# TypeScript 102

Ethan Kent

Spoonflower

April 13, 2021

# Generics

IN A WORLD WITHOUT ABSTRACTION OVER VALUES...

```
const doubleOne = () => 2;
const doubleTwo = () => 4;
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 a new kind of number, we'll have to implement the relevant "functions":

```
const i = "i";

const doubleI = () => "2i";
const tripleI = () => "3i";
```

# 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 `i`).

# Generics, continued

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

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

Annoy-ing! How about this:

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

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

# 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.
>
> Possibly yours (as I have trouble with commitment),
> `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.

# Generics, continued

```
const stringArray: Array<string> =
  ["Some", "good", "stuff"];

interface Bloop {
  grapplingHookLength: number;
  presenceOfPiranas: boolean;
}

const myBloopArray: Array<Bloop> = [
  { grapplingHookLength: 27, presenceOfPiranas: true },
  { grapplingHookLength: 5280, presenceOfPiranas: false },
];
```

# React

To update

# Runtime vs. Compile Time

Repeat this to yourself as a mantra:

> *TypeScript doesn't exist when my code runs.*
> *TypeScript doesn't exist when my code runs.*
> *TypeScript doesn't exist when my code runs.*
> *TypeScript doesn't exist when my code runs.*
> *TypeScript doesn't exist when my code runs.*

Example

# Runtime vs. Compile Time, continued

```
interface TypeScriptDisillusionment {
  intensity: number;
  isRecoverable: boolean;
}

const myFeelingsRightNow: TypeScriptDisillusionment = {
  intensity: 42,
  isRecoverable: true,
}

if (typeof myFeelingsRightNow === "TypescriptDisillusionment") {
  console.log("Maybe I'm not so disillusioned after all.");
} else {
  console.log("I'm very disillusioned.")
}
```

# Runtime vs. Compile Time, continued

Error message:

```
This condition will always return 'false' since the types '"string" |
"number" | "bigint" | "boolean" | "symbol" | "undefined" | "object" |
"function"' and '"TypescriptDisillusionment"' have no overlap.(2367)
```

Output (transpiled despite compiler errors):

```
[LOG]: "I'm very disillusioned."
```

# Runtime vs. Compile Time, continued

*TypeScript doesn't exist when my code runs.*
*TypeScript doesn't exist when my code runs.*
*TypeScript doesn't exist when my code runs.*
*TypeScript doesn't exist when my code runs.*
*TypeScript doesn't exist when my code runs.*

# Type Guards

That's the main point. Is there any solution? Yes, kind of.

> Yo, TypeScript, programmer here. I know you get all weird and start yelling about, "I just can't with these runtime types" and all. I get it. But like, you trust me, right? I mean, you always let me say `any` and `@ts-ignore` and stuff, right?
>
> Okay, so I wrote this function that I promise will figure out the types at runtime. No, no, don't freak out. Just look at it, will you? Yes, at compile time. Yes, I know it doesn't run at compile time. You're missing my point.
>
> I promise it will work at runtime to figure out the types. So can you *please* look at it at compile time to make sure I'm getting everything else right. Aww, you're the best TypeScript.

# Type Guards, continued

1. Manual non typesafe approach.
2. Reminder of TS awesomeness.
3. Demo of a type guard.
4. Not totally typesafe.
5. Using a dedicated field.

# Utility Types

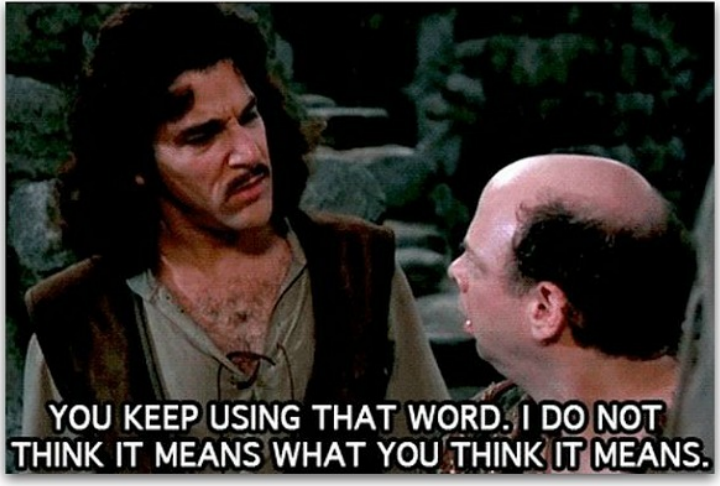
```ts
interface MysteryPerson {
  firstName?: string;
  lastName?: string;
}

const myMysteryPerson: MysteryPerson = {};

type ForthrightPerson = Required<MysteryPerson>;

// Type error: Type '{}' is missing the following properties from type
// 'Required<MysteryPerson>': firstName, lastName ts(2739)
const myForthrightPerson: ForthrightPerson = {};
```

# Advanced Types

Let's build our own Utility Types!

Dauntlessly live coding.

# const and readonly



# const and readonly, continued

```
type ArrayForever<T> = readonly T[];

const myArray: ArrayForever<Number> = [1, 2, 3];

// Error: Index signature in type 'ArrayForever<Number>'
// only permits reading.
myArray[0] = 33;
```

# const and readonly, continued

```
interface Person {
  readonly firstName: string;
  readonly lastName: string;
  yesterdaysDinner: string;
}

const ethan: Person = {
  firstName: "Ethan",
  lastName: "Kent",
  yesterdaysDinner: "Rotisserie Chicken",
};

// Fine
ethan.yesterdaysDinner = "Bananas Foster";

// Error: Cannot assign to 'lastName' because it is a read-only property.
ethan.lastName = "Vigliodogsworthy";
```

# Index Types

Fixme

# Conditional Types

Implement me

# Conditional Types, continued

```
type Validation<T> = (value: T) => boolean;

type FormValidation = {
  [k in keyof Form]: Form[k] extends object
    ? {
        [l in keyof Form[k]]: Form[k][l] extends object
          ? { [m in keyof Form[k][l]]: Validation<Form[k][l][m]> }
          : Validation<Form[k][l]>;
      }
    : Validation<Form[k]>;
};
```