

TypeScript 3.2 is now available. [Download \(/#download-links\)](#) our latest version today!

Documentation ▼

Type Inference

Introduction

In this section, we will cover type inference in TypeScript. Namely, we'll discuss where and how types are inferred.

Basics

In TypeScript, there are several places where type inference is used to provide type information when there is no explicit type annotation. For example, in this code

```
let x = 3;
```

The type of the `x` variable is inferred to be `number`. This kind of inference takes place when initializing variables and members, setting parameter default values, and determining function return types.

In most cases, type inference is straightforward. In the following sections, we'll explore some of the nuances in how types are inferred.

Best common type

When a type inference is made from several expressions, the types of those expressions are used to calculate a "best common type". For example,

```
let x = [0, 1, null];
```

To infer the type of `x` in the example above, we must consider the type of each array element. Here we are given two choices for the type of the array: `number` and `null`. The best common type algorithm considers each candidate type, and picks the type that is compatible with all the other candidates.

Because the best common type has to be chosen from the provided candidate types, there are some cases where types share a common structure, but no one type is the super type of all candidate types. For example:

```
let zoo = [new Rhino(), new Elephant(), new Snake()];
```

Ideally, we may want `zoo` to be inferred as an `Animal[]`, but because there is no object that is strictly of type `Animal` in the array, we make no inference about the array element type. To correct this, instead explicitly provide the type when no one type is a super type of all other candidates:

```
let zoo: Animal[] = [new Rhino(), new Elephant(), new Snake()];
```

When no best common type is found, the resulting inference is the union array type, `(Rhino | Elephant | Snake)[]`.

Contextual Type

Type inference also works in “the other direction” in some cases in TypeScript. This is known as “contextual typing”. Contextual typing occurs when the type of an expression is implied by its location. For example:

```
window.onmousedown = function(mouseEvent) {  
    console.log(mouseEvent.clickTime); //<- Error  
};
```

For the code above to give the type error, the TypeScript type checker used the type of the `Window.onmousedown` function to infer the type of the function expression on the right hand side of the assignment. When it did so, it was able to infer the type of the `mouseEvent` parameter. If this function expression were not in a contextually typed position, the `mouseEvent` parameter would have type `any`, and no error would have been issued.

If the contextually typed expression contains explicit type information, the contextual type is ignored. Had we written the above example:

```
window.onmousedown = function(mouseEvent: any) {  
    console.log(mouseEvent.clickTime); //<- Now, no error is given  
};
```


The function expression with an explicit type annotation on the parameter will override the contextual type. Once it does so, no error is given as no contextual type applies.

Contextual typing applies in many cases. Common cases include arguments to function calls, right hand sides of assignments, type assertions, members of object and array literals, and return statements. The contextual type also acts as a candidate type in best common type. For example:

```
function createZoo(): Animal[] {  
    return [new Rhino(), new Elephant(), new Snake()];  
}
```

In this example, best common type has a set of four candidates: `Animal`, `Rhino`, `Elephant`, and `Snake`. Of these, `Animal` can be chosen by the best common type algorithm.

Made with ❤ in Redmond Follow @Typescriptlang (<https://twitter.com/typescriptlang>)

Privacy (<https://go.microsoft.com/fwlink/?LinkId=521839>) ©2012-2018 Microsoft  Microsoft