JavaScript TypeScript

Autori:
Milan Segedinac
Goran Savić

1. TypeScript

TypeScript je sintaktički nadskup programskog jezika JavaScript koji se može transpiliranjem prevesti u čist JavaScript. To znači da će svaki program napisan u JavaScript-u automatski biti i validan TypeScript program, ali i da će svaki program napisan u TypeScript-u moći da se prevede u JavaScript program.

U nastavku ove lekcije biće dat pregled najvažnijih svojstava koja uvodi TypeScript.

Tipovi

Svakako najznačajnija razlika u odnosu na JavaScript je u tome što TypeScript dozvoljava (ali nas ne obavezuje da!) da deklarišemo tipove. Podsetimo se da je JavaScript dinamički tipiziran jezik. To znači da vrednosti imaju tip, ali da, prilikom deklaracije varijable ne navodimo kog tipa je ta varijabla, pa jedna varijabla može u različitim delovima programa da čuva vrednosti koje su različitog tipa. Za razliku od toga, ukoliko deklarišemo tip varijable u TypeScript-u, ta varijabla ne može da primi vrednost drugog tipa.

Deklaracija varijable je slična kao i u JavaScriptu, jedino nakon naziva varijable sledi dvotačka i naziv tipa. Sintaksa je data listingom ispod.

```
var|let|const <nazivVarijable> : <tip> = <vrednost>;
```

Listing – sintaksa deklaracije varijable

Činjenica da nam TypeScript omogućuje dekaraciju tipova omogućuje da se tipovi zadaju i prilikom deklaracije funkcije. Deklaracija funkcije počinje rečju function nakon čega u zagradi sledi lista parametara. Svaki parametar može da ima tip koji se zadaje tako što nakon naziva parametra navedemo dvotačku i naziv tipa. Nakon liste parametara sledi dvotačka pa naziv tipa povratne vrednosti funkcije. Nakon toga sledi telo funkcije u vitičastim zagradama.

```
function <nazivFunkcije>(<parametar1>: <tipParametra1>,

 <parametar2>: <tipParametra2>, ...): <tipPovratneVrednosti>
{
     <telo funkcije>
}
```

U nastavku je dat pregled TypeScript tipova.

Boolean

Booelan je logički tip podataka koji ima vrednosti `true` i `false`. Na primer, varijablu `studira` mogli bismo deklarisati kao boolean sledećim iskazom

```
let studira: boolean = false;
```

Listing - `boolean` tip

Koristili smo let deklaraciju čime smo odredili da opseg vidljivosti varijable `studira` bude blok u kom je varijabla definisana. Nakon naziva varijable sledi dvotačka i naziv tipa varijable.

Number

U TypeScript-u je moguće deklarisati varijable brojčanog tipa. Listing ispod prikazuje primer takve deklaracije.

```
let ocena: godinaStudija = 2;
```

Listing - `number` tip

Kao i u JavaScript-u postoji jedan tip za predstavljanje i celih i razlomljenih brojeva.

String

Lisitngom ispod prikazana je deklaracija string varijable.

```
let ime: string = 'Pera Peric';

console.log('Zovem se '+ime+' i upisan sam na '+godinaStudija+'.
godinu studija');
```

Listing - `string` tip

Kao i u JavaScript-u, prilikom „sabiranja“ stringa i broja izvršiće se implicitna konverzija broja u string, pa će se uraditi konkatenacija.

Niz

Podsetimo se da su, u JavaScript-u, nizovi *heterogene* kolekcije. To znači da elementi niza mogu da budu različitog tipa. U TypeScript-u nizovi se mogu deklarisati kao *homogene* kolekcije, odnosno moguće je zadati tip elemenata niza. Listingom ispod prikazana je deklaracija niza.

```
let ocene: number[] = [8,9,8];

for(let ocena of ocene){
      console.log(ocena);
}
```

Listing - niz brojeva

Deklarisali smo varijablu ocene kao niz numeričkih vrednosti i odmah joj dodelili vrednost.

Tuple

Često nam se može desiti da nam treba heterogena kolekcija u kojoj unapred znamo kog tipa će biti vrednosti na kojoj poziciji. U tom slučaju koristimo tuple. Primer je dat listingom ispod.

```
let student: [string, string, boolean, number, number[]] =
['Pera', 'Peric', true, 2, [8,9,8]];

for(let podatak of student){
     console.log(podatak);
}
```

Listing - tuple

Deklarisali smo varijablu `student` kao tuple u kom se na prva dva mesta nalaze stringovi, na trećem mestu boolean, na četvrtom broj, a na petom mestu niz brojeva. Toj varijabli odmah smo i dodelili vrednost u skladu sa deklaracijom.

Enum

Kao i u programskom jeziku Java, enumeracija nam omogućuje da uvedemo novi tip nabrajanjem vrednosti koje mu pripadaju. Obratite pažnju da je to prvi mehanizam koji smo u ovoj lekciji videli kojim *proširujemo* skup tipova. Primer je dat listingom ispod.

```
enum Smer{ Racunarstvo, Masinstvo, Saobracaj };

let smerStudija: Smer = Smer.Masinstvo;
```

Listing - enumeracija

Kreirali smo enumeraciju `Smer` koja ima tri vrednosti. Zatim smo deklarisali varijablu `smerStudija` i dodelili joj vrednost `Smer.Masinstvo` iz enumeracije.

Any

Dinamička tipiziranost programskog jezika JavaScript se u pojedinim situacijama pokazala jako korisnom. I u TypeScript-u možemo da imamo varijable koje mogu da prime vrednosti različitih tipova. Za deklarisanje tih varijabli koristimo tip `any`. Primer je dat listingom ispod.

```
let a: any = 'tekst';

a = 55;
a = [1,2,3,'cetiri'];
```

Listing - tip `any`

Deklarisali smo varijablu `a` tipa `any` i dodelili joj vrednost tipa string, zatim broj i na kraju heterogeni niz.

Void

Tip void dodeljujemo varijablama koje mogu da prime vrednosti tipa undefined i vrednosti tipa null. Jedna situacija u kojoj bismo ga koristili je za deklarisanje povratne vrednosti funkcija koje *nemaju return*, na primer funkcija koja ispisuje poruku na ekran korisniku. Setimo se da takve funkcije implicitno vraćaju undefined.

```
function obavestenje(): void{

      alert('Vasa prijava za ispit je obradjena!');
}
```

Listing - tip `void`

Null i undefined

Naravno, moguće je specificirati da će varijabla biti baš tipa null ili baš tipa undefined. Primer je dat listingom ispod.

```
let u: undefined = undefined;

let n: null = null;
```

Never

U JavaScriptu postoje funkcije koje *nikada ne vraćaju vrednost*. Takve su funkcije koje uvek izazivaju izuzetak, ali i funkcije koje se nikada ne završavaju, kao što su beskonačni generatori (takav bi, na primer bio generator Fibonačijevih brojeva ili autoinkrementirajućih identifikator ili bilo koji ciklični generator). Za predstavljanje povratne vrednosti takvih funkcija koristi se tip never. Primer je dat listingom ispod.

```
function greska(poruka: string): never {

    throw new Error(poruka);
}
```

Listing - tip `never`

Listing prikazuje funkciju koja uvek izaziva izuzetak. Pošto nikada neće vratiti vrednost, čak ni implicitno vratiti undefined, povratna vrednost ove funkcije je tipa `never`.

Interfejsi

Kao i u „čistom“ JavaScript-u, i u TypeScriptu se pri proveri tipova proverava struktura objekata, odnosno podržan je duck typing. Interfejsi nam omogućuju da zadamo tip tako što ćemo propisati strukturu koju imaju objekti tog tipa. To znači da ćemo zadati koje atribute objekti mogu ili moraju da imaju i koje metode objekti mogu ili moraju da imaju. Primer interfejsa dat je listingom ispod.

```
interface Tacka {

    x: number;
    readonly y: number;
    zaPrikaz?: boolean;
    // //proizvoljan broj polja tipa string ili broj
    [naziv: string]: string | number | boolean | Function;
    prikazi(poruka: string):void;
}

let t1:Tacka = { x:100, y:200, z:400, u:'[1,2,3]',
prikazi:function (p:string) {
     console.log(p,this.x,this.y);
} };
```

Listing – interfejs

Navedeni primer zahteva detaljno objašnjenje. Napravili smo interfejs nazvan `Tacka`. U interfejsu navodimo nazive i tipove atributa koje objekat treba da ima. Naš interfejs ima atribute `x`, `y` i `zaPrikaz`. Modifikat `readonly` koji vidimo ispred atributa `y` znači da će vrednost tog atributa moći da se postavi jedino prilikom kreiranja objekta i da kasnije neće moći da se izmeni. Modifikator `?` koji vidimo iza atributa `zaPrikaz` znači da će ovaj atribut biti opcioni i da neće morati da se postavi za kreirane objekte. Nakon naziva atributa sledi dvotačka, pa tip. Vidimo da će `x` i `y` biti brojevi, a `zaPrikaz` biti boolean.

Možemo da zadamo i da objekat ima i proizvoljne atribute. Ključ atributa u tom slučaju zadaje se sa svojim tipom u uglastim zagradama. Nakon toga sledi tip vrednosti proizvoljnih atributa. Vidimo da smo u našem interfejsu omogućili da bude dodato proizvoljno mnogo atributa čiji ključ će biti string.

Ukoliko želimo da atribut može da primi vrednosti različitih tipova, nazive tipova razdvajamo `|`. U našem slučaju proizvoljni atributi mogu da budu ili string ili broj ili boolean ili funkcija.

U interfejs smo dodali i jednu metodu koja se zove `prikazi`. Ova metoda prima jedan parametar tipa `string` i, pošto očekujemo da će samo ispisati vrednost u konzolu i neće eksplicitno pozvati return, vraća `undefined` ili `null`.

Interfejs funkcije

Pored strukture objekta, interfejsom je moguće propisati i strukturu koju funkcija treba da zadovoljava. U tom slučaju propisuje se kog tipa će biti parametri funkcije i kog tipa će biti povratna vrednost funkcije. Primer je dat listingom ispod.

```
interface Filtriranje {

  (originalnaLista: string[], podstring: string): string[];
}

let f: Filtriranje = function (lista: string[], term: string) {
    return lista.filter(function (item) {
        return item.indexOf(term)!=-1;
    });
}

console.log(f(['foo', 'bar', 'baz'], 'ba'));
```

Listing - interfejs funkcije

Na listingu ispod vidimo da smo napravili interfejs `Filtriranje`. Funkcije napisane prema ovom interfejsu trebaju da prime dva parametra: prvi je lista stringova, a drugi je string. Pored toga, trebaju da vrate listu stringova. Napisali smo jednu funkciju (`f`) koja implementira navedeni interfejs i koja vraća sve elemente lista koji kao podstring imaju navedeni term.

Nasleđivanje interfejsa

Veza nasleđivanja interfejsa ostvaruje se pomoću rezervisane reči `extends`. Ako bismo hteli da naš interfejs za predstavljanje tačaka proširimo na trodimenzioni koordinatni sistem dodajući novu koordinatu `z`, to bismo uradili kao što je prikazano listingom ispod.

```
interface Tacka3D extends Tacka {

    z: number;
}
```

Listing - nasleđivanje interfejsa

Klase

Obzirom da je TypeScript nadskup ES6, i u TypeScript-u su podržane klase. Međutim, u odnosu na ES6 TypeScript uvodi i nove koncepte. Uvedeni su modifikatori pristupa: `private`, `public` i `protected` koji imaju isto značenje kao i u programskom jeziku Java. Takođe, atribut i metodu je moguće proglasiti statičkom modifikatorm `static`. Klase koje mogu da se naslede, ali ne mogu da se instanciraju su `abstract`.

Klase mogu i da implementiraju interfejse. Listingom ispod dat je primer implementacije interfejsa.

```
interface Tacka2D {

      x: number;
      y: number;
      prikazi(): void;
}

class Tacka implements Tacka2D{
      public x: number;
      public y: number;
      constructor(x: number, y: number){
            this.x = x;
            this.y = y;
      }
      public prikazi(){
            console.log('(',this.x,',',this.y,')');
      }
}

let t1: Tacka2D = new Tacka(1,2);
t1.prikazi();
```

Listing – klasa koja implementira interfejs

Na listingu vidimo da smo zadali interfejs `Tacka2D` koji ima dva numericka atributa, `x` i `y`, i jednu metodu koja ne prima parametre i vraća `null` ili `undefined`. Klasa `Tacka` implementira taj interfejs. Prilikom deklaracije varijable `t1` za njen tip smo zadali da je interfejs `Tacka2D` i dodelili joj instancu klase `Tacka`.

Dekoratori

Dekoratori se mogu postaviti na klase, atribute, metode i parametre metoda i oni omogućuju anotiranje i meta-programiranje. U vreme pisanja ovih materijala dekoratori su još uvek dostupni jedino kao eksperimentalno svojstvo TypeScript-a[1]. Meta-programiranje predstavlja naprednu tehniku programiranja i izlazi iz opsega ovih materijala, ali obzirom na ulogu koju dekorateri igraju u radnom okviru Angular biće ilustrovano i objašnjeno njihovo korišćenje.

Dekoratori se zadaju u obliku gde `@izraz` se `izraz` evaluira u funkciju koja se poziva vremenu izvršavanja programa. Ta funkcija može da pristupi dekorisanom objektu i da ga, po potrebi, izmeni.

Pogledajmo primer dekoratora dat listingom ispod.

[1] Kompajliranje .ts fajova u verziji jezika u kojoj su podržani dekoratori ostvaruje se pokretanjem typescript paketa uz navođenje: `tsc --target ES5 --experimentalDecorators`

```
function dodajPozdrav(target: any) {

  target.prototype.pozdrav = function () {
    console.log('Zdravo');
  }
}

@dodajPozdrav
class Osoba{

  public ime: string;
  public prezime: string;

  constructor(ime : string, prezime : string) {
    this.ime = ime;
    this.prezime = prezime;
  }

}

let pera = new Osoba('Pera','Peric');
pera.pozdrav();
```

Listing – dekorator

Napisali smo funkciju `dodajPozdrav` koja prima jedan parametar i u prototip prosleđenog parametra dodaje funkciju pozdrav. Dekoratorom smo dekorisali klasu `Osoba`. To znači da će dekorator biti pozvan prilikom kreiranja *klase* Osoba i da će pri tom pozvu parametar target dobiti vrednost klase koju smo prosledili (`Osoba`). Setimo se da je klasa u JavaScript-u u stvari konstruktorska funkcija, tako da dodavanje nove funkcije u prototip klase rezultuje time da će svaki objekat kreiran instanciranjem te klase u stvari imati dodatu metodu. Time smo dobili da dekorisanjem klase `Osoba` dekoratorom `dodajPozdrav` svi objekti klase `Osoba` imaju i pristup metodi `pozdrav`. To i vidimo u poslednjoj liniji koda u kojoj pozivamo metodu `pozdrav` objekta `pera`.