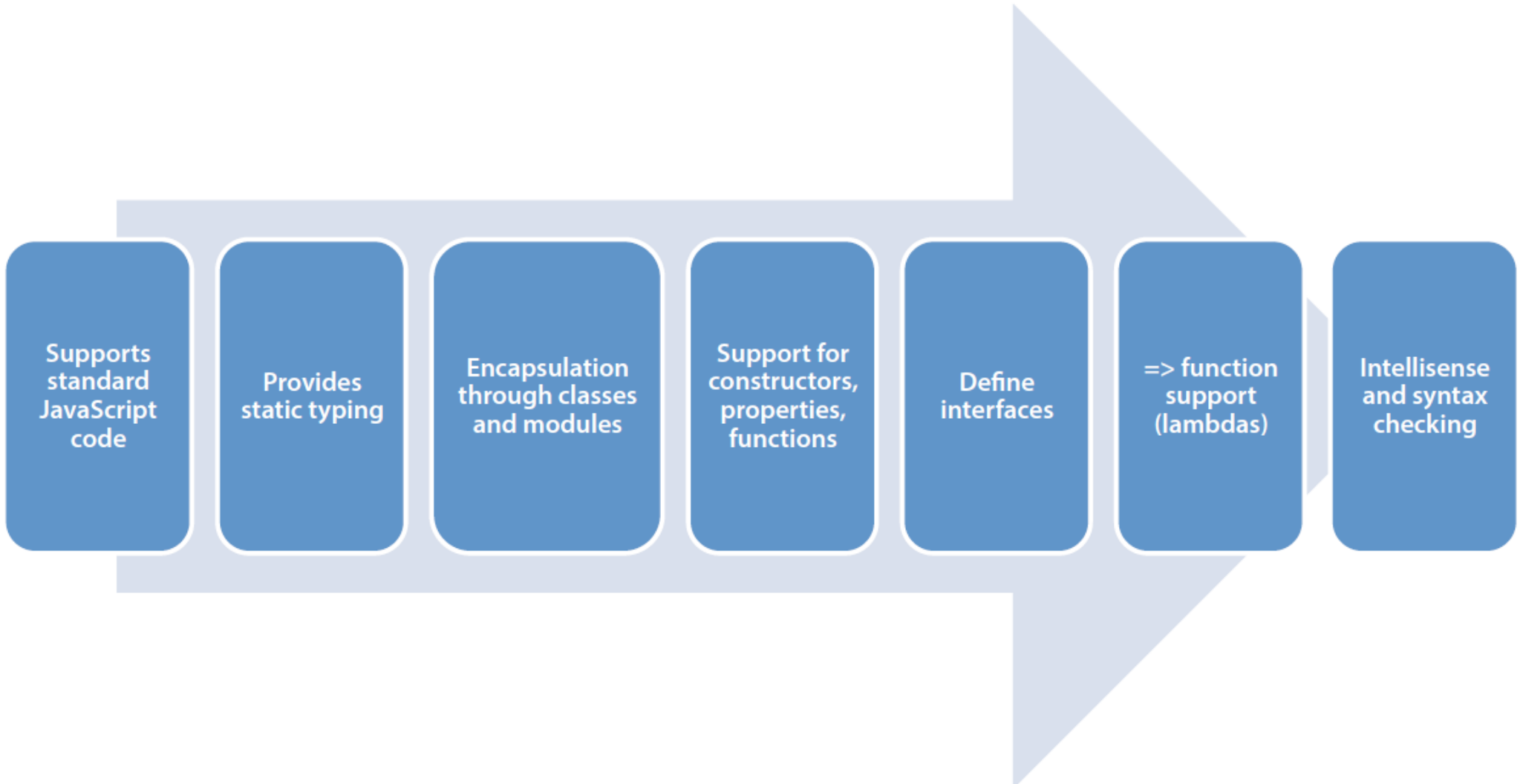
TypeScript

TypeScript

- http://www.typescriptlang.org/
- Problemi sa JavaScriptom:
  - Dynamic typing
  - Manjak mehanizama sa struktuiranje koda
  - Function Spaghetti Code

Key TypeScript Features



TypeScript - Instalacija

- http://www.typescriptlang.org/#download-links
- TypeScript je najlakse instalirati kao Node.js paket
- Instalirati Node.js https://nodejs.org/
- U command prompt ukucajte komandu:
  - `npm install -g typescript`
- Kako bi proverili da li je TypeScript instaliran, u command promptu ukucajte komandu za ispis verzije kompajlera:
  - `tsc -v`

```
PS C:\> tsc -v
Version 2.7.1
```

Tipovi podataka

http://www.typescriptlang.org/docs/handbook/basic-types.html

- Boolean
- Number
- String
- Array
- Tuple
- Enum
- Any
- Void
- Null
- Undefined
- Never
- …
- (HTMLElement… DOM tipovi)

Type Annotations

- Tipovi se “dodeljuju” promenljivima tako sto se nakon deklaracije promenljive stavi karakter ‘:’ i ime tipa:

```
let num: number;
let x: string= 'I will forever be a string.';
```

- Pokušaj dodele vrednosti drugog tipa rezultuje greškom:

```
x = 45;
Type '45' is not assignable to type 'string'.
let x: string
```

Type Annotations

- Nizovi je moguće definisati na dva načina:

```
let names: string[] = ["Jack", "John", "Jill", "James"];

let surnames: Array<string> = ["Smith", "Jones", "Burns"];
```

- Ukoliko ne anotiramo tip, TypeScript će pokušati da ga automatski odredi (Type Inference):

```
let ime = "Pera";
    let ime: string
```

Union Types

- Moguće je definisati unije dva ili više tipova.
- Unija omogućava da se u jednu promenljivu mogu smestiti različiti tipovi podataka (tipovi navedeni u uniji):

```
let someValue: number | string;

someValue = 42; //Ispravna dodela

someValue = 'Hello World'; //Ispravna dodela

someValue = true; //Greska
```

Type Assertions

- U određenim situacijama developeru će biti poznat tip promenljive, dok TypeScript kompajleru neće.
- U tim situacijama možemo kompajleru eksplicitno reći koja vrednost se nalazi u promenljivoj (Type Assertion).
- Sintaksa:

```
<tip> promenljiva
        ili
promenljiva as tip
```

- Dobra praksa je kada se Type Assertion koristi u izrazu je da bude okruzen zagradama:

```
let n: number = (<number> br) * 5;
let n: number = (br as number) * 5;
```

Type Assertions

- Type Assertion primer:
  - Problem: želimo da ispišemo brojčanu vrednost neke promenljive sa četiri decimale. Pri čemu vrednost te promenljive dobijamo preko globalne promenljive koja ne pripada ts (TypeScript) fajlu.

```
let value: any = 5;

let firstString: string = (<number>value).toFixed(4);
console.log(firstString); // 5.0000

let secondString: string = (value as number).toFixed(4);
```

Type Assertions

- Type Assertion primer:

```
let value: any = 5;

let firstString: string = (<number>value).toFixed(4);
console.log(firstString); // 5.0000

let secondString: string = (value as number).toFixed(4);
```

  - Promenljiva **`value`** u sebi sadrzi vrednost tipa **`number`**, to TS kompajleru nije poznato (TS je tretirao kao tip **`any`**) ali programeru jeste.
  - Ispis sa četiri decimale moguć je upotrebom metode **`.toFixed(4)`** koja se može pozvati samo nad promenljivama tipa **`number`**.
  - Rešenje: Eksplicitno reći TSu kog tipa je promenljiva pomoću Type Assertion-a i pozvati odgovarajuću metodu nad tim tipom.

Funkcije

```
function dullFunc(value1, value2) {
    return "I'm boring and difficult. Don't be like me.";
}

function funFunc(score: number, message?: string): string {
    return "I've got personality and I'm helpful! Be like me!";
}
```

- Opcioni parametri imaju znak ‘?’ nakon imena parametra.
- Opcioni parametri se obavezno navodne nakon svih obaveznih parametara.

Funkcije

- Podrazumevana (Defualtna) vrednost parametara:

```
function sendGreeting(greeting: string = 'Good morning!'): void {
    console.log(greeting);
}

sendGreeting(); //Good morning!
sendGreeting('Good afternoon!'); //Good afternoon
```

Funkcije

- Funkcije imaju svoje tipove.
- Tu osobinu možemo iskoristiti kada određujemo tip promenljive u koje smeštamo funkcije, npr. kod korišćenja callback funkcija.
- Tip funkcije se sastoji od dva dela:
  - Tipovi parametara
  - Tip povratne vrednosti
- Npr.:
  - Funkcija:

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

  - je tipa:

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

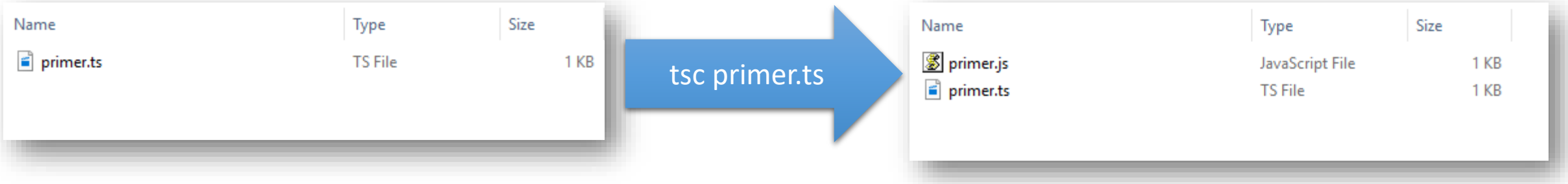
Funkcije

- Tip funkcije: imena parametara ne moraju biti identična, bitan je redosled i tip parametara:

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

let pomnozi = function(x: number, y: number): number { return x * y; };

let funk: (p1: number, p2:number) => number;

funk = add;
console.log(funk(5,10)); // 15

funk = pomnozi;
console.log(funk(5,10)); // 50
```

Kompajliranje

- Da bi preveli TypeScript kod u JavaScript kod potrebno je pokrenuti instalirani TypeScript kompajler
- Da bi to odradili trebamo da pozicioniramo terminal (ili command prompt) u folderu u kome se nalazi nas projekat (nasi TypeScript fajlovi)
- Kompajliranje se pokreće kucanjem komande: `tsc imeFajla.ts`
- U istom folderu dobijamo odgovarajući .js fajl



Kompajliranje

```
function foo(...x: number[]) : void {
    console.log(x);
}

foo(1,2,3,4,5,6,7,8);
```

```
function foo() {
    var x = [];
    for (var _i = 0; _i < arguments.length; _i++) {
        x[_i] = arguments[_i];
    }
    console.log(x);
}
foo(1, 2, 3, 4, 5, 6, 7, 8);
```

Domaći: pronaći značenje za … kod parametra x!

tsconfig.json

- http://www.typescriptlang.org/docs/handbook/tsconfig-json.html
- Kompajliranje većeg broja .ts fajlova može se optimizovati podešavanjem opcija TypeScript kompajlera
- Podešavanje opcija kompajlera je moguće odraditi upotrebom *tsconfig.json* fajla
- Prisustvo *tsconfig.json* fajla u folderu naznačava da je taj folder root folder TypeScript projekta
- Moguće je vršiti nasleđivanje *tsconfig.json* fajlova
- Pokretanje kompajliranja se vrši:
  - Pozivom komande `tsc` u folderu u kome se nalazi tsconfig.json (kao u prošlom primeru ali bez naziva typescript fajla koji kompajliramo)
  - Pozivom komande `tsc` sa dodavanjem opcije `-p` sa kojom navodimo putanju do *tsconfig.json fajla*

tsconfig.json

- Moguće je napraviti inicijalni tsconfig.json fajl pozivom komande `tsc` sa opcijom `–init:`

```
C:\primeri>tsc --init
message TS6071: Successfully created a tsconfig.json file.
```

tsconfig.json

```
{
    "compilerOptions": {
        "target": "es5",
        "watch": true,
        "sourceMap": true,
        "outDir": "js"
    },
    "files": [
        "ts/primer.ts"
    ]

}
```

- **`compilerOptions`** – objekat u kojem navodimo opcija za kompajlrianje
  - `target` – na koju verziju JS prevodimo nas kod
  - `watch` – omogućava konstantan rad kompajlera, posmatra naše .ts fajlove i vrši njihovo rekompajliranje ako dođe do promene
    - ukoliko je ova opcija uključena; nakon završetka rada na zadatku isključiti kompajler sa komandom Ctrl+C u command promptu (terminalu)

tsconfig.json

```
{
    "compilerOptions": {
        "target": "es5",
        "watch": true,
        "sourceMap": true,
        "outDir": "js"
    },
    "files": [
        "ts/primer.ts"
    ]

}
```

- `sourceMap` – omogućava debagovanje u browseru tako što generiše .js.map fajlove
- `outDir` – putanja do foldera u koji želimo da smestimo prekompajlirane fajlove (svaki .ts fajl će imati odgovarajući .js fajl u navedenom folderu)
- `outFile` - * može da zameni outDir tako da se svi .ts fajlovi kompajliraju u jedan veliki .js fajl
- **`files`** – niz koji sadrži putanje do fajlova koje želimo da kompajliramo

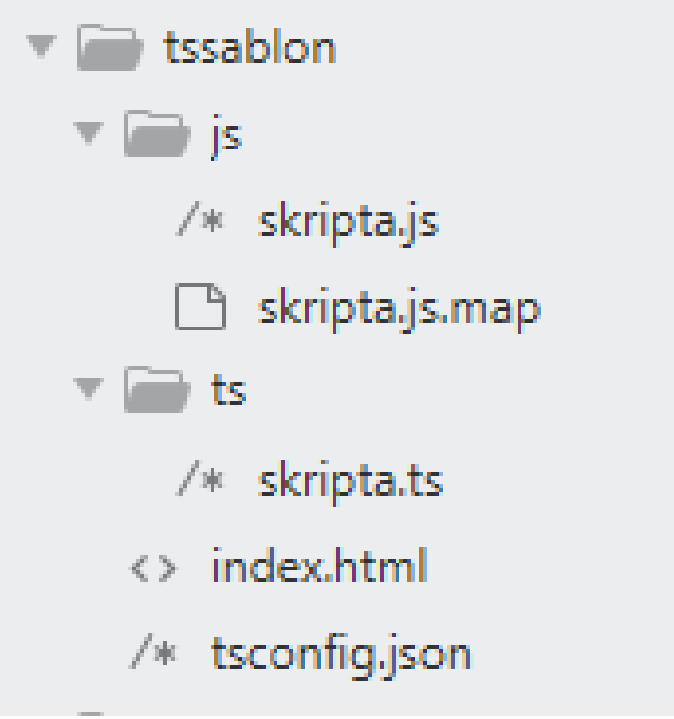
tsconfig.json

- `files` je moguće zameniti sa includes i excludes nizovima
- `includues` – prima glob šablone koje predstavljaju putanje do fajlova koje treba kompajlirati
- `excludes` – prima glob šablone koje predstavljaju putanje do fajlova koje ne treba kompajlirati

```
{
    "compilerOptions": {
        "target": "es5",
        "watch": true,
        "sourceMap": true,
        "outDir": "js"
    },

    "include": [
        "./**/*"
    ],

    "exclude": [
        "node_modules",
        "**/*.spec.ts"
    ]

}
```

tsconfig.json

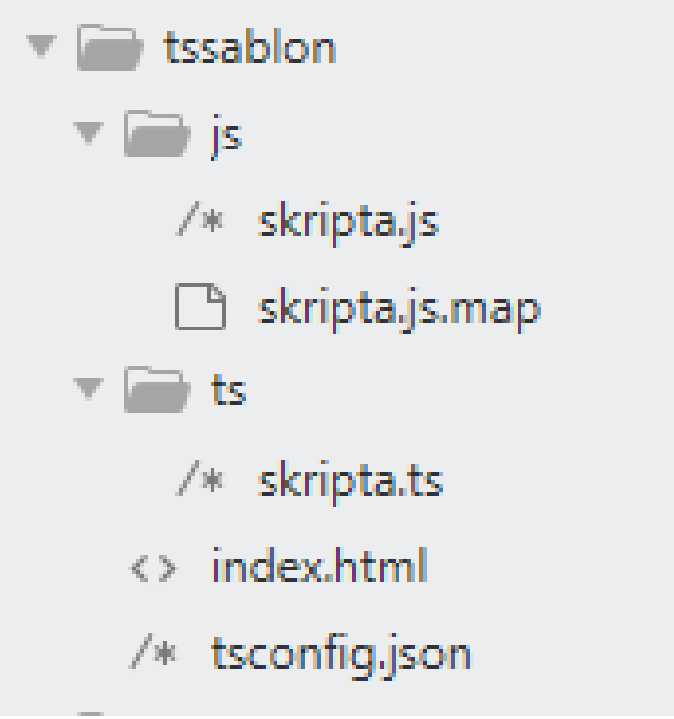
- Struktura TypeScrpit projekata na časovima (tssablon.zip):



- korenski (root) folder
  - **js folder** (dobija se nakon pokretanja kompajlera)
    - u njega TypeScript smešta rezultat kompajliranja
  - **ts folder**
    - u njega ili njegove podfoldere smestamo .ts fajlove
  - **html stranice** - <script> tagovi u html stranicama treba da učitavaju skripte iz **js foldera**
  - **tsconfig.json**

tsconfig.json

- Struktura TypeScrpit projekata na časovima (tssablon.zip):

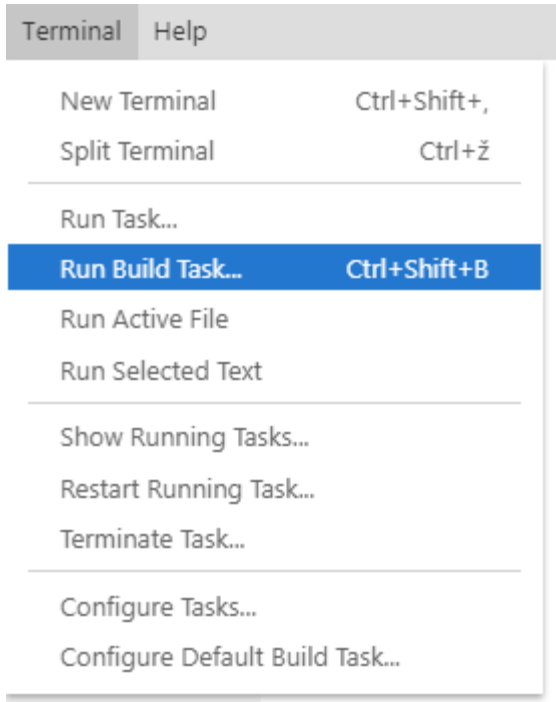


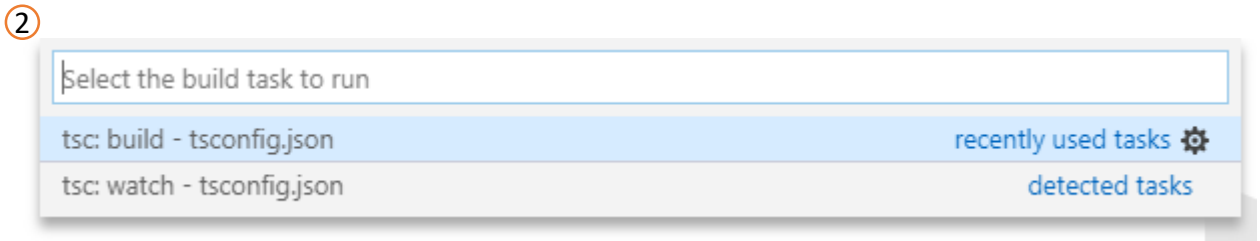
- tsconfig.json:

```
{
    "compilerOptions": {
        "target": "es5",
        "watch": true,
        "sourceMap": true,
        "outDir": "js"
    },
    "include": [
        "./**/*"
    ]
}
```

Kompajliranje – Visual Studio Code

- Pokretanje kompajliranje u Visual Studio Code-u se može odraditi na klasičan način (slajd 18) iz ugrađenog terminala (Ctrl+`) ili da odete na opciju View > Terminal
- Drugi način je pokretanjem (1) Run Build Task opcije (skraćenica Ctrl+Shift+ B) i izborom (2) tsc: build – tsconfig.json





Objektno-orijentisano programiranje - OOP

- Objektno-orijentisani program predstavlja kolekciju slabo povezanih objekata koji međusobno interaguju
- Svaki objekat ima svoj skup zadataka
- Objekti su opisani svojim stanjem i ponašanjem:
  - Stanje – vrednosti atributa (svojstava)
  - Ponašanje – metode (funkcije) objekata
- Modelovanje na osnovu potrebne funkcionalnosti
- OOP pristup omogućava:
  - Modularnost
  - Sakrivanje implementacionih detalja (Information-hiding)
  - Code re-use

OOP - Klasa

- Šablon (recept, blue print) koji definiše objekat
  - Klasa je apstraktan pojam koji opisuje neku grupu objekata. Klasa može da bude **stolica**, **tabla**, ili **čovek**.
  - Objekat predstavlja konkretnu instancu neke klase. Npr. učionica je puna objekata čija klasa je stolica. To su, dakle, konkretne stolice koje se mogu locirati u prostoru (odnosno u memoriji, kada pričamo o računarskom programu).
- Može se posmatrati i kao tip podatka koji je definisao korisnik
- Klasa sadrži:
  - Naziv
  - Atribute (polja, podatke članove)
  - Mehanizme za stvaranje objekata na osnovu definicije (konstruktore)
  - Metode (operacije, funkcije)

OOP - Klasa

- Problem modelovanja:
- Kako izmodelovati radnika ili automobil?

TypeScript Classes

- ES5 i ranijie verzije JavaScripta korisiti funkcije i prototipsko nasledjivanje za implementaciju klasa.
- ES6 standard uvodi mogućnost korišćenja klasa.
- TypeScript nam omogućava upotrebu klasa u našem kodu, i po potrebi njihovo prevođenje (kompajliranje) na ranije verzije JavaScripta

TypeScript Classes

```
class Animal {
    name: string; //Polje - property
    constructor(theName: string) { //Konstruktor
        this.name = theName;
    }
    move(distanceInMeters: number = 0) { //Metoda
        console.log(`${this.name} moved ${distanceInMeters}m.`);
    }
}
```

Kontrola pristupa (access control)

- Postoje tri modifikatora pristupa:
  1. private – atributi i metode su vidljivi samo unutar klase
  2. protected – atributi i metode su vidljivi i za klase naslednice
  3. public – atributi i metode su vidljivi za sve klase u programu
- Modifikatori pristupa se navode ispred definicija metoda i atributa
- Ukoliko eksplicitno ne navedmo modifikator pristupa, TypeScript će smatrati da su oni public
- Ukoliko imamo private atribute u TypeScript klasama, preporuka je da ime tog atributa počne sa donjom crtom „_“ (npr. _ime)

Kontrola pristupa (access control)

- Private:

```
class Animal {
    private name: string;
    constructor(theName: string) { this.name = theName; }
}

new Animal("Cat").name; // Error: 'name' is private;
```

Geteri i Seteri (Accessors)

- U OOP atributi u klasi se obično proglašavaju privatnim (loša praksa je da se ostavljaju da budu public)
- Da bi se moglo pristupati tim atributima potrebno je kreirati metode get i set za svaki pojedinačni atribut
- Na taj način se može kontrolisati mogućnost pisanja (samo set metoda) i mogućnost čitanja vrednosti atributa (samo get metoda)
- U TypeScriptu za kreiranje getera i setera koriste se ključne reči **get** i **set** pre imena metode
  - Za razliku od ostalih metoda geteri i seteri u typscript-u se koriste kao public atributi
  - Ne pozivaju se kao metode sa zagradama (), već im se pristupa isključivo sa dot (.) notacijom
  - U TypeScriptu get i set metode trebalo bi da imaju isti naziv (bez donje crtice) kao i private atribut kome pristupaju

Geteri i Seteri (Accessors)

```
let passcode = "secret passcode";

class Radnik {
    private _fullName: string;

    get fullName(): string {
        return this._fullName;
    }

    set fullName(newName: string) {
        if (passcode && passcode == "secret passcode") {
            this._fullName = newName;
        }
        else {
            console.log("Error: Unauthorized update of employee!");
        }
    }
}

let employee = new Radnik();
employee.fullName = "Pera Peric";
if (employee.fullName) {
    console.log(employee.fullName);
}
```

Seter, poziva se set metoda fullName

Getter, poziva se get metoda fullName

Statička polja

- TypeScript omogućava upotrebu i statičkih atributa/metoda
- Statičko polje je polje koje je vezano za klasu a ne za objekat (instancu) klase, tj. statičko polje je zajedničko za sve instance te klase
- Statička polja se deklarišu upotrebom ključne reči **`static`**
- Statičkim poljima u kodu se pristupa navođenjem imena klase i imena statičkog polja:

  `ImeKlase.imeAtributa` ili **`ImeKlase.imeMetode(...)`**

Statička polja

- Primer upotrebe:

```
let a1: Automobil = new Automobil("111", "1111");
let a2: Automobil = new Automobil("222", "2222");
let a3: Automobil = new Automobil("333", "3333");

console.log(Automobil.brojProizvedenih); //3
```

```
//Primer statickog polja, voditi evidenciju ukupnog
//broja instanciranih objekta

class Automobil {
    static brojProizvedenih: number = 0;

    private _boja: string;
    private _tablice: string;
    private _brzina: number;

    constructor(boja: string, tablice: string){
        this._boja = boja;
        this._tablice = tablice;
        this._brzina = 0;
        Automobil.brojProizvedenih += 1;
    }

    public ubrzaj(delta: number): void {
        this._brzina += delta;
    }

    public uspori(delta: number): void {
        this._brzina -= delta;
    }

    get brzina(): number {
        return this._brzina;
    }

}
```