# TypeScript 102

Ethan Kent

Spoonflower

March 22, 2021

# Generics

IN A WORLD WITHOUT ABSTRACTION OVER VALUES...

```
const doubleOne = () => 2;
const doubleTwo = () => 3;
const doubleThree = () => 6;

// ... approximately infinity lines later:

const tripleOne = () => 3;
const tripleTwo = () => 6;
const tripleThree = () => 9;
```

# Generics, continued

Why is it hard? No abstracting over values. Really we're just using gussied-up constants.

# Generics, continued

Futhermore, if we invented new value, we'll have to implement the relevant "functions":

```
const TheNumberBlumpflorp = "Blumpflorp";

const doubleBlumpflorp = () => "BlumpflorpTimes2";
const tripleBlumpflorp = () => "BlumpflorpTimes3";
```

# Generics, continued

IN A WORLD WITHOUT ABSTRACTION OVER VALUES, ONE RENEGADE PROGRAMMER INVENTED FUNCTIONS...

```
const doubleIt = (input: number): number => 2 * input;

doubleIt(4); // => 8
```

# Generics, continued

Some terminology: `input` is a *parameter*. (Think *p* for *p*otential.)

```
const doubleIt = (input: number): number => 2 * input;
```

> Dearest TypeScript:
>
> I shall write you in the future and tell you what value to bind to `input`. Until then, all my love.
>
> Forever yours (as I am `const` bound),
> `doubleIt`, function.

# Generics, continued

Some more terminology: the thing inside the parentheses is an *argument*. (Think *a* for *a*ctual.)

```
doubleIt(4); // => 8
```

Most Honorable `doubleIt`:

I write on behalf of TypeScript.

I am in receipt of the missive of last clock tick. It is with pleasure that I have bound the number `4` to `input`, and proceededing, *mutatis mutandis*, in the witty & delightful & v. droll manner that you have prescribed, thereby obtained the result `8`.

Yr. faithful & obedient servant, &c.,
The JavaScript Runtime

# Generics, continued

What did we just learn?

- If we can't abstract over values, the programmer, language, or framework must explicitly define the cases (as with `doubleOne` and `tripleTwo`).
- If we can't abstract over values, a language or framework cannot support operations with user-defined values, so the programmer must provide all implementations she will rely on (as with the new number `Blumpflorp`).

# Generics, continued

```
const firstName: string | undefined | null =
  getFirstName();

const lastName: string | undefined | null =
  getFirstName();
```

Annoy-ing! How about this:

```
type MaybeString = string | undefined | null;

const firstName: MaybeString = getFirstName();
const lastName: MaybeString = getFirstName();
```

# Generics, continued

IN A WORLD WITHOUT ABSTRACTION OVER ~~VALUES~~ TYPES...[1]

Imagine we whip up a library for this *Maybe* idea.

```
type MaybeString = String | undefined | null;
type MaybeNumber = Number | undefined | null;
type MaybeBoolean = Boolean | undefined | null;
type MaybeSymbol = Symbol | undefined | null;
type MaybeObject = Object | undefined | null;
```

⋮

If we can't abstract over ~~values~~ types, the programmer, language, or framework must explicitly define the cases.

[1] We don't actually have to imagine. This dystopian world is real. It is a place of weeping and gnashing of teeth: a place called Golang.

# Generics, continued

IN A WORLD WITHOUT ABSTRACTION OVER ~~VALUES~~ TYPES...

But now our user has defined a `Name` interface. Our library is of no help, except as inspiration. Our user must implement `MaybeName` himself:

```
interface Name {
  givenName: string;
  familyName: string;
}

type MaybeName = Name | undefined | null;
```

> If we can't abstract over ~~values~~ types, a language or framework cannot support operations with user-defined ~~values~~ types.

# Generics, continued

IN A WORLD WITHOUT ABSTRACTION OVER ~~VALUES~~ TYPES, ONE RENEGADE PROGRAMMER INVENTED ~~FUNCTIONS~~ GENERICS...

```
type Maybe<T> = T | undefined | null;

type MaybeString = Maybe<String>;
type MaybeName = Maybe<Name>;
```

# Generics, continued

Some terminology: `T` is a type *parameter*. (Think *p* for *p*otential.)

```
type Maybe<T> = T | undefined | null;
```

> Dearest TypeScript:
>
> I shall write you in the future and tell you what ~~value~~ type to bind to `T`. Until then, all my love.
>
> Forever yours,
> `Maybe`, generic type.

# Generics, continued

Some more terminology: `String` is a type *argument*. (Think *a* for *a*ctual.)

```
type MaybeString = Maybe<String>; // => String | undefined | null
```

Most Honorable `Maybe`:

I am in receipt of your compile-time missive. It is with pleasure that I have bound the type `String` to `T`, and proceeding, *mutatis mutandis*, in the witty & delightful & v. droll manner that you have prescribed, thereby obtained the result `String | undefined | null`.

Yr. faithful & obedient servant, &c.,
The TypeScript Compiler[a]

[a] Oooh, interesting, this is a little different than last time. Is this—dare I even hope—foreshadowing a coming topic?

# Generics, continued

So generics are like functions, except:

- You pass in a type instead of a value.
- You use < and > instead of ( and ).
- Instead of returning a value, the expression returns a type.

# Runtime vs. Compile Time

# Utility Types

# React

# Index Types

# const and readonly

# Type Guards