TYPED IS THE WAY

HI, MY NAME IS DIMI.

Developer
Ordina Belgium
https://github.com/DimiDeKerf

JAVASCRIPT


THE NOT-SO-GOOD PARTS
JavaScript is weakly typed
Type coercion
Callback hell
Refactoring
JavaScript
The Definitive Guide
David Flanagan
O'REILLY
Unearthing the Excellence in JavaScript
JavaScript
The Good Par
O'REILLY
YAHOO! PRESS

WEAK TYPING

```
var myNumber = 2;
myNumber = '4';
myNumber = true;
```

TYPE COERCION

When operands have a different type, JS tries to convert one of them so they are equal.

```
var myNumber = 2;
var myNotANumber = '4';
console.log(myNumber + myNotANumber); // Will print '24'
```

CALLBACK HELL

Dealing with a callback from a callback within a callback...

```
function hell(win) {
 // for listener purpose
 return function() {
   loadLink(win, REMOTE_SRC+'/assets/css/style.css', function() {
     loadLink(win, REMOTE_SRC+'/lib/async.js', function() {
       loadLink(win, REMOTE_SRC+'/lib/easyXDM.js', function() {
         loadLink(win, REMOTE_SRC+'/lib/json2.js', function() {
           loadLink(win, REMOTE_SRC+'/lib/underscode.min.js', function() {
             loadLink(win, REMOTE_SRC+'/lib/backbone.min.js', function() {
               loadLink(win, REMOTE_SRC+'/dev/base_dev.js', function() {
                 loadLink(win, REMOTE_SRC+'/assets/js/deps.js', function() {
                   loadLink(win, REMOTE_SRC+'/src/' + win.loader_path + '/loader.js', function() {
                     async.eachSeries(SCRIPTS, function(src, callback) {
                       loadScript(win, BASE_URL+src, callback);
                     });
                   });
                 });
               });
             });
           });
         });
       });
     });
   });
 };
}
```

REFACTORING

Difficult to refactor JS code, not suitable for large scale apps.

ECMASCRIPT 6

Solves a lot of issues and brings tons of new features, but still lacks typings and support by browsers...

HELP ME, TYPESCRIPT.
YOU'RE MY ONLY HOPE.

TYPESCRIPT

TYPESCRIPT

Superset of JavaScript

Developed by Microsoft

Open-source, object-oriented language

Strongly typed

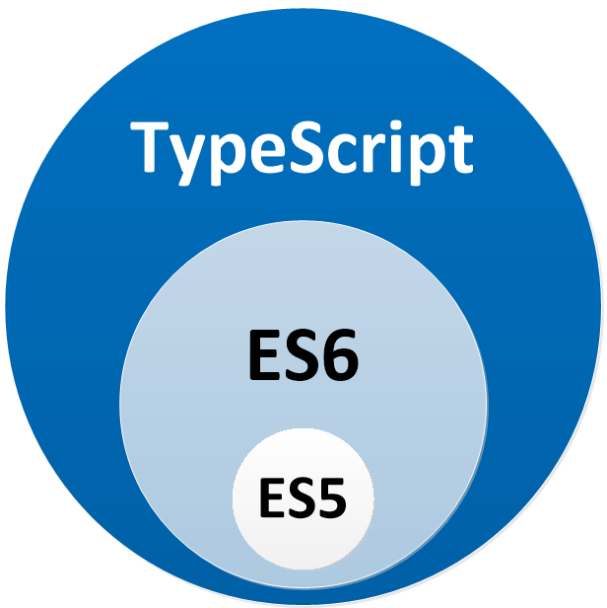
Transpiles to JavaScript

SUPERSET OF JAVASCRIPT

TypeScript is JavaScript

Shares same syntax and semantics

Allows JavaScript to be used at scale



STRONGLY TYPED

Define types for variables

Code is more readable

Refactor existing code with ease

Detect errors at compile time

Increase code quality

Optional

TRANSPILES TO JAVASCRIPT

TypeScript cannot be understood by JS engine

Transpiled output of TypeScript is plain JavaScript

Runs on any modern browser and JS engine like Node

TYPED SCRIPT

USING TYPES

Define the type after a colon.

```
const name: type;
```

```
const nameOfPerson: string = 'Luke';

function getName(): string {
    return nameOfPerson;
}
```

BOOLEAN

Most simple type, `true` or `false`.

```
const toggle: boolean = true;
```

NUMBER

Define numbers in decimal, hexadecimal, binary,... format.

```
const decimal: number = 10;
const hexaDecimal: number = 0x010;
const binary: number = 0b0101;
const octal: number = 0o010;
```

STRING

For textual data, use the string type.

```
const name: string = 'Luke';
```

TEMPLATE STRING

Useful for string interpolation.

```
const name: string = 'Luke';

const concatenatedString: string = 'The Force is with you, young ' + name
    + ', but you are not a Jedi yet';

const templateString: string = `The Force is with you, young ${name},
    but you are not a Jedi yet`;
```

ARRAY

Define a list of items.

```
const heroes: string[] = ['Luke', 'Leia', 'Han'];

const villains: Array<string> = ['Darth Vader', 'The Emperor', 'Boba Fett'];
```

ENUM

Name a numeric value for more readable code.

```
enum Rank {
    Padawan,
    Knight,
    Master
};

const myRank: Rank = Rank.Padawan;
```

ENUM

You can also assign a `string` to an Enum.

```
enum Rank {
    Padawan = 'padawan',
    Knight = 'knight',
    Master = 'master'
};

const myRank: Rank = Rank.Padawan;
console.log(myRank); // Logs "padawan"
```

TUPLE

Declare the types of a fixed array, useful to create pairs.

```
let nameAgePair: [string, number];
nameAgePair = ['Luke', 19];
nameAgePair = [19, 'Luke']; // Won't compile
nameAgePair = ['Luke', 19, 'Tatooine']; // Won't compile either
```

NULL AND UNDEFINED

Subtype of all other types. Not useful on their own though...

```
const u: undefined = undefined;

const n: null = null;

let myName: string = 'Luke';
myName = null;
```

ANY

Opt-out of type-checking, can have any value.

```
let myName: any = 'Luke';
myName = 2;
myName = true;
```

VOID

Opposite of any, no type at all.

```
const unused: void = null;

function sleep(): void {
    console.log('Zzzz...');
}
```

TYPE INFERENCE

Compiler infers type of variables, so you don't have to explicitly set the type yourself.

```
const myName: string = 'Luke';

const inferredName = 'Luke'; // Inferred as string
```

VARIABLE DECLARATION

LET

Use `let` to declare variables, without the negative effects of `var`.

```
let myName: string = 'Luke';
myName = 'Han';
```

CONST

With `const`, the value of the variable cannot be changed once it has been set.

```
const myName: string = 'Luke';
myName = 'Han'; // Won't compile
```

DESTRUCTURING

Destruct an object to unpack its values in distinct variables.

```
const jedi = {
    name: 'Luke',
    age: 19
};
const {name, age} = jedi;
console.log(`My name is ${name} and I am ${age} years old.`);
```

SPREAD

Spread an array or object into another array/object.

```
const oddNumbers = [1, 3, 5];
const evenNumbers = [2, 4];

const numbers = [...oddNumbers, ...evenNumbers]; // 1, 3, 5, 2, 4

const padawan = {
    name: 'Luke',
    age: 19
    rank: Rank.Padawan
};

const knight = {
    ...padawan,
    rank: Rank.Knight // Overwrite the rank, other properties will remain the same
};
```

SETUP DEV ENVIRONMENT

PREREQUISITES

Node & npm

Editor of choice

INSTALL TYPESCRIPT USING NPM

Run the following command

```
npm install -g typescript
```

Check if the TypeScript compiler is working

```
tsc -v
```

PROJECT CONFIGURATION

COMPILER CONFIGURATION

Running `tsc` will trigger the compiler with default settings.

To set a custom configuration, you can use compiler flags

```
tsc --target ES5 file.ts
```

TSCONFIG.JSON

Specifies compiler options, configures how the compiler will compile your project.

Mostly located in the root of your project.

TSCONFIG.JSON

```
{
    "compilerOptions": {
        "module": "amd",
        "noImplicitAny": true,
        "removeComments": true,
        "preserveConstEnums": true,
        "sourceMap": true
    }
}
```

TSLINT

Static code analysis tool for TypeScript. Helps you to write more readable, maintainable code.

Supported by most editors.

DEPRECATED

TSLint is deprecated in favour of ESLint.

ESLINT SETUP

Configure ESLint using a `.eslintrc.js` file.

ESLINT SETUP

Configuration depends on preferences, align with your team and organisation.

EXERCISE 1: BASIC PROJECT

Get familiar with the basic TypeScript syntax and project setup.

FUNCTIONS

FUNCTIONS

Fundamental building block of JavaScript applications. Define how to perform a specific task.

DEFINE A FUNCTION

NAMED FUNCTION

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

ANONYMOUS FUNCTION

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

ARROW FUNCTION

Alternative way to declare anonymous functions. Also known as lambda functions in other languages.

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

// With fat arrow
const addArrow = (x: number, y: number): number => {
    return x + y;
};
```

ARROW FUNCTION

If the body of the function has only one statement, you can omit the curly brackets and `return` keyword.

```
const addArrow = (x: number, y: number): number => x + y;

// Same as

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

OPTIONAL PARAMS

Mark parameters optional with a ?

```
function square(x: number, unit?: string): string {
    const square = x * x;
    if (unit != null) {
        return `${square} ${unit}`;
    } else {
        return `${square}`;
    }
}

console.log(square(2, 'm2')); // Prints 4 m2
console.log(square(2)); // Prints 4
```

DEFAULT-INITIALIZED PARAMS

Gives the parameter a default value when calling the function, if none is given.

```
function square(x: number, unit = 'm2'): string {
    const square = x * x;
    return `${square} ${unit}`;
}

console.log(square(2)); // Prints 4 m2
console.log(square(2, 'f2')); // Prints 4 f2
```

REST PARAMS

When you have no idea how many parameters a function will need, you can gather all parameters in Rest params.

```
function add(...numbers: number): number {
    const sum = 0;
    for (let number of numbers) {
        sum += number;
    }
    return sum;
}

console.log(add(2, 1, 4)); // Prints out 7
```

CLASSES

CLASSES

Object-oriented approach to write your application.

CLASS

```
class Jedi {
    name: string;
    amountOfLightsabers: number;

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

    wield() {
        console.log(`${this.name} wields ${this.amountOfLightsabers} lightsaber(s)`);
    }
}
```

ACCESS MODIFIERS

Defines which properties and methods will be available to other classes.

- `public`
- `protected`
- `private`

PUBLIC BY DEFAULT

All members are `public` accessible by default. You can still add the `public` modifier.

```
class LightsaberWielder {
    public name: string;
    amountOfLightsabers: number; // Also public
}
```

PROTECTED

Marks members as `protected` so they can't be accessed from outside the class, unless for subclasses.

```
class LightsaberWielder {
    public name: string;
    protected amountOfLightsabers: number;

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

class Jedi extends LightsaberWielder {
    wield() {
        console.log('${name} is wielding ${amountOfLightsabers} saber(s).');
    }
}

const obiWan = new Jedi('Obi-Wan', 1);
obiWan.wield(); // Prints "Obi-wan is wielding 1 saber(s)."
console.log(obiWan.amountOfLightsabers); // Compile error: 'amountOfLightsabers' is p
```

PRIVATE

Marks members as `private` so they can't be accessed from outside the class.

```
class LightsaberWielder {
    public name: string;
    private amountOfLightsabers: number;

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

const obiWan = new LightsaberWielder('Obi-Wan', 1);
console.log(obiWan.amountOfLightsabers); // Compile error:
                                         // 'amountOfLightsabers' is private
```

#PRIVATE

ECMAScript's private fields that will remain private in transpiled JS

Mark the fields as private with #

```
class LightsaberWielder {
    name: string;
    #amountOfLightsabers: number;

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

const obiWan = new LightsaberWielder('Obi-Wan', 1);
console.log(obiWan.amountOfLightsabers); // Compile error:
                                         // 'amountOfLightsabers' is private
```

WHICH ONE TO USE?

It depends... If you don't want to expose your properties/functions from your API at all, go for # privates

ACCESSORS

Have more control of how a member of an object is accessed by using the `get` and `set` accessors.

```
class Jedi {
    private _name?: string;

    get name(): string {
        if (this._name) {
            return this._name;
        } else {
            return 'Unknown';
        }
    }

    set name(newName: string) {
        if (newName) {
            // Uppercase first letter
            this._name = newName.charAt(0).toUpperCase() + newName.slice(1);
        } else {
            this._name = newName;
        }
    }
}
```

INHERITANCE

Derive a class from another class with the `extends` keyword.

```
class LightsaberWielder {
    name: string;
    amountOfLightsabers: number;

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

    wield() {
        console.log(`${this.name} wields ${this.amountOfLightsabers} lightsaber(s)`);
    };
}

class Jedi extends LightsaberWielder {
    swing() { };
}

class Sith extends LightsaberWielder {
    stab() { };
}
```

ABSTRACT CLASS

Base class that other classes can extend. An abstract class can't be instantiated by itself.

```
abstract class LightsaberWielder {
    name: string;
    amountOfLightsabers: number
}

class Jedi extends LightsaberWielder {
    // Implementation omitted...
}

const obiWan: LightsaberWielder; // Declaring an abstract class is just fine
const obiWan = new LightsaberWielder(); // Compile error:
                                // create an instance of an abstract class
const obiWan = new Jedi();
```

READONLY

The property will be read only with the `readonly` keyword.
Readonly properties must be initialized.

```
class Jedi {
    readonly name: string;
    amountOfLightsabers: number;

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

const obiWan = new Jedi('Obi-Wan', 1);
obiWan.amountOfLightsabers = 2; // All good
obiWan.name = 'Obi-Wan Kenobi'; // Compile error: name is readonly
```

CONSTRUCTOR ASSIGNMENT

Concise way to create and assign properties. Make sure to explicitly add the accessor in the constructor.

```
class Jedi {
    constructor(public name: string) { }
}

// Same as

class Jedi {
    public name: string

    constructor(name: string) {
        this.name = name;
    }
}
```

STATIC PROPERTIES

Static members are visible on the class itself and will be shared across all instances of that class.

```
class Jedi {
    static amountOfLightsabers = 1;
}

const luke = new Jedi();
const obiWan = new Jedi();
luke.amountOfLightsabers = 2;
console.log(obiWan.amountOfLightsabers); // Will print 2
```

EXERCISE 2 & 3

Fix the failing unit tests in a TypeScript way.

INTERFACES

INTERFACE

Defines a contract in your application. Classes that implement an interface must follow the structure of that interface.

INTERFACE

```
interface DeathStarPlan {
    radius: number;
    fireWhenReady: (boolean) => string;
}

class DeathStar implements DeathStarPlan {
    radius: number;

    fireWhenReady(ready: boolean): string {
        if (ready) {
            return 'Fire!';
        } else {
            return 'Powering up primary weapon...';
        }
    }
}
```

MODEL INTERFACE

Use an interface to represent a model. Preferable way to create read-only models because no JavaScript code will be generated.

```
interface Jedi {
    name: string
}

function greet(jedi: Jedi) {
    console.log(`Hello there, ${jedi.name}`)
}

const jedi: Jedi = { name: 'Luke' };
greet(jedi);
```

OPTIONAL PROPERTIES

Mark properties optional inside an interface. Useful for config params.

```
interface DeathStarPlan {
    radius?: number;
    color?: string;
}

function assembleDeathStar(plan: DeathStarPlan) {
    const deathStar = {};
    if (plan.radius) {
        deathStar.area = plan.radius * plan.radius * Math.PI;
    } else {
        deathStart.area = 0;
    }
    deathStart.color = plan.color;
    return deathStar;
}

const deathStar = assembleDeathStar({ color: 'red' });
```

EXERCISE 4

Complete the Walker app.

OPTIONAL CHAINING

UNDEFINED OBJECTS

Calling methods or properties of undefined objects leads to undesired results.

```
const demolishedAlderaan = deathStar.superLaser.fire();
// What if deathStar or superLaser is undefined?
```

OPTIONAL CHAINING

Write code that stops running the expression when `null` or `undefined` is encountered.

```
const demolishedAlderaan = deathStar?.superLaser?.fire();
```

OPTIONAL CHAINING

```
// Before
let demolishedAlderaan;
if (deathStar != null && deathStar.superLaser != null) {
    demolishedAlderaan = deathStar.superLaser.fire();
}

// After
const demolishedAlderaan = deathStar?.superLaser?.fire();
```

NULLISH COALESCING

If first value is null or undefined, use fall back value.

```
const radius = deathStar.radius ?? 5000;
```

NULLISH COALESCING

```
// Before
const radius = deathStar.radius != null
    ? deathStar.radius
    : 5000 ;

// After
const radius = deathStar.radius ?? 5000;
```

GENERICS

GENERICS

Building reusable components is key to develop flexible and scalable apps.

Generics allow components to handle any kind of data instead of being restricted to a specific type.

QUEUE SOME NAMES

```
class Queue {
    private data = [];

    push(item) {
        this.data.push(item);
    }

    pop() {
        return this.data.shift();
    }
}

const nameQueue = new Queue();
nameQueue.push('luke');
nameQueue.push(123); // Oops, this won't be a valid name...

console.log(`Hello there, my name is ${nameQueue.pop().toUpperCase()}`);
console.log(`Hello there, my name is ${nameQueue.pop().toUpperCase()}`);
                    // Error: toUpperCase() is not a function
```

DEFINE A TYPE

Set a type for our queue to only allow one type.

```
class Queue {
    private data: string[] = [];

    push(item: string) {
        this.data.push(item);
    }

    pop(): string {
        return this.data.shift();
    }
}

const nameQueue = new Queue();
nameQueue.push('luke');
nameQueue.push(123); // Compile error: Argument of type '123' is not assignable
                     // to parameter of type 'string'...
```

But what if we want to have a queue of numbers, Persons, Cars,... ?

GENERIC QUEUE

To only allow a predefined type to be added to our queue, we introduce a Generic.

```
class Queue<T> {
    private data: T[] = [];

    push(item: T) {
        this.data.push(item);
    }

    pop(): T {
        return this.data.shift();
    }
}

const nameQueue = new Queue<string>(); // Here we define the type as string
nameQueue.push('luke');
nameQueue.push(123); // Compile eror: Argument of type '123'
                     // is not assignable to parameter of type 'string'...

const numberQueue = new Queue<number>(); // Here we define the type as number
numberQueue.push(123);
```

COMMON USECASE

Define type of data that will be returned from a Service.

```
class Service<T> {
    private data: T[] = [];

    get(): T[] {
        return this.data;
    }

    post(item: T): T {
        return this.data.push(item);
    }
}

const personService = new Service<Person>();
const carService = new Service<Car>();
```

UNION TYPE

UNION TYPE

Defines several types that are allowed for a value.

```
class XWing {
    wings: number;
}

class TieFighter {
    turrets: number;
}

class ATAT {
    legs: number;
}

let starfighter: XWing | TieFighter;
starfighter = new XWing();
starfighter = new TieFighter();
starfighter = new ATAT(); // Compile error: Type 'ATAT' is not assignable
                          // to type 'XWing | TieFighter'.
```

UNION TYPE WITH PARAMS

Allow several types as param for a function.

```
class XWing { wings: number; }

class ATAT { legs: number; }

class RebelOutpost { hangars: number; }

function getAmountOfAttachments(vehicle: XWing | ATAT): number {
    if (vehicle instanceof XWing) {
        return vehicle.wings;
    } else {
        return vehicle.legs;
    }
}

getAmountOfAttachments(new XWing());
getAmountOfAttachments(new ATAT());
getAmountOfAttachments(new RebelOutpost()); // Compile error: Argument of type
            // 'RebelOutpost' is not assignable to parameter of type 'XWing | ATAT'.
```

TYPE GUARD

TypeScript will be aware of the type inside the conditional block.

```
class XWing { wings: number; }

class ATAT { legs: number; }

function getAmountOfAttachments(vehicle: XWing | ATAT): number {
    console.log(vehicle.wings); // Compile error: Property 'wings'
                // does not exist on type 'ATAT'
    if (vehicle instanceof XWing) {
        return vehicle.wings;
    } else {
        return vehicle.legs; // Since vehicle isn't a XWing, it must be an ATAT
    }
}
```

WRITE YOUR OWN TYPE GUARDS

```
class XWing { wings: number; }

class ATAT { legs: number; }

function isXWing(vehicle: any): vehicle is XWing { // Returns a type predicate
    return vehicle instanceof XWing;
}

function getAmountOfAttachments(vehicle: XWing | ATAT): number {
    if (isXWing(vehicle)) {
        return vehicle.wings;
    } else {
        return vehicle.legs;
    }
}
```

MODULES

GLOBAL SCOPE

All code is in global scope by default. Variables, functions, etc. are accessible from other files.

MODULES

Organize code and prevent it from being accessible on the global scope by using modules.

EXPORT

Export any declaration you want to expose by adding the export prefix.

```
export interface DeathStarPlan {
    radius: number;
}

interface SecretDeathStarPlan {
    color: string;
}
```

IMPORT

Import other modules with the import statement.

```
import { DeathStarPlan } from 'path_to_interface';

class DeathStar implements DeathStarPlan {
    // Implementation...
}
```

RENAME IMPORT STATEMENTS

To prevent name collision or for cleaner code, you can rename the module that you're importing.

```
import { DeathStarPlan as DeathStarBlueprint } from 'path_to_interface';
import { DeathStarPlan as DeathStarScaffold } from 'path_to_other_interface';

class DeathStar implements DeathStarBlueprint, DeathStarScaffold {
    // Implementation...
}
```

IMPORT * AS

Imports all the exported declarations from another module into a single variable.

```
// death-star-plan.ts

export interface DeathStarPlan {
    radius: number;
}

export const DEATH_STAR_1_RADIUS = 60;
```

```
// death-star.ts

import * as DeathStarBlueprint from './deathstar-plan.ts';

class DeathStar implements DeathStarBlueprint.DeathStarPlan {
    radius = DeathStarBlueprint.DEATH_STAR_1_RADIUS;
}
```

BARREL

Collect all exports from several modules in a singe module. The barrel will re-export them to other modules. Common pattern in a lot of libraries.

WITHOUT A BARREL

```
// death-star/death-star.ts
export class DeathStar implements DeathStarPlan { }
```

```
// death-star/death-star-plan.ts
export interface DeathStarPlan { }
```

```
// app.ts
import { DeathStar } from './death-star/death-star';
import { DeathStarPlan } from './death-star/death-star-plan';

class App {}
```

WITH A BARREL

```
// death-star/index.ts
export { DeathStar } from './death-star';
export { DeathStarPlan } from './death-star-plan';
```

```
// app.ts
import { DeathStar, DeathStarPlan } from './death-star';

class App {}
```

EXERCISE 4: SHIPYARD

Create multiple classes that interact with eachother.

UTILITY TYPES

UTILITY TYPES

Global available types

Bundle of common transformations

PARTIAL

Marks all properties of given type as optional

```
interface DeathStarPlan {
    radius: number;
    color: string;
}

const partialPlan: Partial<DeathStarPlan> = {
    radius: 12
} // radius and color is not mandatory
```

REQUIRED

Opposite of Partial, makes all properties required

```
interface DeathStarPlan {
    radius?: number;
    color?: string;
}

const requiredPlan: Required<DeathStarPlan> = {
    radius: 12,
    color: 'blue'
} // even though DeathStarPlan has optional properties, the requiredPlan must provide
```

READONLY

Prevents properties for being re-assigned

```
interface DeathStarPlan {
    radius: number;
    color: string;
}

const finalPlan: Readonly<DeathStarPlan> = {
    radius: 12,
    color: 'blue'
}

finalPlan.color = 'red'; // Won't compile
```

PICK

Pick properties from the given type

```
interface DeathStarPlan {
    radius: number;
    color: string;
}

const colorPlan: Pick<DeathStarPlan, 'color'> = {
    color: 'red'
}
```

OMIT

Omits properties from the given type

```
interface DeathStarPlan {
    radius: number;
    color: string;
}

const colorPlan: Omit<DeathStarPlan, 'radius'> = {
    color: 'red'
}
```

PROMISES

PROMISE

Placeholder for a future value.

Same as callbacks, but a cleaner solution to write asynchronous code.

PROMISE

Promise either resolves with a value or rejects with an error.

```
function myAsyncCall(): Promise<string> {
    return new Promise<string>((resolve, reject) => {
        setTimeout(() => {
            if (2 * 2 === 4) {
                resolve('Hi');
            } else {
                reject(new Error('Oops!'));
            }
        }, 1000);
    });
}

myAsyncCall()
    .then(result => console.log('Received result', result))
    .catch(error => console.error('Something went wrong...', error));
```

CHAIN PROMISES

Promises can be chained together.

```
function printLightsaberColor(name: string) {
    getJedi(name)
        .then((jedi: Jedi) => getLightsaber(jedi.id))
        .then((lightsaber: Lightsaber) =>
            console.log('Color of lightsaber', lightsaber.color)
        )
        .catch((error: Error) =>
            console.error('Something went wrong...', error.message)
        );
}
```

ASYNC/AWAIT

Write your asynchronous code in a synchronous way.

```
async function printLightsaberColor(name: string) {
    try {
        const jedi = await getJedi(name);
        const ligthsaber = await getLightsaber(jedi.id);
        console.log('Color of lightsaber', lightsaber.color);
    } catch (error) {
        console.error('Something went wrong...', error.message);
    }
}
```

TYPE DEFINITIONS

TYPE DEFINITIONS

Community-driven collection of declaration files, covering over 90% of all JavaScript projects.

Integrate JS libraries in your project and use its API with all TypeScript goodies.

http://definitelytyped.org/

HOW CAN I USE IT?

First get a definition from @types.

```
npm install @types/jquery --save-dev
```

Afterwards you can access the definitions straight from your code.

```
import * as $ from 'jquery';

$('button').text('Click me!');
```

BEST PRACTICES

LOWERCASE TYPES

Keep in mind to lowercase types such as `number`, `string`,...

```
const badTypedString: String = 'Uppercase primary types are bad';
const goodTypedString: string = 'Be aware to lowercase primary types';
```

INFERENCE AWAY!

Let the compiler do the work for you: infer the type instead of adding the type yourself.

```
const randomText: string = 'Test';
const inferedText = 'Test';
```

PREVENT ANY TYPES

Use the `any` type as little as possible.

STRICT MODE

Forces you to use types, null checks,...

TSLINT

Get your code checked for best practices, readability,... before committing

EXERCISE 5: WEB APP

Develop a web app that communicates with an API.

ADDITIONAL RESOURCES

ADDITIONAL RESOURCES

- TypeScript documentation
- TypeScript playground
- Pluralsight course
- TypeScript Deep Dive book

THANKS FOR WATCHING!

May the Type be with you