TypeScript features

https://angularexperts.io/blog/advanced-typescript

Typescript offers so many great features. Here's a summary of some of the greatest advanced Typescript features.

Union and intersection types

Typescript allows us to combine multiple types to create a new type. This approach is similar to logical expressions in JavaScript where we can use the logical OR `||` or the logical AND `&&` to create new powerful checks.

Union types

A union type is similar to Javascripts OR expression. It allows you to use two or more types (union members) to form a new type that may be any of those types.

```
function orderProduct(orderId: string | number) {
   console.log('Ordering product with id', orderId);
}

// 👍
orderProduct(1);

// 👍
orderProduct('123-abc');

// 👎 Argument is not assignable to string | number
orderProduct({ name: 'foo' });
```

We type the `orderProduct` method with a union type. TypeScript will throw an error once we call the `orderProduct` method with anything that is not a `number` or a `string`.

Intersection types

An intersection type, on the other hand, combines multiple types into one. This new type has all the features of the combined types.

```
interface Person {
   name: string;
   firstname: string;
}

interface FootballPlayer {
   club: string;
}

function transferPlayer(player: Person & FootballPlayer) {}

// 👍
transferPlayer({
   name: 'Ramos',
   firstname: 'Sergio',
   club: 'PSG',
});

// 👎 Argument is not assignable to Person & FootballPlayer
transferPlayer({
   name: 'Ramos',
   firstname: 'Sergio',
});
```

The `transferPlayer` method accepts a type that contains all features of both `Person` and `FootballPlayer`. Only an object containing the `name`, `firstname` and the `club` property is valid.

Keyof

Now that we know the union type. Let's have a look at the `keyof` operator. The `keyof` operator takes the keys of an interface or an object and produces a union type.

```
interface MovieCharacter {
   firstname: string;
   name: string;
   movie: string;
}
type characterProps = keyof MovieCharacter;
```

Got it! But when is this useful? We could also type the `characterProps` out.

```
type characterProps = 'firstname' | 'name' | 'movie';
```

`keyof` makes our code more robust and always keeps our types up to date. Let's explore this with the following example.

```
interface PizzaMenu {
   starter: string;
   pizza: string;
   beverage: string;
   dessert: string;
}
const simpleMenu: PizzaMenu = {
   starter: 'Salad',
   pizza: 'Pepperoni',
   beverage: 'Coke',
   dessert: 'Vanilla ice cream',
};

function adjustMenu(
   menu: PizzaMenu,
   menuEntry: keyof PizzaMenu,
   change: string,
) {
   menu[menuEntry] = change;
}
```

```
// 👍
adjustMenu(simpleMenu, 'pizza', 'Hawaii');
// 👍
adjustMenu(simpleMenu, 'beverage', 'Beer');

// 👎 Type - 'bevereger' is not assignable
adjustMenu(simpleMenu, 'bevereger', 'Beer');
// 👎 Wrong property - 'coffee' is not assignable
adjustMenu(simpleMenu, 'coffee', 'Beer');
```

The `adjustMenu` function allows you to change a menu. For example, imagine you like the `menuSimple` but you prefer to drink beer over a Coke. In this case, we call the `adjustMenu` function with the `menu`, the `menuEntry` and the `change`, in our case, a Beer.

The interesting part of this function is that the `menuEntry` is typed with the `keyof` operator. The nice thing here is that our code is very robust. If we refactor the `PizzaMenu` interface, we don't have to touch the `adjustMenu` function. It is always up to date with the keys of the `PizzaMenu`.

Typeof

`typeof` allows you to extract a type from a value. It can be used in a type context to refer to the type of a variable.

```
let firstname = 'Frodo';
let name: typeof firstname;
```

Of course, this doesn't make much sense in such simple scenarios. But let's look at a more sophisticated example. In this example, we use `typeof` in combination with `ReturnType` to extract typing information from a functions return type.

```
function getCharacter() {
   return {
       firstname: 'Frodo',
       name: 'Baggins',
   };
}

type Character = ReturnType<typeof getCharacter>;

/*
equal to

type Character = {
 firstname: string;
 name: string;
}
*/
```

In the example above, we create a new type based on the return type of the `getCharacter` function. Same here, if we refactor the return type of this function changes, our `Character` type is up to date.

Conditional types

The conditional ternary operator is a very well-known operator in Javascript. The ternary operator takes three operands. A condition, a return type if the condition is true, and a return type is false.

```
condition ? returnTypeIfTrue : returnTypeIfFalse;
```

The same concept also exists in TypeScript.

```
interface StringId {
   id: string;
}

interface NumberId {
   id: number;
}

type Id<T> = T extends string ? StringId : NumberId;

let idOne: Id<string>;
// equal to let idOne: StringId;

let idTwo: Id<number>;
// equal to let idTwo: NumberId;
```

In this example, we use the `Id` type util to generate a type based on a `string`. If `T` extends `string` we return the `StringId` type. If we pass a `number`, we return the `NumberId` type.

Utility types

Utility types are helper tools to facilitate common type transformations. Typescript offers many utility types.
The official TypeScript documentation offers a great list of all utility types.

Partial

The Partial utility type allows you to transform an interface into a new interface where all properties are optional.

```
interface MovieCharacter {
   firstname: string;
   name: string;
   movie: string;
}

function registerCharacter(character: Partial<MovieCharacter>) {}

// 👍
registerCharacter({
   firstname: 'Frodo',
});

// 👍
registerCharacter({
   firstname: 'Frodo',
   name: 'Baggins',
});
```

`MovieCharacter` requires a `firstname`, `name` and a `movie`. However, the signature of the `registerPerson` function uses the `Partial` utility to create a new type with optional `firstname`, optional `name` and optional `movie`.

Required

`Required` does the opposite of `Partial`. It takes an existing interface with optional properties and transforms it into a type where all properties are required.

```
interface MovieCharacter {
   firstname?: string;
   name?: string;
   movie?: string;
}

function hireActor(character: Required<MovieCharacter>) {}

// 👍
hireActor({
   firstname: 'Frodo',
   name: 'Baggins',
   movie: 'The Lord of the Rings',
});

// 👎
hireActor({
   firstname: 'Frodo',
   name: 'Baggins',
});
```

In this example the properties of `MovieCharacter` were optional. By using `Required` we transformed into a type where all properties are required. Therefore only objects containing the `firstname`, `name` and `movie` properties are allowed.

Extract

Extract allows you to extract typing information from a type. Extract accepts two parameters, first the Interface and second the type that should be extracted.

```
type MovieCharacters =
   | 'Harry Potter'
   | 'Tom Riddle'
   | { firstname: string; name: string };

type hpCharacters = Extract<MovieCharacters, string>;
// equal to type hpCharacters = 'Harry Potter' | 'Tom Riddle';

type hpCharacters = Extract<MovieCharacters, { firstname: string }>;
// equal to type hpCharacters = {firstname: string; name: string };
```

`Extract<MovieCharacters, string>` creates a union type `hpCharacters` that consists only of strings. `Extract<MovieCharacters, {firstname: string}>` on the other hand, it extracts all object types that contain a `firstname: string` type.

Exclude

Exclude does the opposite of extract. It allows you to generate a new type by excluding a type.

```
type MovieCharacters =
    | 'Harry Potter'
    | 'Tom Riddle'
    | { firstname: string; name: string };

type hpCharacters = Exclude<MovieCharacters, string>;
// equal to type hpCharacters = {firstname: string; name: string };

type hpCharacters = Exclude<MovieCharacters, { firstname: string }>;
// equal to type hpCharacters = 'Harry Potter' | 'Tom Riddle';
```

First, we generate a new type that excludes all strings. Next, we generate a type that excludes all object types containing `firstname: string`.

Infer type

`infer` allows you to create a new type. It's similar to creating a variable in Javascript with the keyword `var`, `let` or `const`.

```
type flattenArrayType<T> = T extends Array<infer ArrayType> ? ArrayType : T;

type foo = flattenArrayType<string[]>;
// equal to type foo = string;

type foo = flattenArrayType<number[]>;
// equal to type foo = number;

type foo = flattenArrayType<number>;
// equal to type foo = number;
```

Wow, the `getArrayType` looks pretty complicated. But actually, it’s not. Let’s go through it. `T extends Array<infer ArrayType>` checks if `T` extends an Array. Furthermore, we use the `infer` keyword to get a hold of the array type. Think of it as storing the type in a variable.
We then use the conditional type to return the ArrayType if `T` extends Array. If not, we return `T`.