Advanced TypeScript Concepts

o Advanced Types

o Type Guards

o How to use type guards with type of, instance of

o Mapped Types

o Conditional Types

TypeScript Type Guards

Type Guards allow you to narrow down the type of a variable within a conditional block.

Typeof

Let's look at the following example:

```
type alphanumeric = string | number;

function add(a: alphanumeric, b: alphanumeric) {
    if (typeof a === 'number' && typeof b === 'number') {
        return a + b;
    }

    if (typeof a === 'string' && typeof b === 'string') {
        return a.concat(b);
    }

    throw new Error('Invalid arguments. Both arguments must be either numbers or strings.');
}
```

How it works:

- First, define the alphanumeric type that can hold either a string or a number.
- Next, declare a function that adds two variables a and b with the type of alphanumeric.
- Then, check if both types of arguments are numbers using the typeof operator. If yes, then calculate the sum of arguments using the + operator.

- After that, check if both types of arguments are strings using the typeof operator. If yes, then concatenate two arguments.
- Finally, throw an error if arguments are neither numbers nor strings.

In this example, TypeScript knows the usage of the typeof operator in the conditional blocks. Inside the following if block, TypeScript realizes that a and b are numbers.

```
if (typeof a === 'number' && typeof b === 'number') {
    return a + b;
}
```

Similarly, in the following if block, TypeScript treats a and b as strings, therefore, you can concatenate them into one:

```
if (typeof a === 'string' && typeof b === 'string') {
    return a.concat(b);
}
```

instanceof

Like the typeof operator, TypeScript is also aware of the usage of the instanceof operator. For example:

```
class Customer {
    isCreditAllowed(): boolean {
        // ...
        return true;
    }
}

class Supplier {
    isInShortList(): boolean {
        // ...
        return true;
    }
}

type BusinessPartner = Customer | Supplier;

function signContract(partner: BusinessPartner) : string {
    let message: string;
    if (partner instanceof Customer) {
        message = partner.isCreditAllowed() ? 'Sign a new contract with the customer' : 'Credit issue';
    }

    if (partner instanceof Supplier) {
        message = partner.isInShortList() ? 'Sign a new contract the supplier' : 'Need to evaluate further';
    }

    return message;
}
```

How it works:

First, declare the Customer and Supplier classes.
Second, create a type alias BusinessPartner which is a union type of Customer and Supplier.

Third, declare a function signContract() that accepts a parameter with the type BusinessPartner.

Finally, check if the partner is an instance of Customer or Supplier, and then provide the respective logic.

Inside the following if block, TypeScript knows that the partner is an instance of the Customer type due to the instanceof operator:

```
if (partner instanceof Customer) {
    message = partner.isCreditAllowed() ? 'Sign a new contract with the customer' : 'Credit issue';
}
```

Likewise, TypeScript knows that the partner is an instance of Supplier inside the following if block:

```
if (partner instanceof Supplier) {
    message = partner.isInShortList() ? 'Sign a new contract with the supplier' : 'Need to evaluate further';
}
```

When an if narrows out one type, TypeScript knows that within the `else` it is not that type but the other. For example:

```
function signContract(partner: BusinessPartner) : string {
    let message: string;
    if (partner instanceof Customer) {
        message = partner.isCreditAllowed() ? 'Sign a new contract with the customer' : 'Credit issue';
    } else {
        // must be Supplier
        message = partner.isInShortList() ? 'Sign a new contract with the supplier' : 'Need to evaluate further';
    }
    return message;
}
```

Mapped Types

Mapped types are a feature in TypeScript which allow you to map over a union of types to create a new type.

The syntax looks like this:

```
type Fruit = "apple" | "banana" | "orange";

type NewType = {
  [F in Fruit]: {
    name: F;
  };
};
/**
 * {
 *   apple: { name: "apple" };
 *   banana: { name: "banana" };
 *   orange: { name: "orange" };
 * }
 */
```

Let's break down each piece of syntax we're seeing here. The F in Fruit acts as a kind of index signature, which allows us to loop over each member of the Fruit union. You can think of this as being similar to a JavaScript for...of loop:

```
const fruit = ["apple", "banana", "orange"];

for (const f of fruit) {
  console.log(f);
}
```

In both cases, we get a very important thing: a closure over the current thing we're iterating over. In the JavaScript example, we get the current item. In the TypeScript example, we get the current member of the union.

This ability to cleanly map over each member of a union in a simple for...of model is what makes mapped types so powerful.

With keyof

Using the keyof operator with mapped types gives you a smooth API to create object types from other object types.

```
interface Person {
  name: string;
  age: number;
}

type NullablePerson = {
  [P in keyof Person]: Person[P] | null;
};

/**
 * {
 *   name: string | null;
 *   age: number | null;
 * }
 */
```

Here, we gain access to P, which represents either name the first time this is iterated over, then age the second time. This means we can gain access to the type of each property's value in Person by using Person[P].

We can then use this to create a new type that has the same keys and values as Person but with each property being nullable.

Conditional types in TypeScript

Conditional types let us deterministically define type transformations depending on a condition. In brief, they are a ternary conditional operator applied at the type level rather than at the value level.

Conditional types are defined as follows:

```
type ConditionalType = SomeType extends OtherType ? TrueType : FalseType
```

Conditional types can be recursive; that is, one, or both, of the branches can themselves be a conditional type:

```
type Recursive<T> = T extends string[] ? string : (T extends number[] ? number : never)

const a: Recursive<string[]> = "10" // works
const b: Recursive<string> = 10 // Error: Type 'number' is not assignable to type 'never'.
```

Constraints on conditional types

One of the main advantages of conditional types is their ability to narrow down the possible actual types of a generic type.

For instance, let's assume we want to define ExtractIdType<T>, to extract, from a generic T, the type of a property named id. In this case, the actual generic type T must have a property named id. At first, we might come up with something like the following snippet of code:

```
type ExtractIdType<T extends {id: string | number}> = T["id"]

interface NumericId {
    id: number
}

interface StringId {
    id: string
}

interface BooleanId {
    id: boolean
}

type NumericIdType = ExtractIdType<NumericId> // type NumericIdType = number
type StringIdType = ExtractIdType<StringId> // type StringIdType = string
type BooleanIdType = ExtractIdType<BooleanId> // won't work
```

Here, we made it explicit that T must have a property named id, with type either string or number. Then, we defined three interfaces: NumericId, StringId, and BooleanId.

If we attempt to extract the type of the id property, TypeScript correctly returns string and number for StringId and NumericId, respectively. However, it fails for BooleanId: Type 'BooleanId' does not satisfy the constraint '{ id: string | number; }'. Types of property 'id' are incompatible. Type 'boolean' is not assignable to type 'string | number'.

Still, how can we enhance our ExtractIdType to accept any type T and then resort to something like never if T did not define the required id property? We can do that using conditional types:

```
type ExtractIdType<T> = T extends {id: string | number} ? T["id"] : never

interface NumericId {
    id: number
}

interface StringId {
    id: string
}

interface BooleanId {
    id: boolean
}

type NumericIdType = ExtractIdType<NumericId> // type NumericIdType = number
type StringIdType = ExtractIdType<StringId> // type StringIdType = string
type BooleanIdType = ExtractIdType<BooleanId> // type BooleanIdType = never
```

By simply moving the constraint in the conditional type, we were able to make the definition of BooleanIdType work. In this second version, TypeScript knows that if the first branch is true, then T will have a property named id with type string | number.

Type inference in conditional types

It is so common to use conditional types to apply constraints and extract properties' types that we can use a sugared syntax for that. For instance, we could rewrite our definition of ExtractIdType as follows:

```
type ExtractIdType<T> = T extends {id: infer U} ? T["id"] : never

interface BooleanId {
    id: boolean
}

type BooleanIdType = ExtractIdType<BooleanId> // type BooleanIdType = boolean
```

In this case, we refined the ExtractIdType type. Instead of forcing the type of the id property to be of type string | number, we've introduced a new type U using the infer keyword. Hence, BooleanIdType won't evaluate to never anymore. In fact, TypeScript will extract boolean as expected.

infer provides us with a way to introduce a new generic type, instead of specifying how to retrieve the element type from the true branch.

Distributive conditional types

In TypeScript, conditional types are distributive over union types. In other words, when evaluated against a union type, the conditional type applies to all the members of the union. Let's see an example:

```
type ToStringArray<T> = T extends string ? T[] : never

type StringArray = ToStringArray<string | number>
```

In the example above, we simply defined a conditional type named ToStringArray, evaluating to string[] if and only if its generic parameter is string. Otherwise, it evaluates to never.

Let's now see how TypeScript evaluates ToStringArray<string | number> to define StringArray. First, ToStringArray distributes over the union:

```
type StringArray = ToStringArray<string> | ToStringArray<number>
```

Then, we can replace ToStringArray with its definition:

```
type StringArray =
  | (string extends string ? string[] : never)
  | (number extends string ? number[] : never);
```

Evaluating the conditionals leaves us with the following definition:

```
type StringArray = string[] | never
```

Since never is a subtype of any type, we can remove it from the union:

```
type StringArray = string[]
```

Most of the times the distributive property of conditional types is desired. Nonetheless, to avoid it we can just enclose each side of the extends keyword with square brackets:

```
type ToStringArray<T> = [T] extends [string] ? T[] : never
```

In this case, when evaluating StringArray, the definition of ToStringArray does not distribute anymore:

```
type StringArray = ((string | number) extends string ? (string | number)[] : never)
```

Hence, since string | number does not extend, string, StringArray will become never.

For exercises, check in class.