TypeScript Type Inference

Summary: in this tutorial, you will learn about type inference in TypeScript.

Type inference describes where and how TypeScript infers types when you don't explicitly annotate them.

Basic type inference

When you declare a variable, you can use a type annotation to explicitly specify a type for it. For example:

```
let counter: number;
```

However, if you initialize the `counter` variable with a number, TypeScript will infer the type the `counter` to be `number`. For example:

```
let counter = 0;
```

It is equivalent to the following statement:

```
let counter: number = 0;
```

Likewise, when you assign a function parameter a value, TypeScript infers the type of the parameter to the type of the default value. For example:

```
function setCounter(max=100) {
    // ...
}
```

In this example, TypeScript infers the type of the `max` parameter to be `number`.

Similarly, TypeScript infers the following return type of the `increment()` function as `number`:

```
function increment(counter: number) {
    return counter++;
}
```

It is the same as:

```
function increment(counter: number) : number {
    return counter++;
}
```

The best common type algorithm

Consider the following assignment:

```
let items = [1, 2, 3, null];
```

To infer the type of `items` variable, TypeScript needs to consider the type of each element in the array.

It uses the best common type algorithm to analyze each candidate type and select the type that is compatible with all other candidates.

In this case, TypeScript selects the number array type (`number[]`) as the best common type.

If you add a string to the `items` array, TypeScript will infer the type for the items as an array of numbers and strings: `(number | string)[]`

```
let items = [0, 1, null, 'Hi'];
```

When TypeScript cannot find the best common type, it returns the union array type. For example:

```
let arr = [new Date(), new RegExp('\d+')];
```

In this example, TypeScript infers the type for `arr` to be `(RegExp | Date)[]`.

Contextual typing

TypeScript uses the locations of variables to infer their types. This mechanism is known as contextual typing. For example:

```
document.addEventListener('click', function (event) {
    console.log(event.button); //
});
```

In this example, TypeScript knows that the `event` parameter is an instance of `MouseEvent` because of the `click` event.

However, when you change the `click` event to the `scroll` the event, TypeScript will issue an error:

```
document.addEventListener('scroll', function (event) {
    console.log(event.button); // compiler error
});
```

Error:

```
Property 'button' does not exist on type 'Event'.(2339)
```

TypeScript knows that the `event` in this case, is an instance of `UIEvent`, not a `MouseEvent`. And `UIEvent` does not have the `button` property, therefore, TypeScript throws an error.

You will find contextual typing in many cases such as arguments to function calls, type assertions, members of objects and array literals, return statements, and right-hand sides of assignments.

Type inference vs. Type annotations

The following show the difference between type inference and type annotations:

Type inference	**Type annotations**
TypeScript guesses the type	You explicitly tell TypeScript the type

So, when do you use type inference and type annotations?

In practice, you should always use the type inference as much as possible. And you use the type annotation in the following cases:

- When you declare a variable and assign it a value later.
- When you want a variable that can't be inferred.
- When a function returns the `any` type and you need to clarify the value.

Summary

- Type inference occurs when you initialize variables, set parameter default values, and determine function return types.
- TypeScript uses the best common type algorithm to select the best candidate types that are compatible with all variables.
- TypeScript also uses contextual typing to infer types of variables based on the locations of the variables.