Strict Null Checks

This lesson introduces the most important `strict` mode compiler flag - `strictNullChecks`. This flag can help you elevate the type safety of your code to a completely new level. It addresses the billion-dollar mistake of programming languages, null references.

WE'LL COVER THE FOLLOWING

The billion-dollar mistake

The expression comes from Thomas Hoare, a famous and influential computer scientist who introduced null references to ALGOL in 1965. Many years later he admitted that it was a mistake, as it resulted in *"innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years."*

What does it mean in the context of JavaScript and TypeScript?

```
Uncaught TypeError: Cannot read property 'foo' of undefined
```

Are you familiar with this error message? I bet you are. It occurs whenever you're trying to access a property or method of an object that you think is present, but turns out to not be there.

There are two ways of representing empty values in JavaScript - `null` and `undefined`. This kind of error can occur in both cases. `strictNullChecks` enables you to detect such mistakes at compile-time. It is an invaluable help and, when done properly, can lead to the complete eradication of this class of runtime bugs.

Types as sets

Types as sets

Let's break down the official definition of `strictNullChecks` from the documentation.

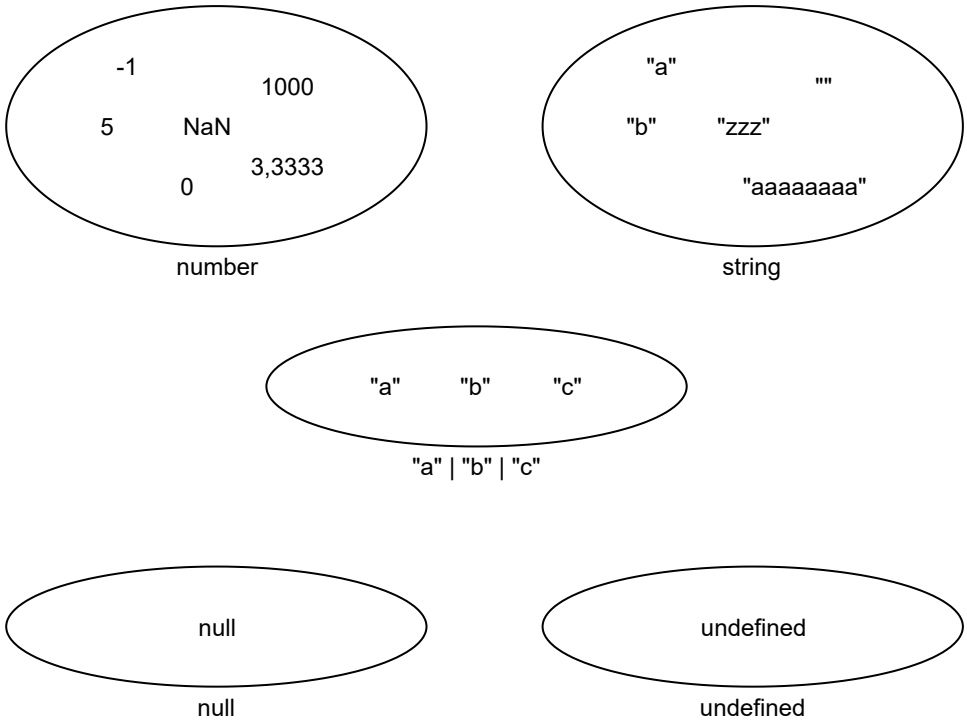
> In strict null checking mode, the null and undefined values are not in the domain of every type and are only assignable to themselves and any (the one exception being that undefined is also assignable to void).

The key sentence here is: *the null and undefined values are not in the domain of every type*. To fully understand this, let's look at TypeScript types from a mathematical perspective.

There is a concept called *set* in mathematics. A set is basically a grouping of things: numbers, fruit, other sets, functions. Sets can be finite or infinite. For example, `{1, 2, 5, 6}` is a finite set. The set of all the natural numbers is an infinite set.

Sets correspond to types very well. For example:

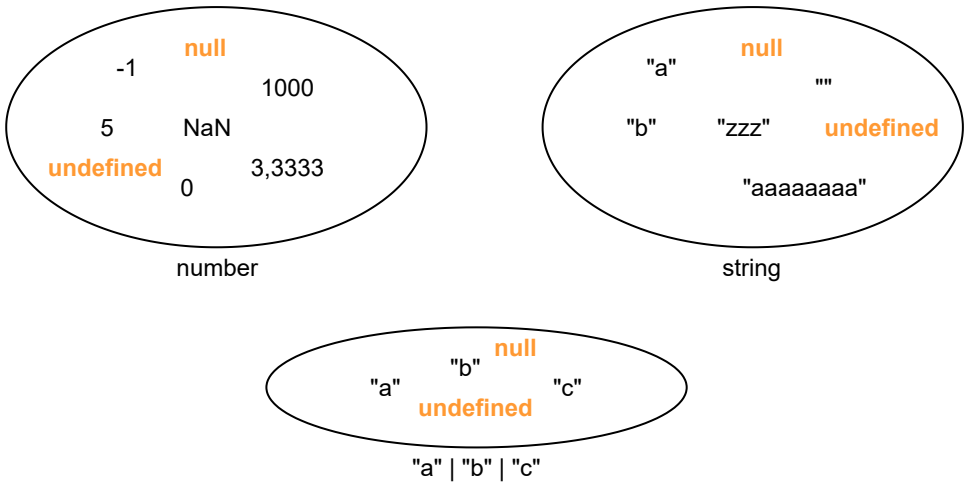
- `number` type can be looked at as a set of all possible numbers
- `string` type is a set of all possible strings
- `'a' | 'b' | 'c'` type corresponds to the `{'a', 'b', 'c'}` set
- `null` type is a singleton type with only one element, the `null` value (which happens to have the same name)
- `undefined` type is a singleton type with only one element, the `undefined` value (which happens to have the same name)



Set representation of some basic types

`strictNullChecks` explained

Back to `strictNullChecks`. The definition says that when the flag is *not* enabled, `null` and `undefined` values *are* in the domain of every type. In other words, `null` and `undefined` values belong to sets corresponding to every type.



Set representation of some basic types in TypeScript with `strictNullChecks` disabled

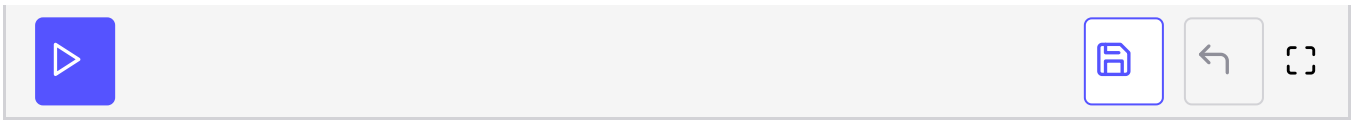
This is bad. This means that you can assign `null` and `undefined` values to a variable of every type because they are in the domain of every type. Consequently, you can later call a method on a variable whose value is `undefined` or `null`.

```
interface Person {
  hello(): void;

}

const n: number = undefined;
const f: string = null;
const person: Person = null;

person.hello(); // 🔴 Runtime Error!
```



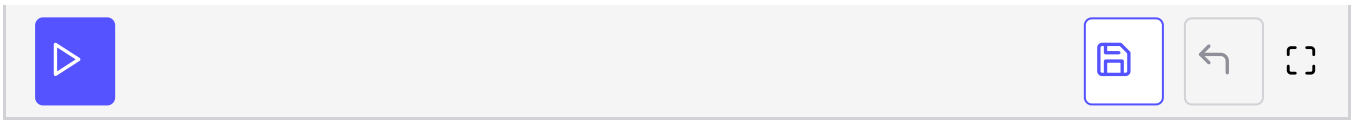
`strictNullChecks` flag disabled

With `strictNullChecks` enabled, this is no longer the case. `null` and `undefined` do *not* automatically belong to all the types. Therefore, TypeScript won't let you use `null` or `undefined` when a type that doesn't include these values is required (line 8). You can fix this by extending the type to include `null` or `undefined` (line 10).

```
interface Person {
  hello(): void;
}

function foo(person: Person) {
  person.hello();
}
foo(null); // 🔴 Error!

function bar(person: Person | null) {
  person.hello(); // 🔴 Error!
  if (person !== null) {
    person.hello();
  }
}
bar(null); // OK
```



`strictNullChecks` flag enabled

However, if you do that, TypeScript won't let you call a method on such a variable (line 11)! And it's completely right to do so; the value of this variable can be `null`, so you shouldn't be calling any methods on it without first making sure that it is not empty.

Why is this not the default behavior, you might ask. Interestingly, this is how

Why is this not the default behavior, you might ask. Interestingly, this is how popular languages like Java, C++, or C# work; they don't have null checks enabled by default. This is the mental model that most programmers are used to. TypeScript authors decided that enabling strict null checks could make the learning curve too steep which could negatively impact TypeScript adoption. Fortunately, they also made it possible to opt-out of the million-dollar mistake and make your code much safer.

In the next lesson, we'll see how enabling the `strictNullChecks` flag changes the way we work with optional types.