TypeScript iz ugla Angulara

Jedna od osnovnih osobenosti kreiranja aplikacija korišćenjem Angulara jeste upotreba TypeScript jezika. Angular je u potpunosti napisan korišćenjem TypeScripta, a preporuka je da se TypeScript koristi i prilikom pisanja Angular aplikacija. To nije obaveza, zato što je programsku logiku Angular aplikacija moguće pisati i korišćenjem čistog JavaScript jezika. Ipak, kao što je rečeno, to nije preporučeni način za korišćenje Angulara, a pored toga, kompletna dokumentacija i svi primeri koje možete pronaći na internetu podrazumevaju korišćenje TypeScripta. Stoga ćemo i mi za razvoj Angular aplikacija da koristimo TypeScript.

U prethodnoj lekciji ste već mogli da vidite prve primere koda koji je napisan TypeScript jezikom, i to unutar fajlova koje je za nas automatski generisao Angular CLI. Na primer, evo sadržaja fajla `app.component.ts`:

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'my-first-app';
}
```

TypeScript kod možete pronaći i u svim ostalim fajlovima koji poseduju ekstenziju `.ts`., zato što je to ekstenzija koja se koristi za čuvanje programskog koda TypeScript jezika. Na primer, evo sadržaja još jednog veoma značajnog fajla za izvršavanje naše prve Angular aplikacije (`app.module.ts`):

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

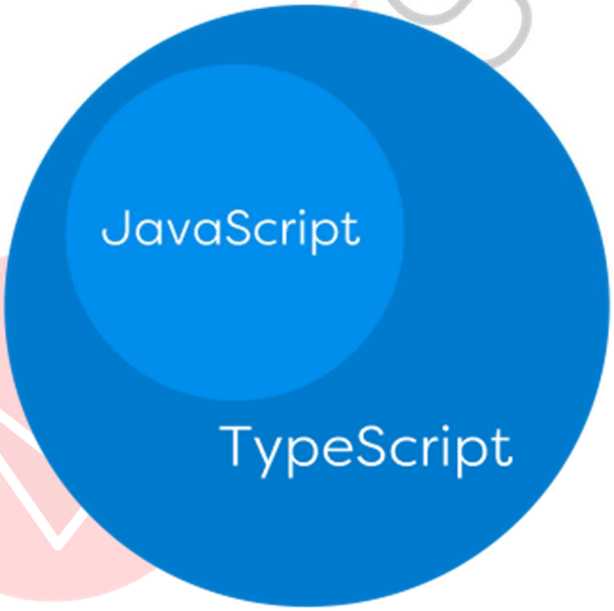
Da bismo lakše razumeli kako funkcioniše logika prisutna u upravo prikazanim fajlovima, neophodno je da se upoznamo sa osnovnim osobenostima TypeScript jezika. Tome će biti posvećena kompletna sledeća lekcija.

Šta je TypeScript?

TypeScript je 2012. godine kreirala kompanija Microsoft kako bi omogućila kreiranje prezentacione logike web aplikacija korišćenjem jezika sa bogatijim skupom mogućnosti od onih koje je u tom trenutku nudio JavaScript jezik.

Slika 13.1. TypeScript – logo

TypeScript je nadskup (engl. *superset*) JavaScript jezika. To praktično znači da TypeScript proširuje sintaksu JavaScript jezika uvođenjem dodatnih jezičkih funkcionalnosti i mogućnosti. To takođe znači i da je svaki JavaScript kôd ujedno potpuno validan TypeScript (slika 13.2).



Slika 13.2. TypeScript je nadskup jezika JavaScript

Iako je svaki JavaScript kod ujedno validan TypeScript, obrnuto ne važi, pa web pregledači nisu u stanju da razumeju programski kod napisan TypeScript jezikom. Stoga je pre nego što web pregledači izvrše kod, TypeScript potrebno prevesti u JavaScript, korišćenjem alata koji se nazivaju transkompajleri.

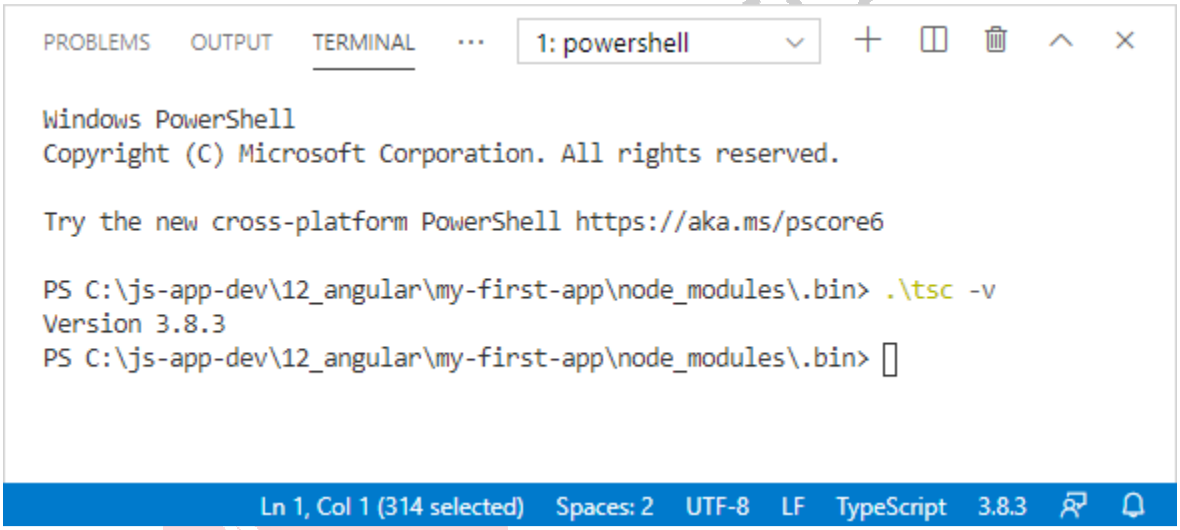
Prevođenje TypeScripta u JavaScript

Prilikom kreiranja Angular aplikacija, mi ne moramo da razmišljamo o procesu prevođenja TypeScript koda u JavaScript, zato što su svi potrebni alati unapred pripremljeni i konfigurisani unutar Angular CLI-ja. Ipak, alat za prevođenje TypeScripta u JavaScript se prilikom generisanja Angular projekta smešta lokalno, unutar takvog projekta, odnosno u njegov folder `node_modules/.bin`.

Kako bismo se uverili da unutar ovakvog foldera stvarno postoji *TypeScript Compiler* (`tsc`), dovoljno je pozicionirati se unutar takvog foldera i uputiti sledeću komandu:

```
tsc -v
```

Ukoliko ovakvu komandu želite da uputite korišćenjem Visual Studio Terminala na operativnom sistemu Windows, odnosno korišćenjem PowerShella, neophodno je navesti prefiks `.\`, baš kao na slici 13.3, zato što PowerShell iz sigurnosnih razloga podrazumevano blokira izvršavanje komande.



Slika 13.3. Provera raspoloživosti TypeScript kompajlera

Naravno, TypeScript kompajler je moguće instalirati i kao globalni npm paket, na sledeći način:

```
npm install -g typescript
```

TypeScript kompajler se može upotrebiti na sledeći način:

```
tsc index.ts
```

Na ovaj način će biti obavljeno prevođenje TypeScript koda koji se nalazi unutar fajla `index.ts`, unutar foldera u kome se nalazite prilikom upućivanja ovakve komande.

Pitanje

TypeScript je jezik koji je kreirala kompanija:

a) Microsoft
b) Google
c) Apple
d) Adobe

Objašnjenje:

TypeScript je 2012. godine kreirala kompanija Microsoft kako bi omogućila kreiranje prezentacione logike web aplikacija korišćenjem jezika sa bogatijim skupom mogućnosti od onih koje je u tom trenutku nudio JavaScript jezik.

Šta TypeScript izdvaja od JavaScripta?

Osnovne mogućnosti koje TypeScript izdvajaju od JavaScript jezika su:

- sistem statičkih tipova,
- interfejsi,
- enumeracije,
- dekoratori,
- prostori imena.

Pored ovih mogućnosti, koje ne postoje unutar JavaScript jezika, TypeScript omogućava korišćenje i svih onih jezičkih elemenata koji su tokom vremena uvršteni u JavaScript:

- klase,
- ES6 moduli,
- arrow funkcije,
- opcioni parametri funkcija i njihove podrazumevane vrednosti.

TypeScript sistem tipova

Najznačajnija osobina koja izdvaja TypeScript od JavaScripta jeste postojanje sistema statičkih tipova. Naime, vi već znate da je JavaScript jezik sa dinamičkim tipovima. To praktično znači da JavaScript jezik tipovima rukuje automatski, odnosno da se oni ne definišu eksplicitno prilikom deklaracije promenljivih ili konstanti. Takva osobina doprinosi pisanju jednostavnijeg koda, ali sa druge strane otežava pronalazak grešaka tokom pisanja koda, jer razvojni alati ne mogu da detektuju eventualne greške koje mogu nastati usled tipske nekompatibilnosti. Upravo zbog toga, TypeScript je strogo tipizirani jezik, koji omogućava proveru tipova za vreme procesa kompajliranja. To omogućava da se eventualne greške pronađu pre nego što kod završi u produkcionom okruženju.

Evo osnovnog primera koji odslikava šta podrazumeva sistem statičkih tipova:

```
var message:string = "Hello World";
```

Ovo je primer deklarisanja i inicijalizovanja jedne promenljive korišćenjem TypeScript jezika. Kao što vidite, razlika je u odeljku kojim se eksplicitno definiše tip promenljive (slika 13.4).

Slika 13.4. Deklarisanje i inicijalizovanje promenljive TypeScript

Tip promenljive u TypeScriptu definiše se nakon njenog naziva. Naziv i tip se odvajaju karakterom dve tačke (:).

Kako bi se ovakav kod preveo u JavaScript (pod uslovom da ste TypeScript kompajler instalirali globalno), dovoljno je da se pozicionirate u ovaj folder i da uputite sledeću komandu:

```
tsc index.js
```

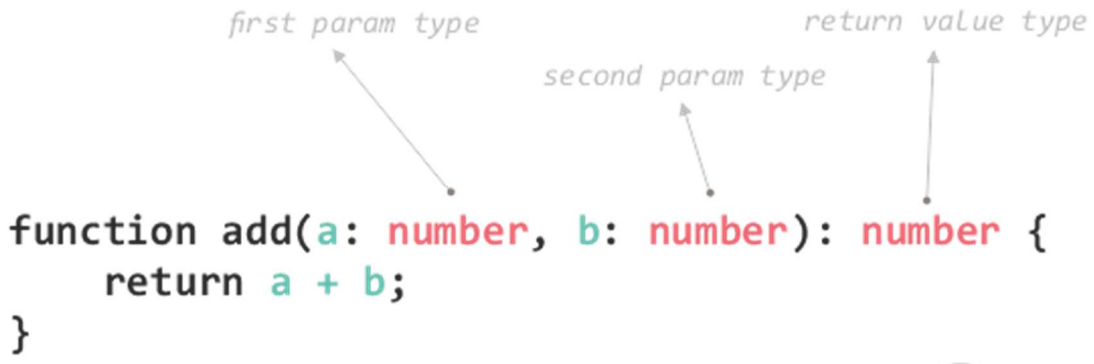
Izvršavanjem ove komande unutar foldera u kome se nalazi `index.ts` fajl pojaviće se i fajl `index.js`, sa sledećim sadržajem:

```
var message = "Hello World";
```

Naravno, TypeScript sistem tipova nije ograničen samo na promenljive, već se koristi na svim onim mestima na kojima se u kodu pojavljuju promenljive. Evo kako može da izgleda definisanje jedne jednostavne funkcije u TypeScriptu:

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

Reč je o funkciji koja prihvata dva ulazna parametra, sabira ih i rezultat emituje kao povratnu vrednost. Korišćenjem sistema tipova koje TypeScript poseduje definisani su tipovi svih ulaznih i izlaznih parametara (slika 13.5).



Slika 13.5. Struktura TypeScript funkcije

Prvo je unutar zagrada, nakon naziva promenljivih (`a`, `b`) i karaktera dve tačke, postavljen tip `number` za oba ulazna parametra. Tip povratne vrednosti kompletne funkcije definisan je nakon njenog naziva i parametara na identičan način – navođenjem karaktera dve tačke, nakon koga sledi odrednica koja definiše tip.

Koje tipove poznaje TypeScript?

Prilikom pisanja TypeScript koda moguće je koristiti sve one osnovne tipove na koje smo navikli i prilikom rada sa JavaScriptom:

- `boolean`,
- `number`,
- `string`,
- `null`,
- `undefined`.

Evo primera definisanja TypeScript promenljivih korišćenjem standardnih, prostih tipova:

```
let isDone: boolean = false;
let width: number = 33;
let message: string = "Hello World";
let myVar: null = null;
let myVar2: undefined = undefined;
```

Nakon prevođenja u JavaScript, ovakav kod će da izgleda ovako:

```
var isDone = false;
var width = 33;
var message = "Hello World";
var myVar = null;
var myVar2 = undefined;
```

Konfigurisanje TypeScript prevodioca

Ukoliko pogledate rezultat upravo prikazanog primera, možete da primetite jednu nelogičnost. Sve promenljive koje smo kreirali korišćenjem TypeScript koda deklarisane su korišćenjem ključne reči `let`, ali je tokom prevođenja takva ključna reč zamenjena ključnom rečju `var`. Zbog čega se ovo dešava?

TypeScript kompajler podrazumevano obavlja prevođenje u onaj kôd koji je kompatibilan sa ES3 verzijom JavaScript jezičke specifikacije. U takvoj verziji još nisu postojale ključne reči `let` i `const`, pa je upravo to razlog zbog koga smo dobili rezultat kao u prethodnom primeru.

Kako bi se ovakvo podrazumevano ponašanje promenilo, moguće je definisati opcije za konfigurisanje kompajliranja, korišćenjem fajla `tsconfig.json`, koji je potrebno postaviti u koreni folder TypeScript projekta. U prethodnoj lekciji ste videli da takav fajl postoji i unutar projektne strukture Angular CLI. Mi ga možemo i samostalno definisati, a kako bismo uticali na proces kompajliranja, dovoljno je da unutar takvog fajla postavimo sledeći sadržaj:

```
{
  "compilerOptions": {
    "module": "esnext",
    "target": "es2015",
    "lib": [
      "es2018",
      "dom"
    ]
  }
}
```

Na ovaj način smo definisali nekoliko podešavanja koja će uticati na proces kompajliranja TypeScripta:

- `module` – ovim ključem i njegovom vrednošću rekli smo da kao modularni sistem želimo da koristimo ES6, a ne CommonJS module,
- `target` – korišćenjem ovoga ključa definisana je ECMAScript verzija, koja će biti korišćena za generisanje koda; postavili smo `es2015`, odnosno `es6`,
- `lib` – ključ čija je vrednost niz biblioteka koje definišu skup funkcionalnosti koje će moći da se koriste prilikom pisanja TypeScript koda; postavili smo `es2018`, što znači da ćemo biti u mogućnosti da prilikom formulacija TypeScript koda koristimo sve one jezičke elemente koji su definisani u ECMAScript specifikacijama, zaključno sa verzijom ES2018.

Fajl `tsconfig.json` je potrebno postaviti na korenu putanju projekta, a kako bi podešavanja unutar njega bila primenjena tokom prevođenja, dovoljno je uputiti komandu bez navođenja putanje do fajla:

```
tsc
```

Naravno, potrebno je da budete pozicionirani unutar korenog foldera projekta, unutar koga se nalazi fajl za podešavanje i `index.ts` fajl koji je potrebno prevesti.

Pored već poznatih osnovnih JavaScript tipova, TypeScript uvodi još nekoliko takvih tipova:

- `any`,
- `void`,
- `never`.

Pored ovih osnovnih tipova, TypeScript uvodi i dva dodatna složena tipa:

`Enum` i
`Tuple`.

any

TypeScript omogućava da se obavi deklarisanje neke promenljive bez definisanja njenog tipa, baš kao što se to obavlja u JavaScript jeziku. Ipak, u TypeScriptu to se postiže navođenjem specifičnog tipa `any`:

```
let notSure: any = 6;
```

Ukoliko nismo sigurni kojeg tipa će biti vrednost neke promenljive, moguće je iskoristiti upravo prikazani tip `any`. Na taj način promenljiva će se ponašati kao da je reč o običnoj JavaScript promenljivoj, odnosno nad njom se neće sprovoditi tipske provere prilikom prevođenja.

void

Tip `void` predstavlja suprotnost upravo prikazanom tipu `any`. Drugim rečima, ukoliko je u TypeScriptu potrebno naglasiti odsutnost tipa, to je moguće učiniti ključnom rečju `void`.

Ključna reč `void` najčešće se koristi prilikom definisanja funkcija koje nemaju povratnu vrednost:

```
function sayHello(name: string): void {
    console.log("Hello " + name);
}
```

Ovo je primer TypeScript funkcije bez povratne vrednosti. Ona ne poseduje povratnu vrednost, zato što parametar koji dobija ispisuje direktno unutar konzole. Eksplicitnim naglašavanjem povratne vrednosti uvodi se dodatna sigurnost prilikom pisanja koda, jer postojanje `void` povratne vrednosti mora biti propraćeno odsustvom `return` naredbe unutar tela funkcije.

never

U TypeScriptu, tip `never` se uglavnom koristi da označi povratnu vrednost funkcija koje se nikada neće završiti uspešno. To se može dogoditi iz dva razloga: kada se unutar funkcije uvek emituje izuzetak ili kada funkcija poseduje beskonačnu petlju.

Ovo je primer funkcije koja uvek izbacuje izuzetak:

```
function sayHello(name: string): never {
    throw new Error(message);
}
```

Funkcija `sayHello()` uvek emituje izuzetak, zato što je unutar njenog tela postavljena naredba `throw` izvan bilo kakvih blokova za kontrolu toka.

Primer funkcije sa beskonačnom petljom može da izgleda ovako:

```
function infiniteLoop(): never {
    while (true) {
    }
}
```

Funkcija `infiniteLoop()` unutar sebe poseduje beskonačnu petlju, zato što je za uslov `while` petlje direktno definisana konstantna `true` vrednost.

Array i Tuple

Kreiranje nizova se u TypeScriptu može postići korišćenjem dva tipa: već poznatog tipa `Array` i korišćenjem još jednog specifičnog TypeScript tipa – `Tuple`.

Nizovi se kreiraju kao i unutar JavaScripta, uz dodatak definisanja tipa niza:

```
let myArray: number[] = [8, 7, 6];
```

Ovo je sada jedan niz, kod koga je eksplicitno definisano da će moći da prihvati samo podatke tipa `number`. Unutar ovakvog niza neće biti moguće spakovati podatak koji nije `number` tipa:

```
let myArray: number[] = [8, "7", 6];
```

Prilikom prevođenja ovakvog koda biće prikazana jasna poruka o grešci:

```
error TS2322: Type 'string' is not assignable to type 'number'.
```

Tip nizova u TypeScriptu je moguće definisati na još jedan način, upotrebom sledeće sintakse:

```
let myArray: Array<number> = [8, 7, 6];
```

Ovoga puta se tip niza definiše na nešto drugačiji način: navođenjem objektnog tipa `Array`, nakon koga se u trouglastim zagradama definiše tip elemenata koji se mogu naći unutar niza (`number`). Oba prikazana pristupa za kreiranje nizova su ekvivalentna.

Još jedan način za definisanje nizova u TypeScript podrazumeva korišćenje tipa **Tuple**. Reč je zapravo o tipu koji omogućava kreiranje nizova sa unapred utvrđenom dužinom i tipovima koji se unutar njega mogu naći:

```
let myTuple: [string, boolean];
```

Ovo je sada jedan `Tuple` tip podatka. Naredba ilustruje deklaraciju takvog tipa, kojom je rečeno da će se unutar promenljive `myTuple` moći naći niz od dva elementa, koji moraju biti tipa `string` i `boolean`. Stoga ispravna inicijalizacija ovakve promenljive može da izgleda ovako:

```
myTuple = ['condition1', false] ;
```

Unutar ovakve promenljive nije moguće smestiti više od dva elementa:

```
myTuple = ['condition1', false, 'condition2', true];
```

Ovakva inicijalizacija proizvodi grešku:

```
error TS2322: Type '[string, false, string, true]' is not assignable to
type '[string, boolean]'
```

Takođe, unutar promenljive `Tuple` tipa vrednosti se moraju definisati baš onim redosledom kojim su i navedene prilikom deklaracije:

```
myTuple = [false, 'condition1'];
```

I ovakav primer će da proizvede grešku, zato što vrednosti nisu definisane odgovarajućim redosledom:

```
error TS2322: Type 'string' is not assignable to type 'boolean'.
```

Enum

Po ugledu na većinu programskih jezika koji se koriste za programiranje pozadinske logike web aplikacija (C#, Java, PHP…), TypeScript uvodi tip `enum`. Reč je o tipu koji omogućava kreiranje enumeracija, što su zapravo konstante kojima je moguće pristupati korišćenjem razumljivih tekstualnih identifikatora.

Primer jedne enumeracije može da izgleda ovako:

```
enum Day { Monday = 1, Tuesday = 2, Wednesday = 3, Thursday = 4, Friday
= 5, Saturday = 6, Sunday = 7 };
```

Ovakvom enumeracijom se predstavljaju dani u sedmici. Pri tome se svaki dan sastoji iz enumeracione konstante, odnosno tekstualnog identifikatora i njegove konkretne numeričke vrednosti. Na ovaj način se olakšava rukovanje numeričkim vrednostima koje predstavljaju dane u sedmici, s obzirom na to da, same po sebi, takve vrednosti ne nude nikakvu dodatnu informaciju o entitetima koje opisuju. Korišćenjem enumeracije omogućeno je da se u programu njima rukuje na mnogo razumljiviji način:

```
let day: Day = Day.Monday;
```

Na ovaj način je deklarisana promenljiva day, koja ima tip Day, odnosno tip upravo kreirane enumeracije.

Korisnički definisani tipovi

enum je prvi od tipova, koji u TypeScriptu omogućava kreiranje korisnički definisanih tipova. Jednostavno, u upravo prikazanom primeru, enum je iskorišćen za kreiranje našeg tipa, koji smo imenovali nazivom Day. Takav tip smo postavili za tip promenljive day.

Enumeracije su vrlo korisne prilikom evaluacije uslovne promenljive unutar switch konstrukcija:

```
let day: number;

switch (day) {
    case 1:
        console.log("Today is Monday!");
        break;
    case 2:
        console.log("Today is Tuesday!");
        break;
    case 3:
        console.log("Today is Wednesday!");
        break;

    //...
}
```

Ovo je primer switch konstrukcije, unutar koje je uslovna promenljiva number tipa. Stoga se kontrolne vrednosti definišu korišćenjem celih brojeva (1, 2, 3…), što svakako loše utiče na opštu čitljivost koda. Čitljivost se može unaprediti korišćenjem enumeracije:

```
enum Day { Monday = 1, Tuesday = 2, Wednesday = 3, Thursday = 4, Friday
= 5, Saturday = 6, Sunday = 7 };

let day: Day;

switch (day) {
    case Day.Monday:
        console.log("Today is Monday!");
        break;
    case Day.Tuesday:
        console.log("Today is Tuesday!");
        break;
    case Day.Wednesday:
        console.log("Today is Wednesday!");
        break;
    //...
}
```

Sada je kontrolna promenljiva switch konstrukcije zamenjena tipom koji predstavlja jednu enumeraciju. To nam je omogućilo da, umesto celobrojnih vrednosti, koristimo mnogo razumljivije tekstualne identifikatore.

Unija tipova

U TypeScriptu je moguće definisati promenljivu koja može imati jedan od nekoliko unapred definisanih tipova. Takva osobina jezika naziva se unija tipova, a evo kako izgleda u praksi:

```
function square(a: (number | string)): number {
    return <number>a * <number>a;
}
```

Funkcija `square()` poseduje jedan ulazni parametar čiji tip je definisan korišćenjem unije tipova. Na ovaj način je rečeno da će ulazni parametar funkcije (koji je imenovan sa `a`), moći da bude tipa `number` ili `string`.

Kastovanje tipova – Type assertions

U upravo prikazanom primeru iskorišćena je još jedna specifična funkcionalnost TypeScript jezika. Reč je o jednoj vrsti kastovanja, kojom je moguće uticati na proces provere koda koji se obavlja tokom kompajliranja TypeScripta u JavaScript.

Takav mehanizam možete da vidite unutar tela prikazane funkcije `square()`, gde je ispred naziva promenljive `a`, unutar trouglastih zagrada (<>), postavljen tip `number`. Bez ove tipske izjave (engl. *type assertion*), prevođenje koda funkcije `square()` ne bi moglo da bude obavljeno, zato što TypeScript obavlja proveru tipova. S obzirom na to da TypeScript zna da promenljiva `a` može biti i tipa `string`, on ne dozvoljava da ona učestvuje u jednom aritmetičkom izrazu (množenje). Ipak, korišćenjem funkcionalnosti *type assertions,* mi smo TypeScript kompajleru rekli da znamo da će tip promenljive biti `number`, s obzirom na to da se oslanjamo na jednu od osnovnih osobina JavaScript jezika, koja se tiče automatskog prevođenja `string` podataka u `number`, prilikom izvršavanja aritmetičkih operacija.

Funkcije

Funkcije u TypeScriptu funkcionišu identično funkcijama u JavaScript jeziku, uz nekoliko dodataka. Za početak, tu je mogućnost definisanja tipova ulaznih i izlaznih parametara:

```
function greet(greeting: string, name: string ) : string {
    return greeting + ' ' + name + '!';
}
```

Ovo je funkcija koja prihvata dva `string` parametra i emituje povratnu vrednost tipa `string`. Obratite pažnju na tipove koji su definisani nakon parametara i nakon deklaracije funkcije.

TypeScript omogućava definisanje fakultativnih parametara, odnosno parametara čije vrednosti nije obavezno definisati. Kako bi se jedan parametar obeležio kao opcioni, dovoljno je, nakon njegovog naziva, postaviti karakter upitnik - **?**:

```
function greet(greeting: string, name?: string): string {
    return greeting + ' ' + name + '!';
}
```

Podrazumevanu vrednost funkcijskih parametara je u TypeScriptu moguće definisati na sledeći način:

```
function greet(name: string, greeting: string = "Hello"): string {
    return greeting + ' ' + name + '!';
}
```

Nakon deklaracije parametra `greeting`, postavljen je karakter `=` i nakon njega podrazumevana vrednost parametra. Na ovaj način će parametar `greeting` imati vrednosti `Hello`, čak i kada se ona ne prosledi.

Klase

Veći deo sintakse za kreiranje klasa deli se između TypeScript i JavaScript jezika, pre svega zbog činjenice da su klase uvršćene u jezičku specifikaciju još sa pojavom njene verzije ES6. Stoga je u TypeScriptu jednostavnu klasu moguće definisati isto kao i u JavaScriptu:

```
class Employee {

    code: number;
    name: string;

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

let emp = new Employee(111, "John");
```

public, private, protected

Najznačajnije unapređenje koje TypeScript uvodi prilikom rada sa klasama tiče se mogućnosti definisanja modifikatora pristupa, s obzirom na to da se na takvoj funkcionalnosti još radi unutar ECMAScript specifikacije i sigurno još neko vreme neće biti široko dostupna unutar web pregledača. Modifikatori pristupa omogućavaju definisanje privatnih, javnih ili zaštićenih klasnih elemenata, čime se upravlja njihovom vidljivošću izvan granica klase ili objekta koji se kreira na osnovu takve klase.

TypeScript modifikatori pristupa definišu se korišćenjem specijalnih ključnih reči koje se postavljaju ispred naziva svojstva ili metode:

- `public` – definiše javnu dostupnost,
- `private` – definiše dostupnost samo unutar klase,
- `protected` – definiše dostupnost unutar matične klase i svih klasa koje je nasleđuju.

Primer jedne TypeScript klase sa nekoliko modifikatora pristupa može da izgleda ovako:

```
class Employee {

    protected code: number;
    private firstName: string;
    private lastName: string;

    name: string;

    constructor(code: number, firstName: string, lastName: string) {
        this.code = code;
        this.firstName = firstName;
        this.lastName = lastName;

              this.name = firstName + " " + lastName;
    }
}
```

Svojstvo `code` je označeno ključnom rečju `protected`, što znači da će njemu moći da se pristupi samo unutar matične klase i izvedenih klasa. Svojstvima `firstName` i `lastName` moći će da se pristupi samo unutar matične klase, zbog postojanja modifikatora pristupa `private`. Na kraju, podrazumevano, klasni članovi bez ikakvog modifikatora pristupa su javni, pa je tako svojstvu `name` moguće pristupiti i izvan klase.

Stoga će naredne naredbe da proizvedu greške:

```
let emp = new Employee(111, "John", "Torrance");

emp.code = 235353;
emp.firstName = "New Name";
```

Nakon instanciranja klase, pokušavamo da redefinišemo vrednosti svojstava `code` i `firstName`. Ipak, to se ne može obaviti zbog postojanja modifikatora pristupa `protected` i `private`:

```
    Property 'code' is protected and only accessible within class
'Employee' and its subclasses.
    Property 'firstName' is private and only accessible within class
'Employee'.
```

readonly

Još jedan dodatak koji TypeScript uvodi u oblast rada sa klasama jeste definisanje svojstava čije je vrednosti moguće samo čitati. Kako bi se jedno svojstvo definisalo kao takvo, dovoljno je ispred njegove deklaracije postaviti ključnu reč `readonly`:

```
class Employee {

    protected code: number;
    private firstName: string;
    private lastName: string;
```

```
    readonly name: string;

    constructor(code: number, firstName: string, lastName: string) {
        this.code = code;
        this.firstName = firstName;
        this.lastName = lastName;

        this.name = firstName + " " + lastName;
    }
}
```

Sada je svojstvo `name` označeno ključnom rečju `readonly`. To će ukinuti mogućnost izmene njegove vrednosti, dok će čitanje i dalje biti moguće:

```
let emp = new Employee(111, "John", "Torrance");
console.log(emp.name);
```

Ovakav kod prevodi se bez problema, što se ne može reći i za sledeću situaciju:

```
let emp = new Employee(111, "John", "Torrance");
emp.name = "New Name";
```

S obzirom na to da se pokušava postaviti vrednost `readonly` svojstva, prilikom prevođenja se dobija greška:

```
Cannot assign to 'name' because it is a read-only property.
```

Interfejsi

Jedna od najznačajnijih funkcionalnosti TypeScript jezika, koja izvorno ne postoji u JavaScriptu, odnosi se na mogućnost definisanja interfejsa. Interfejsima je na lak način moguće predstaviti kostur jednog tipa. To praktično znači da su interfejsi mehanizam pomoću koga je moguće reći da će neki tip imati određena svojstva i metode. Ipak, unutar interfejsa se definišu samo potpisi takvih tipskih elemenata, bez konkretne implementacije. Naredni primer će vam pomoći da razumete ono što je sada rečeno:

```
function sayHello(personObj: { name: string }) {
    console.log("Hello" + name);
}
```

Ovo je jedna TypeScript funkcija koja kao svoj parametar može da prihvati objekat koji unutar sebe mora imati svojstvo `name`. Stoga bi legalan način korišćenja ove funkcije mogao da izgleda ovako:

```
function sayHello(personObj: { name: string }) {
    console.log("Hello" + name);
}

let person = {
    name: "Ben",
    email: "email@email.com"
}

sayHello(person);
```

Primeru je dodato kreiranje jednog objekta, zatim je takav objekat prosleđen funkciji `sayHello()`. S obzirom na to da objekat poseduje svojstvo `name`, sve prolazi bez problema.

Upravo prikazani primer mnogo elegantnije je moguće rešiti upotrebom interfejsa. Za navođenje kompletne specifikacije objekta koji funkcija može da prihvati moguće je upotrebiti interfejs:

```
interface Named {
    name: string;
}

function sayHello(personObj: Named) {
    console.log("Hello" + name);
}
```

Možete videti da se interfejsi kreiraju slično klasama, uz razliku koja se odnosi na upotrebu ključne reči `interface` umesto ključne reči `class`. Unutar interfejsa koji je imenovan nazivom `Named`, definisano je jedno svojstvo – `name`. Zatim je ovakav interfejs iskorišćen za definisanje tipa ulaznog parametra funkcije `sayHello()`.

Iz ovog primera ste mogli da vidite da su interfejsi još jedan način koji omogućava kreiranje korisnički definisanih tipova. Mi smo u primeru definisali naš tip `Named`. Ipak, interfejsi omogućavaju kreiranje apstraktnih tipova, odnosno tipova koji, pored deklaracija svojstava i metoda, ne mogu posedovati logiku. Stoga se interfejsi primarno koriste za definisanje ugovora. Definisanje jednog takvog ugovora upravo je ilustrovao prethodni primer. Interfejs `Named` je ugovor kojim se definiše da će funkcija `sayHello()` kao svoj ulazni parametar moći da prihvati isključivo objekte koji poseduju svojstvo `name`.

Pored svojstava, unutar interfejsa se mogu definisati i metode:

```
interface Named {
    name: string;
    getName(): string;
}
```

Ovakav interfejs sada opisuje tip koji unutar sebe mora imati svojstvo `name` i metodu `getName()`.

Interfejse veoma često implementiraju same klase:

```
class Employee implements Named {

    protected code: number;
    name: string;

    getName(): string {
        return name;
    }
    constructor(code: number, name: string) {
        this.code = code;
        this.name = name;
    }
}
```

Implementiranje interfejsa od klase postiže se korišćenjem ključne reči `implements`. Kada jedna klasa implementira neki interfejs (kao što je u primeru klasa `Employee` implementirala interfejs `Named`), unutar nje se moraju implementirati svi elementi koji su deklarisani unutar interfejsa. To je u primeru i učinjeno, zato što klasa `Employee` poseduje svojstvo `name` i metodu `getName()`. Odsustvo bilo kog od ova dva elementa proizvelo bi grešku prilikom prevođenja:

```
error TS2420: Class 'Employee' incorrectly implements interface
'Named'.
  Property 'getName' is missing in type 'Employee' but required in type
'Named'.
```

Kao što je već rečeno, interfejsi su efikasan način za kreiranje novih, korisničkih tipova. Tako se može reći da će objekti koji se kreiraju korišćenjem klase `Employee` ujedno da budu i tipa `Named`. Stoga su obe sledeće linije validne:

```
let emp: Named = new Employee(111, "John Torrance");

let emp: Employee = new Employee(111, "John Torrance");
```

U prvoj naredbi promenljiva `emp` se tretira kao promenljiva tipa `Named`, a u drugoj kao promenljiva tipa `Employee`.

Vrhunac apstrakcije koja se može postići interfejsima jeste postojanje većeg broja različitih tipova koji implementiraju jedan isti interfejs. To se može uraditi i na primeru našeg `Named` interfejsa:

```
class Teacher implements Named {

    protected university: string;
    name: string;

    getName(): string {
        return name;
    }
    constructor(university: string, name: string) {
        this.university = university;
        this.name = name;
    }
}
```

Sada interfejs `Named` implementira još jedna klasa – `Teacher`. S obzirom na to da dva tipa implementiraju isti interfejs, sada se može napisati nešto ovakvo:

```
let employee: Named = new Employee(111, "John Torrance");
let teacher: Named = new Teacher("Oxford", "Ben Roland");
```

Iako čuvaju reference na objekte koji su kreirani različitim klasama, obe kreirane promenljive su identičnog tipa – `Named`. To je postignuto apstrakcijom kreiranom upotrebom interfejsa.

Dekoratori

Jedna od funkcionalnosti TypeScript jezika koja se intenzivno koristi prilikom razvoja Angular aplikacija jesu dekoratori. Dekoratori omogućavaju da se deklaracije klasa, funkcija i svojstava dekorišu specijalnim oznakama, koje se nazivaju anotacije. Takve oznake koriste se za to da na neki način transformišu, odnosno modifikuju element nad kojim su definisane.

Na primer, dekoratore u praksi možete da vidite na klasi koja predstavlja našu prvu Angular komponentu iz prethodne lekcije:

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'my-first-app';
}
```

Iznad deklaracije klase `AppComponent` možete videti jedan TypeScript dekorator – `@Component()`. Ovakav dekorator se koristi za to da jednu TypeScript klasu proglasi Angular komponentom. Bez ovakve dekoracije, `AppComponent` bi bila samo još jedna obična klasa i ne bi mogla da se posmatra u kontekstu razvoja Angulara. Pored proglašavanja jedne klase za Angular komponentu, `@Component()` dekorator obavlja i konfigurisanje takve komponente, definisanjem korenog elementa i putanja na kojima se nalazi šablon i njegova stilizacija. Ali kako sve ovo funkcioniše u pozadini, na nivou TypeScript jezika?

TypeScript dekoratori definišu se u sledećem obliku:

```
@expression
```

`expression` se zapravo odnosi na naziv funkcije koja će na neki način da transformiše dekorisani element. Tako se dekoratori u pozadini uvek realizuju korišćenjem funkcija sa logikom, koja obavlja neki specifičan zahvat nad elementom koji je dekorisan. Na primeru Angular komponenata, uz korišćenje `@Component()` dekoratora obavlja se definisanje osnovih osobina jedne Angular komponente.

Prostori imena i moduli

Osnovni TypeScript mehanizmi za grupisanje i organizaciju koda jesu moduli i prostori imena. Modularni sistem TypeScript zasniva se na ES6 modulima. Stoga je moguće koristiti sve one pristupe koji su prikazani u jednoj od prethodnih lekcija, a koji podrazumevaju upotrebu ključnih reči `import` i `export`. ES6 moduli se intenzivno koriste i prilikom razvoja Angular aplikacija, pa tako u gotovo svakom fajlu koji je generisan u prethodnoj lekciji možete videti naredbe za importovanje i eksportovanje. Na primer, unutar fajla za kreiranje glavne komponente naše Angular aplikacije napisano je:

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'my-first-app';
}
```

Funkcionalnosti ES6 modula su unutar fajla glavne komponente iskorišćene za uključivanje `@Component()` dekoratora iz modula `@angular/core`. Takođe, kako bi komponenta `AppComponent` mogla da se koristi i izvan ovakvog fajla, ispred deklaracije klase je postavljena ključna reč `export`. Ovi primeri importovanja i eksportovanja klasične su funkcionalnosti ES6 modula.

Pored modula, TypeScript poseduje još jedan mehanizam za grupisanje koda koji izvorno ne postoji u JavaScript jeziku. Reč je o prostorima imena. Prostori imena (engl. *namespaces*) omogućavaju grupisanje koda unutar JavaScript objekata, za razliku od grupisanja na nivou fajlova koje omogućavaju moduli.

Uz korišćenje TypeScripta prostor imena se može kreirati na sledeći način:

```
namespace MyNamespace {

    let myVar: number = 13;

}
```

Ovo je primer jednog prostora imena TypeScripta unutar koga je definisana jedna promenljiva. Prostori imena se u TypeScriptu kreiraju korišćenjem ključne reči **namespace**. Kada se ovakav kod prevede, dobija se sledeći JavaScript kod:

```
var MyNamespace;
(function (MyNamespace) {
    let myVar = 13;
})(MyNamespace || (MyNamespace = {}));
```

Možete videti da se prostori imena realizuju korišćenjem samopozivajućih funkcija. Takav pristup se inače vrlo često koristi i prilikom pisanja čistog JavaScripta, za izolovanje promenljivih i funkcija unutar jednog prostora imena.

Podrazumevano, elementima jednog prostora imena ne može se pristupiti izvan takvog prostora imena, osim ukoliko oni nisu prethodno eksportovani:

```
namespace MyNamespace {

    export let myVar: number = 13;

}
```

Korišćenjem ključne reči `export` označava se promenljiva `myVar` i na taj način se omogućava njeno korišćenje izvan prostora imena:

```
console.log(MyNamespace.myVar);
```

Na ovaj način prostori imena u TypeScriptu kombinuju mogućnosti ES6 modula i sopstvenih prostora imena.

Rezime

- TypeScript je 2012. godine kreirala kompanija Microsoft kako bi omogućila kreiranje prezentacione logike web aplikacija korišćenjem jezika sa bogatijim skupom mogućnosti od onih koje je u tom trenutku nudio JavaScript jezik.
- `.ts` je ekstenzija koja se koristi za čuvanje programskog koda TypeScript jezika.
- Web pregledači ne mogu da razumeju programski kod napisan TypeScript jezikom.
- Pre nego što web pregledač izvrši kod, TypeScript je potrebno prevesti u JavaScript, korišćenjem alata koji se nazivaju transkompajleri.
- Alat koji obavlja prevođenje TypeScript koda u JavaScript naziva se *TypeScript Compiler,* ili skraćeno `tsc`.
- TypeScript kompajler je moguće instalirati kao globalni npm paket korišćenjem komande `npm install -g typescript`.
- TypeScript je strogo tipizirani jezik, koji omogućava proveru tipova tokom procesa kompajliranja.
- Tip promenljive se u TypeScriptu definiše nakon njenog naziva, od koga se odvaja karakterom dve tačke.
- Opcije za konfigurisanje kompajliranja moguće je definisati korišćenjem fajla `tsconfig.json`, koji je potrebno smestiti unutar korenog foldera TypeScript projekta.
- TypeScript omogućava da se obavi deklarisanje neke promenljive bez definisanja njenog tipa, korišćenjem ključne reči `any`.
- Odsutnost tipa je u TypeScriptu moguće definisati ključnom rečju `void`.
- Ključna reč `void` najčešće se koristi prilikom definisanja funkcija koje nemaju povratnu vrednost.
- Tip `never` se koristi za to da označi povratnu vrednost funkcija koje se nikada neće završiti uspešno.
- `Tuple` je tip koji omogućava kreiranje nizova sa unapred utvrđenom dužinom i tipovima koji se unutar njega mogu naći.
- TypeScript uvodi tip `enum`, što su zapravo konstante kojima je moguće pristupati korišćenjem razumljivih tekstualnih identifikatora.
- U TypeScriptu je moguće definisati promenljivu koja može imati jedan od nekoliko unapred definisanih tipova, razdvajanjem tipova korišćenjem karaktera `|`.
- TypeScript omogućava definisanje fakultativnih parametara, odnosno parametara čije vrednosti nije obavezno definisati, korišćenjem karaktera `?`, nakon naziva parametra.
- Korišćenjem modifikatora pristupa `public`, `private` i `protected` moguće je uticati na dostupnost klasnih članova izvan granica klasa i objekata koji se kreiraju na osnovu takvih klasa.
- Interfejsi se u TypeScriptu definišu korišćenjem ključne reči `interface`.
- Implementiranje interfejsa od klase postiže se korišćenjem ključne reči `implements`.

- Dekoratori omogućavaju da se deklaracije klasa, funkcija i svojstava dekorišu specijalnim oznakama kojima se na kraju transformiše, odnosno modifikuje element nad kojim su definisane.
- Za grupisanje koda, TypeScript koristi ES6 module i prostore imena.
- Prostori imena su specifični za TypeScript, a omogućavaju grupisanje koda unutar JavaScript objekata, za razliku od grupisanja na nivou fajlova koje omogućavaju moduli.
- Prostori imena se u TypeScriptu kreiraju korišćenjem ključne reči `namespace`.