Type Constructors

This lesson explains theoretical foundations of generic types and gives you a glimpse into category theory. It provides a mental model that will prove useful in the next chapters.

WE'LL COVER THE FOLLOWING

Two worlds

Type constructor is a *function* (not a regular function, a mathematical function) that takes a type and returns a new type (based on the original type).

Notice that a generic interface with only one type argument is exactly that. For example, the `FormField` type from the previous lesson is a *function* with one type argument, `T`. It takes a type (e.g. `string`) and returns a new type (`FormField<string>` or `{ value?: string; defaultValue: string; isValid: boolean; }`).

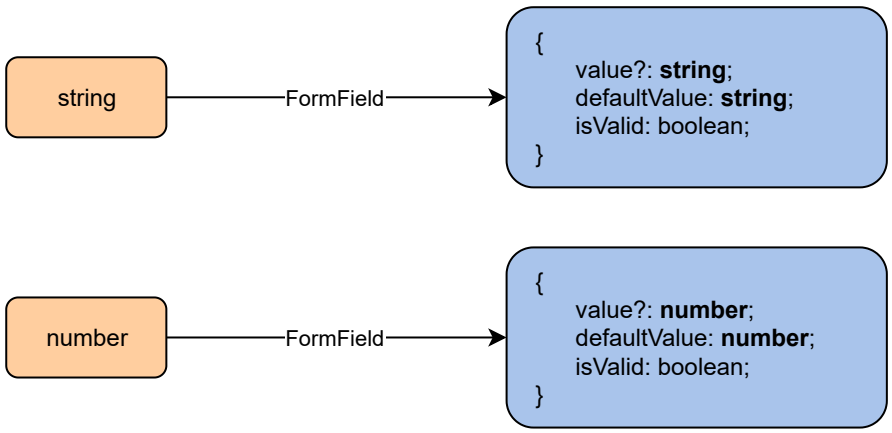

Diagram demonstrates how `FormField` acts like a function that maps a type to another type.

Try to think of *the world of types* as a universe that is parallel to *the world of values*. You're already familiar with *the world of values*, where functions take some values and return other values. In *the world of types*, type constructors

are functions that take some types and return other types.

Category theory

This analogy is not accidental. There is a branch of mathematics called **category theory**. It's a very abstract discipline that talks about **categories**. A category is anything that can be represented by a graph or a set of nodes with some arrows in between. It turns out that *the world of values* and *the world of types* are both categories! What's more, there are some universal laws that are true for all categories.

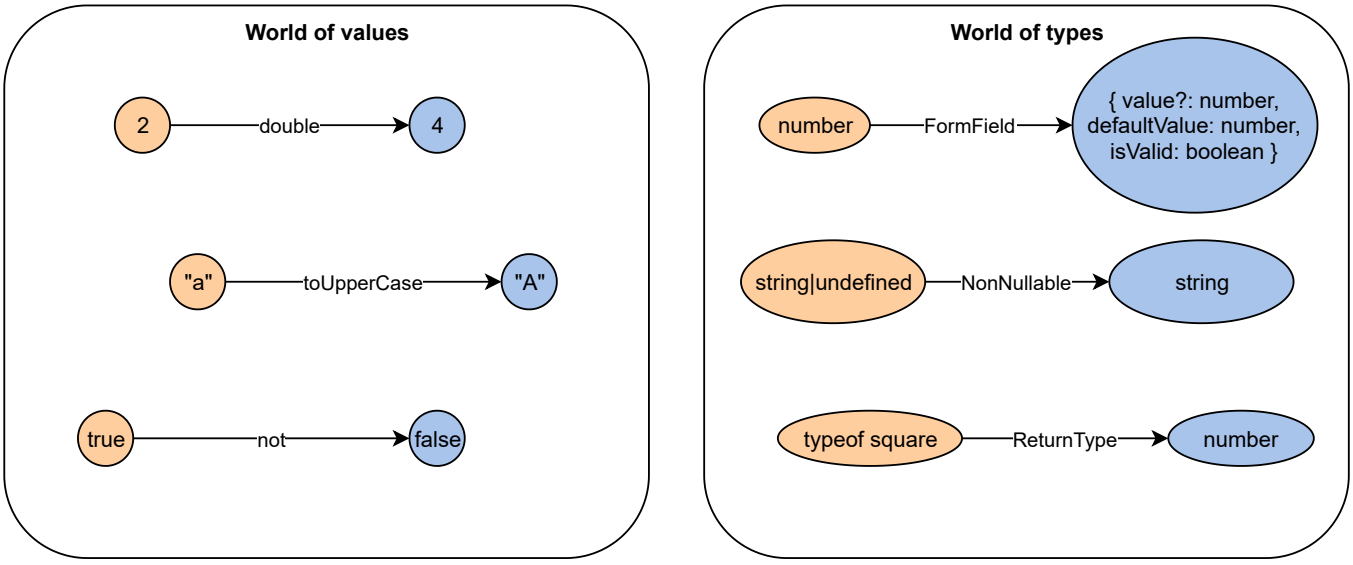

Diagram illustrates commonalities between the world of values and the world of types.

There are a few other interesting type constructors built into TypeScript. For example, `NonNullable<T>` is a *function* that takes a type and returns the same type with `null` and `undefined` removed from the domain.

`Partial` takes a type and returns a type with the same properties as the source, but each one optional.

`ReturnType` takes a function type and returns the return type of that function.

```
type X1 = NonNullable<string | undefined>; // X1 = string

type X2 = Partial<{ name: string; age: number; }>; // X2 = { name?: string; age?: number;}

function square(x: number): number { return x * x; }
type X3 = ReturnType<typeof square>; // X3 = number
```

Hover over `T1`, `T2` and `T3` to see the inferred types.

We'll look much closer at these types and more in the *Advanced types* chapter. The key takeaway from this lesson is that generic interfaces are very much like functions that transform types.

The final lesson in this chapter talks about an advanced typing scenario involving higher-order generic functions.