TypeScript Inference

This lesson describes how TypeScript infers types when we omit to provide one explicitly.

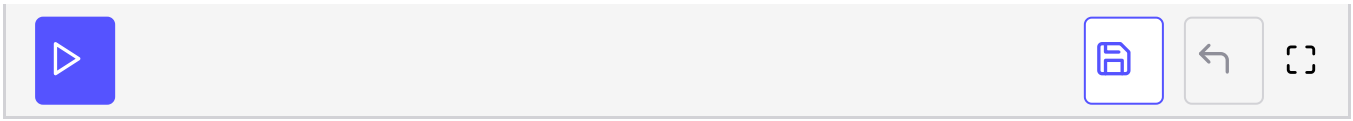
WE'LL COVER THE FOLLOWING

Control flow analysis

Since version 2.0, TypeScript checks the type of all possible flows. This means that TypeScript analyzes all paths to discover the most specific type a variable can be. In other words, if you define a variable to be many types, with a union `|`, TypeScript knows, line by line, what the actual type of the variable is. This is useful to get code to completion.

> **Code completion** is when the developer types, that the editor is suggesting functions or properties available. Good code completion editors provide useful possible code depending on the type.

```
let manyTypes: string | number;
manyTypes = "Line #2 is a string";
console.log(manyTypes.length); // Line 3 is a string, "length" is accessible
manyTypes = 100;
```



The code above shows that the variable has a dual-type of `string` or `number`.

The declaration at **line 1** uses the pipe (`|`) symbol to have the `manyTypes` variable accepting a `string` or a `number`.
During the flow of execution, the variable is actually first a `string` at **line 2** and then a `number` at line 4, never both.

Benefits of inference

The benefit of inference is to have a strongly typed code that adapts to the usage. It gives the right picture of its capabilities anytime. The following example does not compile: **line 5** has `length` variable only exists to the `string` type, not the `number`. The **line 4** changed the type from `string` inferred at **line 2** to a `number`.

> 📜**Note:** The following code does not compile ✕

```
let manyTypes: string | number;
manyTypes = "Line #2 is a string";
console.log(manyTypes.length); // Line 3 is a string, "length" is accessible
manyTypes = 100;
console.log(manyTypes.length); // Line 5 is a number, no access to the "length" function
```

Narrowing inferred type

Furthermore, if TypeScript was not narrowing the type down to a single one, only common properties would be accessible. To access `length`, an explicit cast would be required making the code convoluted to read.

```
function printMoney(money: number | undefined): void {
    if (money === undefined) {
        console.log("No money");
    } else {
        console.log(money.toFixed(2));
    }
}
printMoney(10);
printMoney(undefined);
```

The example above is a classic one where TypeScript can determine that the type is actually there (not undefined) and properly narrow down where the closure of the `if` statement should be. The code accepts the type `number` and `undefined` at **line 1**. At **line 2** it narrows the variable down to a single type by splitting with the if-else statement the `undefined` and `number`.

While at the function level it is not clear whether it is a `number` or `undefined`, the `if` statement indicates that between the curly brackets that the flow is without a doubt `undefined`. Since there are only two possibilities for the type, the `else` must contain a number.

The keyword `infer`

TypeScript 2.8 brings a new keyword, `infer`. The new addition returns a type from a generic. The role of inferring is to tell TypeScript to figure out the type instead of defining the type at the class or function level. Normally, generic requires you to declare the generic type from the start. However, in some situations, the type may not be known.

A use case of `infer` is when a type extends a function that returns an unknown type. It's possible to use `infer` to have the type be a generic variable.

```
type GetReturnedType<T> = T extends (...args: any[]) => infer R ? R : T;

function whatIsMyReturnType(): number {
    return 1;
}
// number from 'R'
type TypeFromReturn = GetReturnedType<typeof whatIsMyReturnType>;
const dynamicallyTyped: TypeFromReturn = 1;
// number from 'T'
type TypeFromReturn2 = GetReturnedType<number>;
```

In the previous code, the type `TypeFromReturn` gets the type by using a type's helper which expects a generic type to be a function. The function returns `infer R`. Before **TypeScript 2.8**, this would have been `any`. However, now, it is possible to infer the type. The example leverages the conditional type. If the generic type is not a function, it returns the actual type. The scenario happen at **line 10**.

at **line 10**.

The second type, named `TypeFromReturn2`, does not return `R` but instead `T` because it calls `GetReturnedType` with a primitive instead of a function. The **line** 7 use the type of the function.

The keyword does exactly what the name suggests, but it is not always obvious how it can be used. Let's see another example.

```
type CombinedType<T> = T extends { t1: infer U, t2: infer U } ? U : never;
type C1 = CombinedType<{ t1: string, t2: string }>;  // string
type C2 = CombinedType<{ t1: boolean, t2: number }>;  // boolean | number
```

The example above has at **line 1** a combined type. It mentions that both parameters need to infer its type inside `U`. Then, `U` is returned.

The **line 2** has one parameter has `string` and the other one `string`, hence `U` is `string`. However, **line 3** parameters are `boolean` and `number`. `U` type is `boolean | number`.

`infer` and classes

So far, `infer` was used with function or object. `infer` can also be used to extract the generic type of a class. In this section of the lesson, we will extract the type from a child class that inherits a generic base class. In the following example, **line 2** defines a base class that can take three types (`boolean`, `number` or `string`). At *line 6-7 * two children are defined, one that accepts a `boolean` and one that accepts a `string`. In the last two lines, **line 14-15** calls the function in which the parameter area strongly typed to the specific type specified in **line 6-7**.

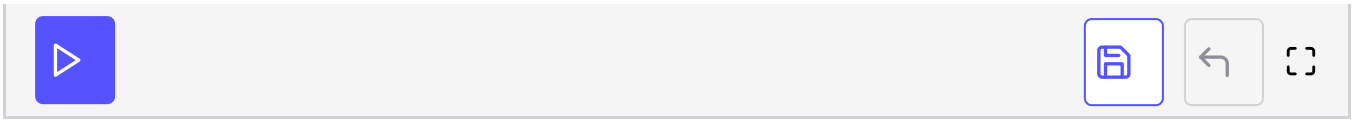
```
// Classes definition
class BaseClass<T extends boolean | number | string> {
  m(p1: T): void { console.log(p1); }
}

class Child1 extends BaseClass<boolean> { }
class Child2 extends BaseClass<string> { }

// Instanciation
const c1 = new Child1();
const c2 = new Child2();

// Invocation
```

```
c1.m(true);
c2.m("Test");
```



The question is how to extract that `c1` is taking only `boolean` and `c2` can only have `string`. The solution is to use `infer`. With the `infer` keyword you can extract the generic type as follow:

```
// Classes definition
class BaseClass<T extends boolean | number | string> {
  m(p1: T): void { console.log(p1); }
}

class Child1 extends BaseClass<boolean> { }
class Child2 extends BaseClass<string> { }

// Instanciation
const c1 = new Child1();
const c2 = new Child2();

// Invocation
c1.m(true);
c2.m("Test");

// Extraction with infer
type ExtractGeneric<T> = T extends BaseClass<infer U> ? U : T;
type E1 = ExtractGeneric<Child1>  // boolean
type E2 = ExtractGeneric<Child2>  // string
```

If you move your cursor above `E1` or `E2`, you will see that **line 19** is a `boolean` while **line 20** is a `string`.