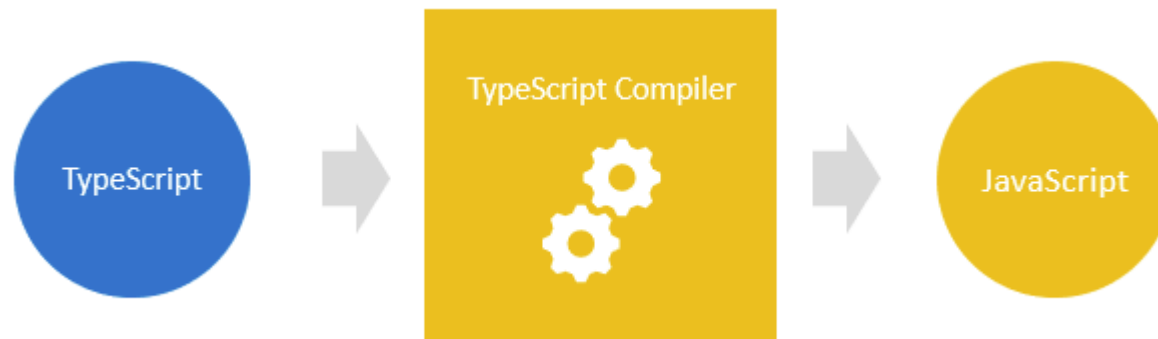
---

# 1 ŠTA JE TYPESCRIPT

---



- TypeScript je nadskup JavaScripta.
- TypeScript je izgrađen na JavaScriptu.
- Kada se napiše TypeScript kod, kompajlira se u običan JavaScript kod upotrebom TypeScript kompajlera. Takav kod se može postaviti u bilo koje okruženje koje izvršava JavaScript.
- TypeScript fajl koristi .ts ekstenziju umesto .js ekstenzije koju koriste JavaScript fajlovi.

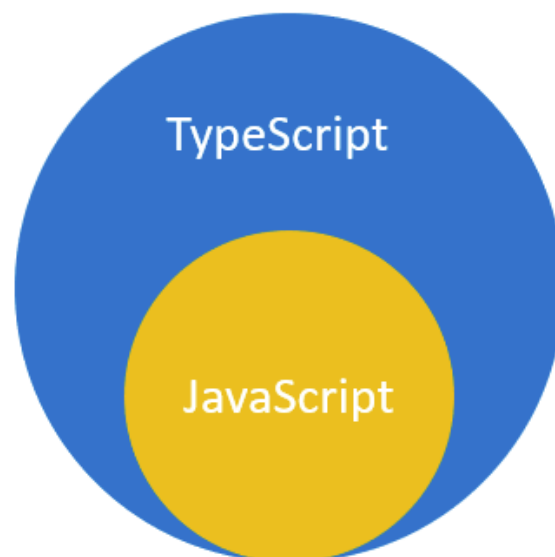


# TypeScript vs JavaScript

---



- TypeScript koristi JavaScript sintaksu i dodaje dodatnu sintaksu za podršku tipovima.
- JavaScript program koji nema sintaksne greške je takođe TypeScript program.
- To znači da su svi JavaScript programi i TypeScript programi, što je korisno kada se migrira postojeći kod na novu platformu.



# Zašto TypeScript

---



- Osnovni ciljevi TypeScripta su:
  1. Uvođenje opcionih tipova u JavaScript
  2. Implementirajte planiranih svojstava budućeg JavaScript-a, zvanog ECMAScript Next ili ES Next u trenutni JavaScript

# 1) Uvođenje tipova da bi se izbegli bagovi



- TypeScript poboljšava produktivnost i istovremeno pomaže u izbegavanju grešaka.
- Tipovi povećavaju produktivnost pomažući da se izbegnu mnoge greške.
- Korišćenjem tipove, mogu se uhvatiti greške u vreme kompajliranja umesto da se pojavljuju tokom izvršavanja.

```
function add(x, y) {  
  return x + y;  
}
```

```
let result = add(input1.value, input2.value);  
console.log(result); // result of concatenating strings
```

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

Ako se ne unese broj kompajler će vratiti grešku!

## 2) TypeScript donosi budućnost JavaScripta danas

---



- TypeScript podržava predstojeće funkcionalnosti planirane u ES Next za trenutne JavaScript endžine.
- To znači da se mogu koristiti nove JavaScript funkcionalnosti pre nego što ih web brauzeri (ili druga okruženja) u potpunosti podrže.

# TypeScript Setup

---



- Node.js – je okruženje u kom se pokreće TypeScript kompajler.
- TypeScript compiler – Node.js modul koji kompajlira TypeScript u JavaScript.
  - `npm install -g typescript`
- Visual Studio Code ili VS Code – je editor koda koji podržava TypeScript. Preporuka je da se koristi VS Code, ali mogu se koristiti i drugi editori.

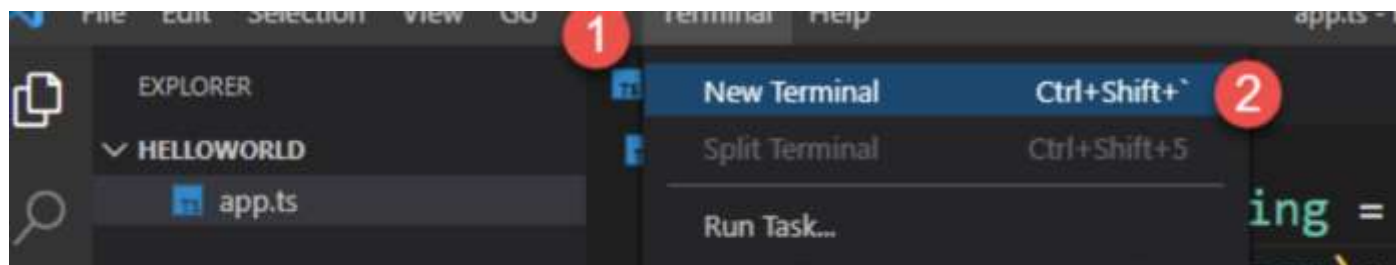
# Hello World primer



```
let message: string = 'Hello, World!';  
console.log(message);
```

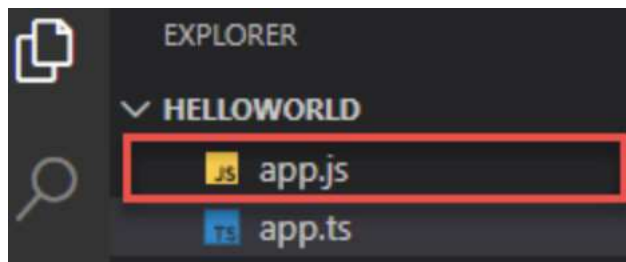
Pisanje koda

Otvaranje terminala



```
tsc app.ts
```

Kompajliranje TypeScript koda u JavaScript kod



Kompajliran kod

```
node app.js
```





# Dinamički tipovi u JavaScript-u

---

- **JavaScript je dinamički tipiziran.** Za razliku od statički tipiziranih jezika kao što su Java ili C#, **umesto promenljivih vrednosti su te koje imaju tipove.**

- "Hello" – string
- 2021 - number

Tip promenljive box se menja na osnovu vrednosti koja joj je dodeljena.

```
let box;  
console.log(typeof(box)); // undefined  
  
box = "Hello";  
console.log(typeof(box)); // string  
  
box = 100;  
console.log(typeof(box)); // number
```

# TypeScript dodajte sistem tipova da bi se izbegli mnogi problemi sa dinamičkim tipovima u JavaScriptu

---



- U JavaScriptu nije potrebno eksplicitno navesti tip. JS će automatski zaključiti tip na osnovu vrednosti. Zato kažemo da su tipovi dinamički.
- Dinamički tipovi nude fleksibilnost, ali dovode do grešaka i problema.  
Primer:
  - obrnuli smo redosled parametara prilikom poziva funkcije
  - Pogrešili smo kapitalizaciju slova promenljive,
  - itd.
- TypeScript dodaje opcioni sistem tipova da bi rešio ove probleme.

# 2 OSNOVNI TIPOVI

---



- TypeScript Types
- Type Annotations
- Type Inference
- Number Type
- String Type
- Boolean Type
- Object Type
- Array Type
- Tuple Types
- Enum Types
- Any Type
- Never Type
- Union Types
- Type Aliases
- String Literal Types



## 2.1 TypeScript tipovi

---

- U TS, tip je zgodan način da se uputi na različite **atribute** i **funkcije** koje **vrednost** ima.
- Vrednost je bilo šta što se može dodeliti promenljivoj, npr. broj, string, niz, objekat ili funkcija.

```
console.log('Hello'.length); // 5
```

```
console.log('Hello'.toLocaleUpperCase()); // HELLO
```

# TypeScript tipovi

---



- Tip je labela koja opisuje različite attribute i metode koje ima vrednost.
- Svaka vrednost ima tip.
- TypeScript nasleđuje ugrađene tipove iz JavaScripta.
- TypeScript tipovi su kategorizovani u:
  - Primitivne tipove
  - Objektne tipove
- Dve osnovne svrhe tipova u TypeScriptu su:
  - Tipove koristi TypeScript kompajler da analizira kod na greške
  - Tipovi omogućavaju da se razume kakve vrednosti su pridružene promenljivama.

# Prmitivni tipovi

---



Name	Description
<code>string</code>	represents text data
<code>number</code>	represents numeric values
<code>boolean</code>	has true and false values
<code>null</code>	has one value: null
<code>undefined</code>	has one value: undefined. It is a default value of an uninitialized variable
<code>symbol</code>	represents a unique constant value

# Objektni tipovi

---



- Objektni tipovi su funkcije, nizovi, klase...
- Može se kreirati custom objektni tip.

# Primer



- TypeScript kompajler zna da je tip headinga `HTMLHeadingElement`

```
const heading = document.querySelector('h1');
```

```
const heading = document.querySelector('h1');  
heading?.
```

- accessKey
- accessKeyLabel
- addEventListener
- after
- align
- animate
- append
- appendChild
- assignedSlot
- attachShadow
- attributes
- autocapitalize



# Rezime

---



- Svaka vrednost u TypeScriptu ima tip.
- Tip je labela koja opisuje attribute i metode koje vrednost ima.
- TypeScript kompajler koristi tipove da analizira kod i pronade bagove.

## 2.2 Anotacije tipova



- TypeScript koristi anotacije tipova da eksplicitno specificira tipove za identifikatore kao što su promenljive, funkcije, objekti, itd.
- TypeScript koristi sintaksu: tip iza identifikatora kao anotacija tipa, gde tip može biti bilo koji validan tip.
- Jednom kada je identifikator anotiran tipom, može biti korišćen samo sa tim tipom, inače će kompajler javiti grešku.

```
let variableName: type;  
let variableName: type = value;  
const constantName: type = value;
```

```
let counter: number;
```

```
let name: string = 'John';  
let age: number = 25;  
let active: boolean = true;
```

Kompajler vraća grešku za pogrešan tip.  
Laboratorija za geoinformatiku

# Primeri



```
let person: {  
  name: string;  
  age: number  
};
```

```
person = {  
  name: 'John',  
  age: 25  
}; // valid
```

```
let greeting : (name: string) => string;
```

```
greeting = function (name: string) {  
  return `Hi ${name}`;  
};
```

```
let names: string[] = ['John', 'Jane', 'Peter', 'David', 'Mary'];
```



# Zaključivanje tipova u TypeScriptu

---

- Kada se tip eksplicitno ne anotira, TypeScript može da zaključi tip, npr. iz dodeljene vrednosti prilikom inicijalizacije.
- Na primer, ako se promenljiva counter inicijalizuje brojem, kompajler će smatrati da je tip number:
  - let counter = 0;
  - let counter: number = 0;

```
function increment(counter: number) {  
    return counter++;  
}
```

```
function increment(counter: number) : number {  
    return counter++;  
}
```

# Anotacija vs Zaključivanje

---



## ▪ Zaključivanje

- Kompajler pogađa tip
  - The best common type algorithm
- `let counter = 0;`
- `let items = [0, 1, null, 'Hi'];`
  - `(number | string)[]`

## ▪ Anotacija

- Programer eksplicitno definiše tip
- `let counter: number;`

# Anotacija vs Zaključivanje

---



- U praksi treba uvek koristiti zaključivanje tipova što je više moguće. Eksplicitne anotacija se koriste u sledećim slučajevima:
  - Kada se deklariše promenljiva a vrednost joj se dodeljuje kasnije.
  - Kada se želi promenljiva čiji tip se ne može zaključiti.
  - Kada funkcija vrati any tip, a potrebno je razjasniti vrednost.



# Kontekstualno tipiziranje

---

- U ovom primeru, TypeScript zna da je parametar događaja instanca `MouseEvent` zbog događaja `click`.

```
document.addEventListener('click', function (event) {  
    console.log(event.button); //  
});
```

- TypeScript zna da je događaj u ovom slučaju instanca `UIEvent`, a ne `MouseEvent`. `UIEvent` nema atribut `dugme`, stoga TypeScript prikazuje grešku.

```
document.addEventListener('scroll', function (event) {  
    console.log(event.button); // compiler error  
});
```

```
Property 'button' does not exist on type 'Event'.(2339)
```

# Rezime

---



- Zaključivanje tipa se dešava kada se inicijalizuje promenljiva, postavi podrazumevana vrednost parametara, i odredi povratni tip funkcije.
- TypeScript koristi best common type algorithm za odabir najboljih tipova kandidata koji su kompatibilni sa svim varijablama.
- TypeScript takođe koristi kontekstualno tipiziranje da bi zaključio tipove promenljivih na osnovu lokacija varijabli.





# Number, String, Boolean

---

```
let counter: number = 0;  
let x: number = 100,  
    y: number = 200;
```

```
let price = 9.95;
```

```
let firstName: string = 'John';  
let title: string = "Web Developer";
```

```
let pending: boolean;  
pending = true;  
// after a while  
// ..  
pending = false;
```

# Objektni tip



- TypeScript objektni tipovi predstavljaju sve vrednosti koje nisu primitivni tipovi.
- Primitivni tipovi su:
  - number
  - bigint
  - string
  - boolean
  - null
  - undefined
  - symbol

```
let employee: object;  
  
employee = {  
    firstName: 'John',  
    lastName: 'Doe',  
    age: 25,  
    jobTitle: 'Web Developer'  
};  
  
console.log(employee);
```

# Objektni tip



```
let employee: object;  
  
employee = {  
    firstName: 'John',  
    lastName: 'Doe',  
    age: 25,  
    jobTitle: 'Web Developer'  
};  
  
console.log(employee);
```

```
{  
    firstName: 'John',  
    lastName: 'Doe',  
    age: 25,  
    jobTitle: 'Web Developer'  
}
```

```
employee = "Jane";
```

```
console.log(employee.hireDate);
```

```
error TS2322: Type '"Jane"' is not assignable to type 'object'.
```

```
error TS2339: Property 'hireDate' does not exist on type 'object'.
```

# Objektni tip



```
let employee: {  
  firstName: string;  
  lastName: string;  
  age: number;  
  jobTitle: string;  
};
```

```
employee = {  
  firstName: 'John',  
  lastName: 'Doe',  
  age: 25,  
  jobTitle: 'Web Developer'  
};
```

```
let employee: {  
  firstName: string;  
  lastName: string;  
  age: number;  
  jobTitle: string;  
} = {  
  firstName: 'John',  
  lastName: 'Doe',  
  age: 25,  
  jobTitle: 'Web Developer'  
};
```

# object vs Object

---



- Tip `object` predstavlja sve ne-primitivne vrednosti, dok tip `Object` opisuje funkcionalnosti svih objekata.
- Na primer, `Object` tip ima metode `toString()` i `valueOf()` kojima se može pristupiti iz bilo kog objekta.
- `empty` tip `{}` se odnosi na objekat koji nema svojih atributa.

# Nizovi

---



```
let skills: string[];
```

```
skills[0] = "Problem Solving";  
skills[1] = "Programming";
```

```
let skills = ['Problem Solving', 'Software Design', 'Programming'];
```

```
skills.push('Software Design');
```

```
let skills: string[];  
skills = ['Problem Solving', 'Software Design', 'Programming'];
```

```
let scores : (string | number)[];  
scores = ['Programming', 5, 'Software Design', 4];
```

Miks tipova



# Atributi i metode niza

---

- **length, forEach(), map(), reduce(), filter(), push()**

```
let series = [1, 2, 3];  
let doubleIt = series.map(e => e* 2);  
console.log(doubleIt);
```

```
[ 2, 4, 6 ]
```

```
let skills: string[];  
skills = ['Problem Solving', 'Software Design', 'Programming'];  
skills.push(100);
```

Argument of type 'number' is not assignable to parameter of type 'string'

# Tuple

---



- Tuple tipovi su slični nizovima sa nekim dodatnim razmatranjima:
  - Broj elemenata u tuple je fiksiran.
  - Tipovi elemenata su poznati, i ne moraju biti isti.
- Na primer, tuple može da predstavlja vrednost kao par broja i stringa

```
let skill: [string, number];  
skill = ['Programming', 5];
```



# Enum

---



- Enum je grupa konstantnih vrednosti koja ima ime.
- Enum tip je enumeracija.

```
enum name {constant1, constant2, ...};
```

# Enum



```
enum Month {  
    Jan,  
    Feb,  
    Mar,  
    Apr,  
    May,  
    Jun,  
    Jul,  
    Aug,  
    Sep,  
    Oct,  
    Nov,  
    Dec  
};
```

```
function isItSummer(month: Month) {  
    let isSummer: boolean;  
    switch (month) {  
        case Month.Jun:  
        case Month.Jul:  
        case Month.Aug:  
            isSummer = true;  
            break;  
        default:  
            isSummer = false;  
            break;  
    }  
    return isSummer;  
}
```

# Any



- TypeScript any tip omogućava da se smesti vrednost bilo kog tipa.
- Any instruiše kompajler da preskoči proveru tipa.
- Koristi se kada ne znamo tip u vreme kompajliranja ili migriramo stari JS projekat u TS projekat.

```
let result: any;  
result = 10.123;  
console.log(result.toFixed());  
result.willExist(); //
```

```
let result;
```

# Never

---



- never tip ne sadrži vrednost - no value.
- Predstavlja povratni tip funkcije koja uvek baca grešku ili sadrži beskonačnu petlju.

# Union



- union tip omogućava da se smesti vrednost jednog ili više tipova u promenljivu

```
let result: number | string;  
result = 10; // OK  
result = 'Hi'; // also OK  
result = false; // a boolean value, not OK
```

# Alias tipova

---



- Alias tipova se koriste da se definiše novi naziv za postojeći tip.

```
type alphanumeric = string | number;  
let input: alphanumeric;  
input = 100; // valid  
input = 'Hi'; // valid  
input = false; // Compiler error
```



# String literal tip

---

- string literal definiše tip koji prihvata specificirani string literal.
- string literali se koriste sa union tipovima i alijasima da se definišu tipovi koji prihvataju konačan skup string literala.

```
type MouseEvent: 'click' | 'dblclick' | 'mouseup' | 'mousedown';  
let mouseEvent: MouseEvent;  
mouseEvent = 'click'; // valid  
mouseEvent = 'dblclick'; // valid  
mouseEvent = 'mouseup'; // valid  
mouseEvent = 'mousedown'; // valid  
mouseEvent = 'mouseover'; // compiler error  
  
let anotherEvent: MouseEvent;
```

# 3 NAREDBE ZA KONTROLU TOKA IZVRŠAVANJA

---



- if else
- switch case
- for
- while
- do while
- break





# if - else, ternarni operator, switch

```
if(condition) {  
    // if-statements  
} else {  
    // else statements;  
}
```

```
const max = 100;  
let counter = 100;  
  
counter < max ? counter++ : counter = 1;  
  
console.log(counter);
```

```
switch ( expression ) {  
    case value1:  
        // statement 1  
        break;  
    case value2:  
        // statement 2  
        break;  
    case valueN:  
        // statement N  
        break;  
    default:  
        //  
        break;  
}
```

# For



```
for(initialization; condition; expression) {  
    // statement  
}
```

```
for (let i = 0; i < 10; i++) {  
    console.log(i);  
}
```

```
for (var i = 0; i < products.length; i++) {  
    if (products[i].price == 900)  
        break;  
}
```

# While



```
while(condition) {  
    // do something  
}
```

```
while(condition) {  
    // do something  
    // ...  
  
    if(anotherCondition)  
        break;  
}
```

```
let counter = 0;  
  
while (counter < 5) {  
    console.log(counter);  
    counter++;  
}
```

# Do - while



```
do {  
    // do something  
} while(condition);
```

```
let i = 0;  
  
do {  
    console.log(i);  
    i++;  
} while (i < 10);
```

```
let index = 9;  
let count = 0;  
  
do {  
    index += 1;  
  
    if (index % 2)  
        continue;  
    count += 1;  
} while (index < 99);
```

Break izlazi iz petlje, a continue preskače ostatak koda i prelazi na novu iteraciju

# 4 Funkcije

---



- Functions
- Function Types
- Optional Parameters
- Default Parameters
- Rest Parameters
- Function Overloadings

# Funkcije

---



```
function name(parameter: type, parameter:type,...): returnType {  
    // do something  
}
```

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



# Function tipovi

---

- TypeScript function tipovi omogućavaju da se definišu tipovi za funkcije.
- function tip ima dva dela: parametri i return tip.

```
(parameter: type, parameter:type,...) => type
```

```
let add: (x: number, y: number) => number;
```

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



# Zaključivanje tipa iz konteksta

---

```
let add: (a: number, b: number) => number =  
  function (x: number, y: number) {  
    return x + y;  
  };
```

```
let add = function (x: number, y: number): number {  
  return x + y;  
};  
  
let result = add(10, 20);
```

```
let add: (x: number, y: number) => number
```



# Opcioni parametri



- Sintaksa **parameter?: type** omogućava da parameter bude opcion.
- Izrazom **typeof(parameter) !== 'undefined'** se proverava da li je parameter bio inicijalizovan.

```
function multiply(a: number, b: number, c?: number): number {  
  
    if (typeof c !== 'undefined') {  
        return a * b * c;  
    }  
    return a * b;  
}
```



# Default parametri

---

- Sintaksa **parameter:=defaultValue** se koristi da se inicijalizuje default vrednost za parametar.
- Default parametri su opcioni.
- Da bi se koristila default inicijalizovana vrednost parametra, taj argument se preskoči prilikom poziva funkcije ili se prosledi undefined u funkciju.

```
function name(parameter1=defaultValue1,...) {  
    // do something  
}
```

```
function applyDiscount(price, discount = 0.05) {  
    return price * (1 - discount);  
}
```

# Rest parametri



- Rest parametri omogućavaju da se beskonačan broj argumenata predstavi kao niz.
- Rest parametri omogućavaju da funkcija primi 0 ili više argumenata određenog tipa.
- U TypeScriptu, rest parametri prate sledeća pravila:
  - Funkcija ima samo jedan rest parametar.
  - Rest parameter se pojavljuje poslednji u listi parametara.
  - Tip rest parametra je niz.

```
function fn(...rest: type[]) {  
    //...  
}
```

```
function getTotal(...numbers: number[]): number {  
    let total = 0;  
    numbers.forEach((num) => total += num);  
    return total;  
}
```



# Function overloading

---

- function overloading omogućava da se uspostavi veza između tipova parametara i rezultujućeg tipa funkcije.
- Primer, ako su parametri number, funkcija treba da vrati number. Ako su parametri string, funkcija treba da vrati string.

```
function addNumbers(a: number, b: number): number {  
    return a + b;  
}  
  
function addStrings(a: string, b: string): string {  
    return a + b;  
}
```

# 5 Klase

---



- Classes
- Access Modifiers
- Readonly Properties
- Inheritances
- Static Methods and Properties
- Abstract Classes

# TypeScript klasa

---



- JavaScript nema koncept klase kao drugi programski jezici kao što su Java i C#.
- U ES5, može se koristiti funkcija konstruktor i prototipsko nasleđivanje da se kreira “klasa”

```
function Person(ssn, firstName, lastName) {  
    this.ssn = ssn;  
    this.firstName = firstName;  
    this.lastName = lastName;  
}
```

```
let person = new Person('171-28-0926', 'John', 'Doe');  
console.log(person.getFullName());
```



# TypeScript klasa u EcmaScript 6

- ES6 omogućava da se definiše klasa što je samo drugačiji sintaksni okvir za kreiranje konstruktor funkcije i prototipsko nasleđivanje :

```
class Person {  
    ssn;  
    firstName;  
    lastName;  
  
    constructor(ssn, firstName, lastName) {  
        this.ssn = ssn;  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
  
    getFullName() {  
        return `${this.firstName} ${this.lastName}`;  
    }  
}
```

```
let person = new Person('171-28-0926', 'John', 'Doe');  
console.log(person.getFullName());
```

# Anotacije za tipove

---



```
class Person {  
    ssn: string;  
    firstName: string;  
    lastName: string;  
  
    constructor(ssn: string, firstName: string, lastName: string) {  
        this.ssn = ssn;  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
  
    getFullName(): string {  
        return `${this.firstName} ${this.lastName}`;  
    }  
}
```



# Modifikatori pristupa



- Modifikatori pristupa menjaju vidljivost atributa i metoda klase. TypeScript nudi 3 modifikatora pristupa:

- private
- protected
- public

```
class Person {  
    private ssn: string;  
    private firstName: string;  
    private lastName: string;  
    // ...  
}
```

```
class Person {  
  
    protected ssn: string;  
  
    // other code  
}
```

```
class Person {  
    // ...  
    public getFullName(): string {  
        return `${this.firstName} ${this.lastName}`;  
    }  
}
```

# Readonly vs. const

---



	<code>readonly</code>	<code>const</code>
Use for	Class properties	Variables
Initialization	In the declaration or in the constructor of the same class	In the declaration

```
class Person {  
    readonly birthDate: Date;  
  
    constructor(birthDate: Date) {  
        this.birthDate = birthDate;  
    }  
}
```

# Geteri i seteri



- TypeScript getters/setters se koriste da kontrolišu pristup atributima klase kroz get i set metode koje čitaju (get) i menjaju (set) vrednost atribut.
- getter/setters su takođe poznati kao accessors/mutators.

```
public set fullName(name: string) {  
    let parts = name.split(' ');  
    if (parts.length !== 2) {  
        throw new Error('Invalid name');  
    }  
    this.firstName = parts[0];  
    this.lastName = parts[1];  
}
```

```
let person = new Person();  
person.fullname = 'John Doe';
```

# Nasleđivanje – klase roditelj-dete



- Ključna reč **extends** se koristi da omogući klasi da nasledi drugu klasu.
- Metoda `super()` se koristi u konstruktoru klase naslednika da pozove konstruktor roditeljske klase.
- `super.methodInParentClass()` sintaksa se koristi da pozove metodu iz roditeljske klase `methodInParentClass()` u metodi dete klase.

```
class Employee extends Person {  
    constructor(  
        firstName: string,  
        lastName: string,  
        private jobTitle: string) {  
  
        super(firstName, lastName);  
    }  
  
    describe(): string {  
        return super.describe() + `I'm a ${this.jobTitle}.`;  
    }  
}
```

Method overriding

# Statičke metode i atributi



- Za razliku od atributa i metoda instance, statički atribut i metode se dele između svih instanca date klase.
- Da bi se atribut ili metoda deklarovali kao statički koristi se ključna reč **static**.
- Da bi se pristupilo statičkom atributu ili metodi koristi se sintaksa:
  - `className.propertyName`
  - `className.methodName()`

```
class Employee {  
    private static headcount: number = 0;  
  
    constructor(  
        private firstName: string,  
        private lastName: string,  
        private jobTitle: string) {  
  
        Employee.headcount++;  
    }  
  
    public static getHeadcount() {  
        return Employee.headcount;  
    }  
}
```

# Apstraktne klase



- Apstraktne klase se ne mogu instancirati.
- Da bi se klasa deklarovala kao apstraktna treba koristiti ključnu reč **abstract**.
- Da bi se koristila apstraktna klasa, treba je naslediti i obezbediti implementaciju apstraktnih metoda.

```
abstract class Employee {  
    constructor(private firstName: string, private lastName: string) {  
    }  
    abstract getSalary(): number  
    get fullName(): string {  
        return `${this.firstName} ${this.lastName}`;  
    }  
    compensationStatement(): string {  
        return `${this.fullName} makes ${this.getSalary()} a month.`;  
    }  
}
```

```
class FullTimeEmployee extends Employee {
```

# 6 Interfejsi



- TypeScript interfejsi definišu ugovore u kodu i pružaju eksplicitne nazive za proveru tipova.
- Interfejsi mogu imati opcione ili readonly atribut.
- Mogu biti korišćeni kao tipovi funkcije.
- Interfejsi se tipično koriste kao tipovi klasa koji prave ugovor između nepovezanih klasa.

```
interface Json {  
    toJSON(): string  
}
```

```
class Person implements Json {  
    constructor(private firstName: string,  
                private lastName: string) {  
    }  
    toJson(): string {  
        return JSON.stringify(this);  
    }  
}
```



# Interfejsi proširuju (**extends**) klase

---

- TypeScript omogućava interfejsu da proširi klasu (**extends**).
- Interfejs nasleđuje attribute i metode klase.
- Interfejs može naslediti **private** i **protected** attribute i metode, a ne samo **public** attribute i metode.
- To znači da kada interfejs proširuje klasu **private** ili **protected** atributima i metodama, interfejs može da implementira samo ta klasa ili podklase te klase iz koje se interfejs proširuje.
- Na taj način se ograničava upotreba interfejsa samo na klasu ili podklase klase iz koje se interfejs proširuje.
- Ako pokušate da implementirate interfejs iz klase koja nije podklasa klase koju je interfejs nasledio, dobićete grešku.



# 7 Napredni tipovi

---



- Intersection Types
- Type Guards
- Type Casting
- Type Assertions



# Intersection tipovi

---

- Tip preseka kombinuje dva ili više tipova da bi napravio novi tip koji ima sva svojstva postojećih tipova.
- Da bi se kombinovali tipovi koristi se operator **&**:

```
type typeAB = typeA & typeB;
```

- typeAB ima sva svojstva (atriburte i metode) tipova typeA i typeB istovremeno.
- Napomena: union tip koristi operator **|** i definiše promenljive koje mogu da sadrže vrednost ili typeA ili typeB

```
type Employee = Identity & Contact;  
type Customer = BusinessPartner & Contact;
```



# Type Guards –čuvvari tipova

---

- Type guards (čuvvari tipova) sužavaju tip promenljive unutar uslovnog bloka.
- Koriste se operatori **typeof** i **instanceof** da bi se implementirali čuvvari tipova uslovnom bloku.

```
if (typeof a === 'string' && typeof b === 'string') {  
    return a.concat(b);  
}
```

```
if (partner instanceof Customer) {  
    message = partner.isCreditAllowed() ?  
}
```

# Type Casting

---



- Type casting omogućava konvertovanje promenljive iz jednog tipa u drugi
- Koristi se ključna reč **as** ili operator **<>** za type casting.

```
let a: typeA;  
let b = a as typeB;
```

```
let a: typeA;  
let b = <typeB>a;
```



# Type Assertions – tvrdnje o tipu

---

- Type assertions instruišu kompajler da tretira vrednost kao određeni specificirani tip.
- Type assertions ne vrše bilo kakvu konverziju tipa.
- Type assertions koriste ključnu reč **as** ili uglaste zagrade **<>**.

```
expression as targetType
```

```
let netPrice = getNetPrice(100, 0.05, true) as string;
```

```
<targetType> value
```

```
let netPrice = <number>getNetPrice(100, 0.05, false);
```

# 8 Generics

---



- TypeScript Generics
- Generic Constraints
- Generic Interfaces
- Generic Classes



# Generics – generici

---

- TypeScript generici omogućavaju da se piše ponovo iskoristiva (reusable) i generalizovana forma funkcija, klasa, i interfejsa.
- Podupiru porvere tipa u vreme kompajliranja.
- Eliminišu type casting.
- Omogućavaju implementaciju generičkih algoritama.

```
function getRandomElement<T>(items: T[]): T {  
    let randomIndex = Math.floor(Math.random() * items.length);  
    return items[randomIndex];  
}
```

# Generičke klase



- Generičke klase imaju listu generičkih tipova parametara u uglastim zagradama `<>` koji slede iza naziva klase.
- TypeScript omogućava da se ima više generičkih tipova u listi tipova parametara.

```
class className<T>{  
    //...  
}
```

```
class Stack<T> {  
    private elements: T[] = [];  
  
    constructor(private size: number) {  
    }  
    isEmpty(): boolean {  
        return this.elements.length === 0;  
    }  
    isFull(): boolean {  
        return this.elements.length === this.size;  
    }  
    push(element: T): void {  
        if (this.elements.length === this.size) {  
            throw new Error('The stack is overflow!');  
        }  
        this.elements.push(element);  
    }  
  
    pop(): T {  
        if (this.elements.length === 0) {  
            throw new Error('The stack is empty!');  
        }  
        return this.elements.pop();  
    }  
}
```




# Generički interfejsi

- Kao i klase, interfejsi takođe mogu biti generički.

```
interface Pair<K, V> {  
    key: K;  
    value: V;  
}
```

```
let month: Pair<string, number> = {  
    key: 'Jan',  
    value: 1  
};
```

```
interface Collection<T> {  
    add(o: T): void;  
    remove(o: T): void;  
}
```



```
interface interfaceName<T> {  
    // ...  
}
```

```
class List<T> implements Collection<T>{  
    private items: T[] = [];  
  
    add(o: T): void {  
        this.items.push(o);  
    }  
  
    remove(o: T): void {  
        let index = this.items.indexOf(o);  
        if (index > -1) {  
            this.items.splice(index, 1);  
        }  
    }  
}
```

## 9 TypeScript moduli

---



- Od ES6, JavaScript je počeo da podržava module kao izvorni deo jezika.
- TypeScript deli isti koncept modula sa JavaScript-om.
- TypeScript modul može da sadrži i deklaracije i kod.
- Modul se izvršava unutar sopstvenog opsega, a ne u globalnom opsegu. To znači da kada deklarirate promenljive, funkcije, klase, interfejse, itd., u modulu, one nisu vidljive izvan modula osim ako ih eksplicitno ne izvezete pomoću naredbe **export**.
- S druge strane, ako želite da pristupite promenljivim, funkcijama, klasama itd. iz modula, morate da ih uvezete pomoću naredbe **import**.
- Kao ES6, kada TypeScript fajl sadrži import ili export najvišeg nivoa, tretira se kao modul.

# Kreiranje novog modula



- Upotreba ključne reči **export**

```
export interface Validator {  
    isValid(s: string): boolean  
}
```

```
interface Validator {  
    isValid(s: string): boolean  
}  
  
export { Validator };
```

```
interface Validator {  
    isValid(s: string): boolean  
}  
  
export { Validator as StringValidator };
```



# Import modula

- Da bi se učitao modul, koristi se import naredba. Sledeći primer kreira novi modul EmailValidator.ts koji koristi Validator.ts modul.

```
import { Validator } from './Validator';

class EmailValidator implements Validator {
  isValid(s: string): boolean {
    const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
    return emailRegex.test(s);
  }
}

export { EmailValidator };
```

```
import { Validator as StringValidator } from './Validator';
```

# Rezime

---



- TypeScript deli isti koncept modula kao i ES6 moduli. Modul može da sadrži i deklaracije i kod.
- U modulu, promenljive, funkcije, klase, interfejsi, itd., izvršavaju se u sopstvenom opsegu, a ne u globalnom opsegu.
- Naredba **export** se koristi za izvoz promenljivih, funkcija, klasa, interfejsa, tipova itd. iz modula.
- Naredba **import** se koristi da bi pristupili exportima iz drugih modula.

# 10 TypeScript i Node JS



- Kreiranje strukture projekta
- Konfigurisanje TypeScript kompajlera
- Instaliranje Node.js modula
  - `npm install`

```
npm install --g nodemon concurrently
```

- `-g` flag instruiše npm da instalira module globalno, što omogućava da se koriste i u drugim projektima.

- Pokretanje

```
npm start
```

```
✓ NODETS
```

```
> build
```

```
> src
```

```
tsc --init
```

```
✓ NODETS
```

```
> build
```

```
> src
```

```
{-} tsconfig.json
```



# HVALA NA PAŽNJI

---

Pitanja?

---