Comparing Variables

This lesson explains how TypeScript compares variables.

WE'LL COVER THE FOLLOWING

Comparing by value or by reference?

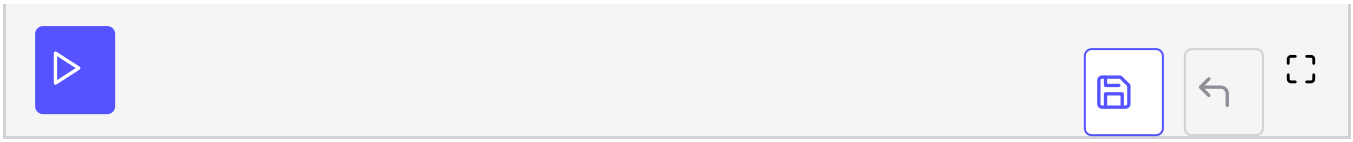
TypeScript inherited its lax comparison rules from Javascript. As we saw earlier, TypeScript reduces potential JavaScript quirks by increasing strictness. However, nothing forces developers to employ the “triple equals” comparison, which is key in avoiding the type conversions that may take place when using “double equals”. TypeScript helps with “double equals” by **removing some edge cases**.

For example, comparing a `number` with a number in a `string` works fine in JavaScript, but won’t compile with TypeScript. At **line 1** we declare a string initialized with the value `"1"` and at **line 2** we define a `number` with the value `1`. What is happening at **line 3** is a comparison that is always false. The reason is that a `string` and a `number` cannot be equal in value: they do not have the same type as a starter.

> 📜 Note: the below code throws an error ✕

```
let value1String: string = "1";
let value1Number: number = 1;
if (value1String == value1Number) { // Culprit
    console.log("TypeScript doesn't compile");
```

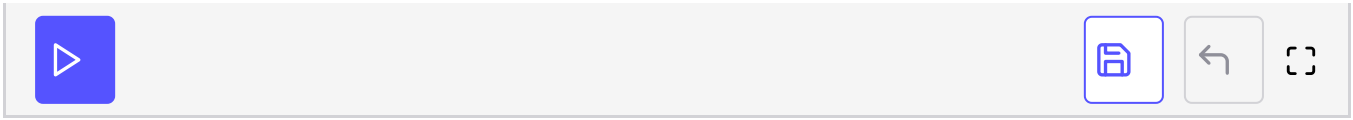
```
}
```



Primitive type compares by value. On the other hand, object literal, class objects, and arrays all compare by reference.

In the following code, **line 3** compares two objects that have the exact structure and values. Yet, it is not equal; the reason is that both objects are different because they are both initialized at two different places (**line 1** and **line 2**).

```
const value1Object = { m1: "test" }; //Not a primitive type: an object
const value2Object = { m1: "test" }; //Not a primitive type: an object
if (value1Object === value2Object) { // Reference check
    console.log("Should not print this line");
}
```



Comparing two objects

TypeScript uses structural typing instead of nominal typing. This means that types are compared by their structures and not by their names. Popular languages like C# or Java are nominal, as well as the alternative language similar to TypeScript called Flow. Nominal comparison happens ar design-time-only because JavaScript doesn't have types. TypeScript removes all types at run-time, hence it cannot have code to compare type names. In contrast, structural comparison, used by TypeScript, is available at design and run-time.

We already covered how to do type checking with `typeof` and `instanceof`. However, the comparison details on how flexible TypeScript remains. TypeScript doesn't allow us to provide the exact type and be able to pass a comparison check.

The important thing is to respect the structure behind the type. The emphasis on structure allows for the usage of any structure that has the minimum required intersection of fields.

For example, if there is a function that requires a parameter of an interface named `A` with a definition of a single member named `B`, you could pass any interface or type or a literal object that has a member named `B` with the same type as the original interface.

```
interface ParameterType {
  m1: number;
}
interface NotRelatedType {
  m1: number;
  m2: string;
  m3: boolean;
}
interface NotRelatedTypeNoM1Number {
  m1: string; // Same member name thant "ParameterType" interface but the type is string
}

const i1: ParameterType = { m1: 1 };
const i2: NotRelatedType = { m1: 1, m2: "1", m3: true };
const i3: NotRelatedTypeNoM1Number = { m1: "1" };

function IWantParameterType(p: ParameterType): void {}
IWantParameterType(i1);
IWantParameterType(i2);
// IWantParameterType(i3); // Doesn't compile
```

The previous code defines a first interface that has a single member: `m1: number`. The second interface has also the same member but defines at **line 6-7** two additionals members.

Line 13 to 15 initializes a variable of each interface. The **line 17** accepts a `ParameterType` and we can see that at **line 18** the exact type is passed successfully. We see that the `NotRelatedType` is also successful even if the structure is not the same. However, `NotRelatedType` intersects the members of `ParameterType` and because TypeScript is a structural language is enough to be compliant with the needed field of the function. However, the **line 20** is commented because it produces an error. The problem is that the `m1` type of `NotRelatedTypeNoM1Number` is `string` which is incompatible with the expected `number` defined at **line 2**.

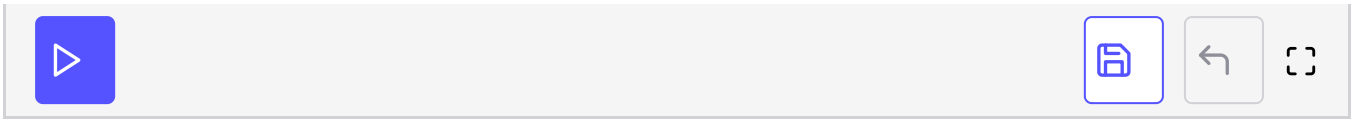
Comparing with `null` and `undefined`

`Null` and `undefined` are different when compared with “triple equals” but are the same when compared with “double equals”. In TypeScript, `undefined` is the type used for optional values, and `null` is used to represent “nothing is set here”. There is a trend of trying to rely only on undefined and completely avoid using null. However, you cannot take this for granted.

There is a pattern to compare `null` or `undefined`, which is to look up for the variable without using an operator. The lack of operator is problematic if the type is a `number` or a `Boolean`.

For example, a `Boolean` maybe `undefined` or `false`, and in both cases, it will fall into the same conditional statement. Confusing the value with `null` or `undefined` might not be frequent, but code tends to change. The return value of a function could begin with an object and end with a `Boolean`, which would still compile but change the behavior of how the code interprets the condition.

```
function fct(): Boolean {
  return new Boolean(false); // False
}
if (fct()) {
  console.log("The value is true"); //But still print because the object is defined
}
```

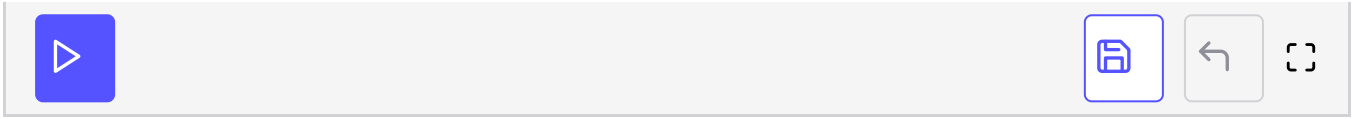


In the end, being safe while using a few more keystrokes is the key to a resilient code, ideally using the “triple equals” to check for undefined and then for `null` (if both cases apply).

Here is a second example showing the mix of `false` and `undefined`. The **line 1** initializes the value with `undefined` but could also have been `true` or `false` because it has a union with a `boolean` type. The short form, without equal signs, gets into the first condition while it may be expected to only be when `false` is defined.

```
let value2Boolean1: boolean | undefined = undefined;
let value2Boolean2: boolean | undefined = true;
let value2Boolean3: boolean | undefined = false;
if (!value2Boolean1) {
    console.log("This is false 1");
```

```
}
if (!value2Boolean2) {
    console.log("This is false 2");
}
if (!value2Boolean3) {
    console.log("This is false 3");
}
```



The pitfall of this comparison technique is when the code evolve. Starting with `boolean` only where it is clear when the condition is `false` but can change and allows `undefined` or `null` or even `0`.

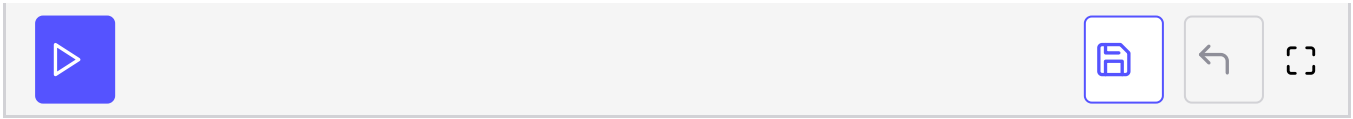
Comparing with “double equals”

The second scenario which requires fewer keystrokes is to use the double equals as `null` or `undefined`. In both cases, the verification tests the value of the object against `null` or `undefined`. Although it’s still using the “double equals”, it is still better than just comparing to nothing and having the object or primitive converted to Boolean.

An important configuration is to set `strictNullChecks` to `true`. This option in `tsconfig.json` will enforce a boundary between `null` and `undefined`.

> Note: the below does work as expected because of the tsconfig.json applied for this course

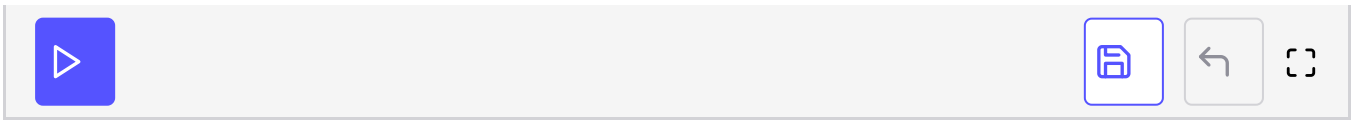
```
const x: boolean | undefined = undefined;
if (x == undefined) {
    console.log("x undefined");
}
if (x == false) { // if strictNullChecks is false it would go in this condition
    console.log("x is false");
}
```



These days, it’s totally safe to compare undefined with a “triple equals” sign. A long time ago, before ECMAScript 5, it was possible to redefine `undefined`, hence not safe to compare against it directly. That redefine is why it was

recommended to use `typeof` and to compare it against the string literal “undefined”. It’s also possible to compare `void 0` with a “triple equals”. The reason is that when the compared variable’s value is undefined, it will equal `void 0`. However, with TypeScript, the cleanest way to compare against undefined is to use the “triple equals” against the variable `undefined` directly.

```
const x: string | undefined = undefined;
if (x === undefined) {
    console.log("x undefined");
}
if (typeof x === "undefined") {
    console.log("x typeof undefined");
}
if (x === void 0) {
    console.log("x void 0");
}
```



TypeScript inherited the flexibility of JavaScript and it is up to the developer to use the style of comparison is appropriated. As a best and sure practice, comparing with the “triple equals” to the exact expected type is always the safest pattern.