Why TypeScript

Summary: in this tutorial, you'll learn why you should use TypeScript over JavaScript to avoid the problems created by the dynamic types.

Why use TypeScript

There are two main reasons to use TypeScript:

- TypeScript adds a type system to help you avoid many problems with dynamic types in JavaScript.
- TypeScript implements the future features of JavaScript a.k.a ES Next so that you can use them today.

This tutorial focuses on the first reason.

Understanding dynamic type in JavaScript

JavaScript is dynamically typed. Unlike statically typed languages such as Java or C#, **values have types instead of variables**. For example:

```
"Hello"
```

From the value, you can tell that its type is `string`. Also, the following value is a number:

```
2020
```

See the following example:

```
let box;
box = "hello";
box = 100;
```

The type of the `box` variable changes based on the value assigned to it.

To find the type of the `box` variable at runtime, you use the `typeof` operator:

```
let box;
console.log(typeof(box)); // undefined

box = "Hello";
console.log(typeof(box)); // string

box = 100;
console.log(typeof(box)); // number
```

In this example, the first statement defines the variable without assigning a value. Its type is `undefined`.

Then, we assign `"Hello"` to `box` variable and show its type. The type of the `box` variable now changes to `string`.

Finally, we assign `100` to the `box` variable. This time, the type of the `box` variable changes to `number`.

As you can see, as soon as the value is assigned, the type of the variable changes.

And you don't need to explicitly tell JavaScript the type. JavaScript will automatically infer the type from the value.

Dynamic types offer flexibility. However, they also lead to problems.

Problems with dynamic types

Suppose you have a function that returns a `product` object based on an id:

```
function getProduct(id){
  return {
    id: id,
    name: `Awesome Gadget ${id}`,
    price: 99.5
  }
}
```

The following uses the `getProduct()` function to retrieve the product with id 1 and show its data:

```
const product = getProduct(1);
console.log(`The product ${product.Name} costs $${product.price}`);
```

Output:

```
The product undefined costs $99.5
```

It isn't what we expected.

The issue with this code is that the `product` object doesn't have the `Name` property. It has the `name` property with the first letter `n` in lowercase.

However, you can only know it until you run the script.

Referencing a property that doesn't exist on the object is a common issue when working in JavaScript.

The following example defines a new function that outputs the product information to the Console:

```
const showProduct = (name, price)  => {
  console.log(`The product ${name} costs $${price}.`);
};
```

And the following uses the `getProduct()` and `showProduct()` functions:

```
const product = getProduct(1);
showProduct(product.price, product.name);
```

Output:

```
The product 99.5 costs $Awesome Gadget 1
```

This time we pass the arguments in the wrong order to the `showProduct()` function. This is another common problem that you often have when working with JavaScript.

This is why TypeScript comes into play.

How Typescript solves the problems of dynamic types

To fix the problem of referencing a property that doesn't exist on an object, you do the following steps:

First, define the "shape" of the `product` object using an interface. Note that you'll learn about the interface in a later tutorial.

```
interface Product{
    id: number,
    name: string,
    price: number
};
```

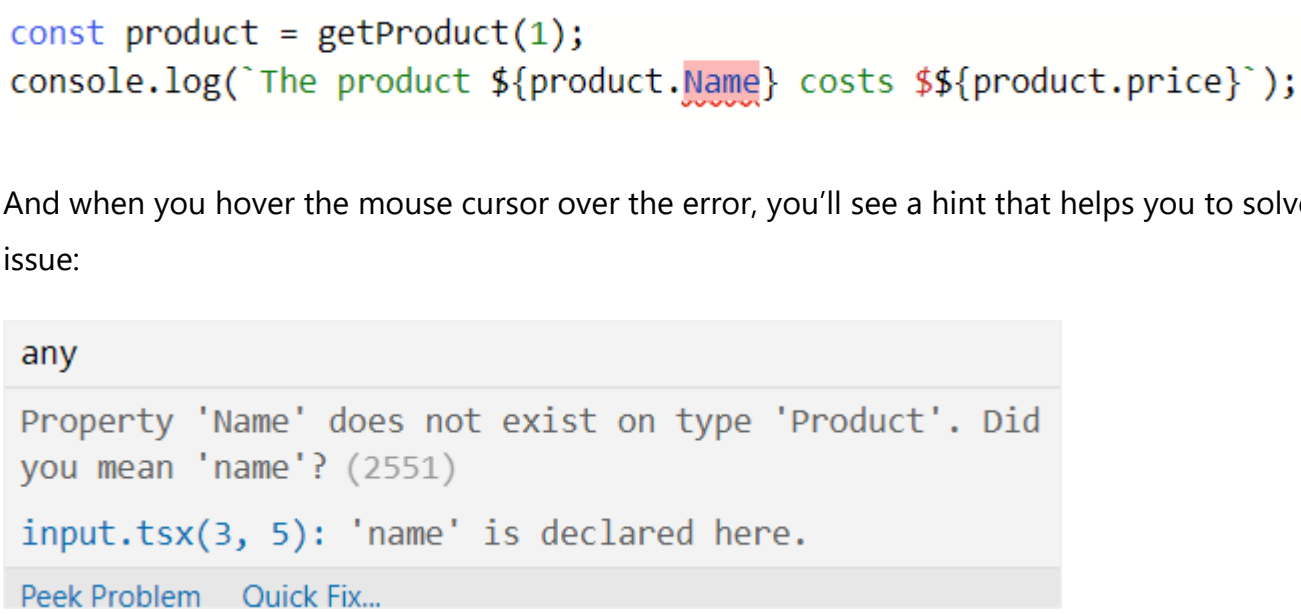
Second, explicitly use the `Product` type as the return type of the `getProduct()` function:

```
function getProduct(id) : Product{
  return {
    id: id,
    name: `Awesome Gadget ${id}`,
    price: 99.5
  }
}
```

When you reference a property that doesn't exist, the code editor will inform you immediately:

```
const product = getProduct(1);
console.log(`The product ${product.Name} costs $${product.price}`);
```

The code editor highlighted the following error on the `Name` property:



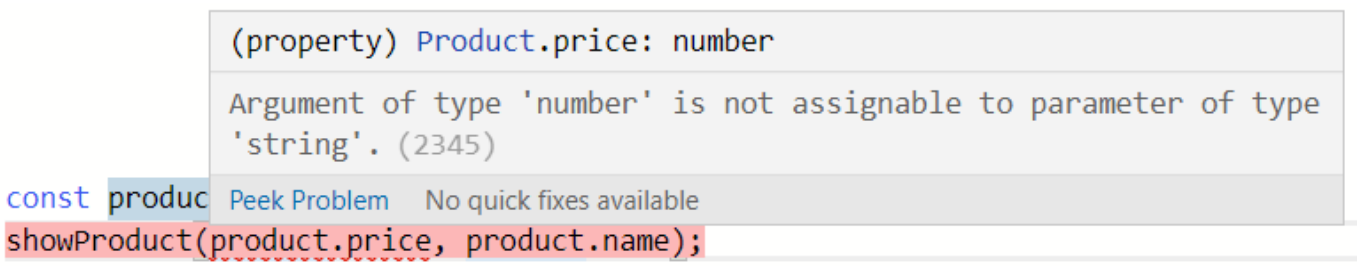
And when you hover the mouse cursor over the error, you'll see a hint that helps you to solve the issue:

To solve the problem of passing the arguments in the wrong order, you explicitly assign types to function parameters:

```
const showProduct = (name: string, price:number)  => {
  console.log(`The product ${name} costs $${price}.`);
};
```

And when you pass the arguments of the wrong types to the `showProduct()` function, you'll receive an error:

```
const product = getProduct(1);
showProduct(product.price, product.name);
```



Summary

- JavaScript is dynamically typed, providing flexibility but also leading to many problems.
- TypeScript adds an optional type system to JavaScript to solve these problems.