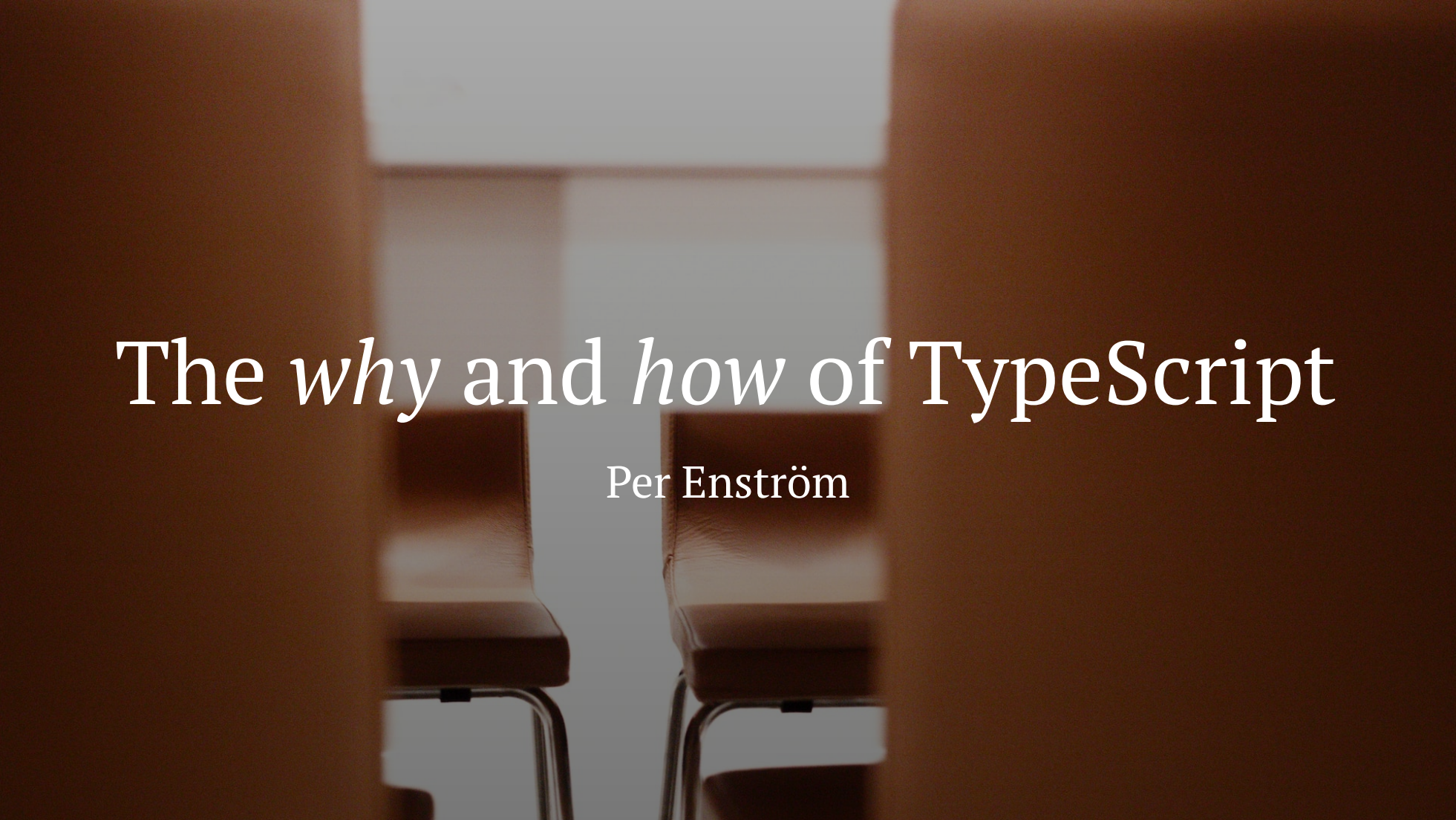


The *why* and *how* of TypeScript

Per Enström

First, a short description on *what*

TypeScript is a strongly typed programming language that builds on JavaScript, giving you better tooling at any scale.

typescriptlang.org

But what does that mean?

- It’s a separate language
- It gets compiled to plain old JavaScript
- "Strongly typed" means that you specify what type everything is
  - String
  - Numbers
  - Booleans
  - Custom types

Why should I use it?

What if I told you…

…that you already are

probably

VS Code

VS Code is built on TypeScript, and that is what is powering the intellisense functionality.

```
const myArr = [1, 2, 3, 4, 5];

myArr.map((item) => null);
// ^? const myArr: number[]
```

Aside: TwoSlash queries

```
1  const myArr = [1, 2, 3, 4, 5];
2
3  myArr.map((item) => null);
4  // ^?
```

Aside: TwoSlash queries

```
const myArr = [1, 2, 3, 4, 5];

myArr.map((item) => null);
// ^?
```

Aside: TwoSlash queries

```
1  const myArr = [1, 2, 3, 4, 5];
2
3  myArr.map((item) => null);
4  // ^? const myArr: number[]
```

Aside: TwoSlash queries

```
1  const myArr = [1, 2, 3, 4, 5];
2
3  myArr.map((item) => null);
4  // ^? const myArr: number[]
```

Works in the playground at typescriptlang.org as well as in VS Code with the plugin `vscode-twoslash-queries`.

Back to the array example

Built in functions

- Autocomplete
- Documentation
- Prevent accidents

```
const myArr = [1, 2, 3, 4, 5];

// Autocompletion for array
myArr.map(num => {
    num.toExponential(); // Autocompletion for number
})

myArr.stringify();
// ! Property 'stringify' does not exist on type 'number[]'.

myArr();
// ! const myArr: number[].
```

Adding your own types

Let’s extend the built-in functionality

```
interface User {
  id: string;
  name: string;
  numberOfPosts: number;
}

const myFunction = (input: User) => {
  console.log(input.numberOfPosts)
}

myFunction({ id: '1', name: 'Per' })
// ^^^ THIS WILL ERROR
// Property 'numberOfPosts' is missing in type
```

Adding your own types

Let’s extend the built-in functionality

```
interface User {
  id: string;
  name: string;
  numberOfPosts: number;
}

const myFunction = (input: User) => {
  console.log(input.numberOfPosts)
}

myFunction({ id: '1', name: 'Per' })
// ^^^ THIS WILL ERROR
// Property 'numberOfPosts' is missing in type
```

Adding your own types

Let’s extend the built-in functionality

```
interface User {
  id: string;
  name: string;
  numberOfPosts: number;
}

const myFunction = (input: User) => {
  console.log(input.numberOfPosts)
}

myFunction({ id: '1', name: 'Per' })
// ^^^ THIS WILL ERROR
// Property 'numberOfPosts' is missing in type
```

Adding your own types

Let’s extend the built-in functionality

```
interface User {
  id: string;
  name: string;
  numberOfPosts: number;
}

const myFunction = (input: User) => {
  console.log(input.numberOfPosts)
}

myFunction({ id: '1', name: 'Per' })
// ^^^ THIS WILL ERROR
// Property 'numberOfPosts' is missing in type
```

Recap on *why*

Recap on *why*

- Autocomplete in your editor

Recap on *why*

- Autocomplete in your editor
- Documentation for standard methods

Recap on *why*

- Autocomplete in your editor
- Documentation for standard methods
- Catch errors when developing instead of shipping to users

Recap on *why*

- Autocomplete in your editor
- Documentation for standard methods
- Catch errors when developing instead of shipping to users
- Covers edge-cases automatically

Recap on *why*

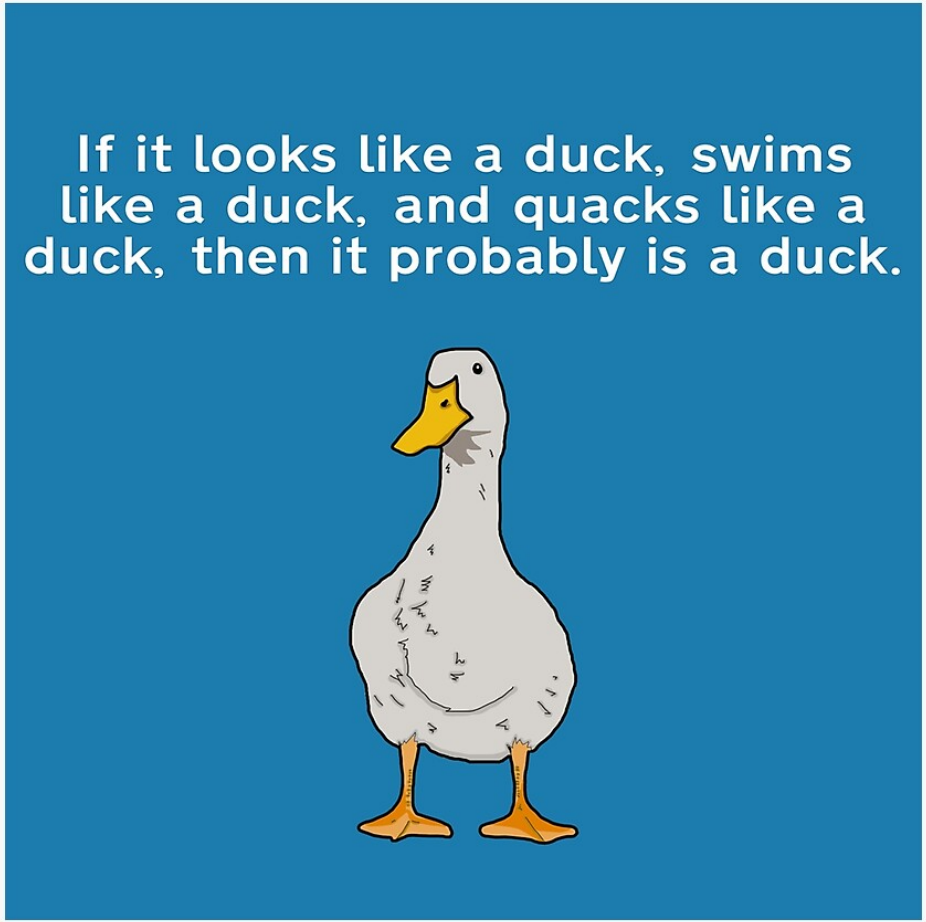
- Autocomplete in your editor
- Documentation for standard methods
- Catch errors when developing instead of shipping to users
- Covers edge-cases automatically
- Reduces hopping between files to check what the function you just wrote returns

Time to dive into *how*

Duck typing

Also called *structural typing*

Typescript is meant to prevent runtime errors, if you pass in extra properties in an object to a function, Javascript will not care. And neither does Typescript.

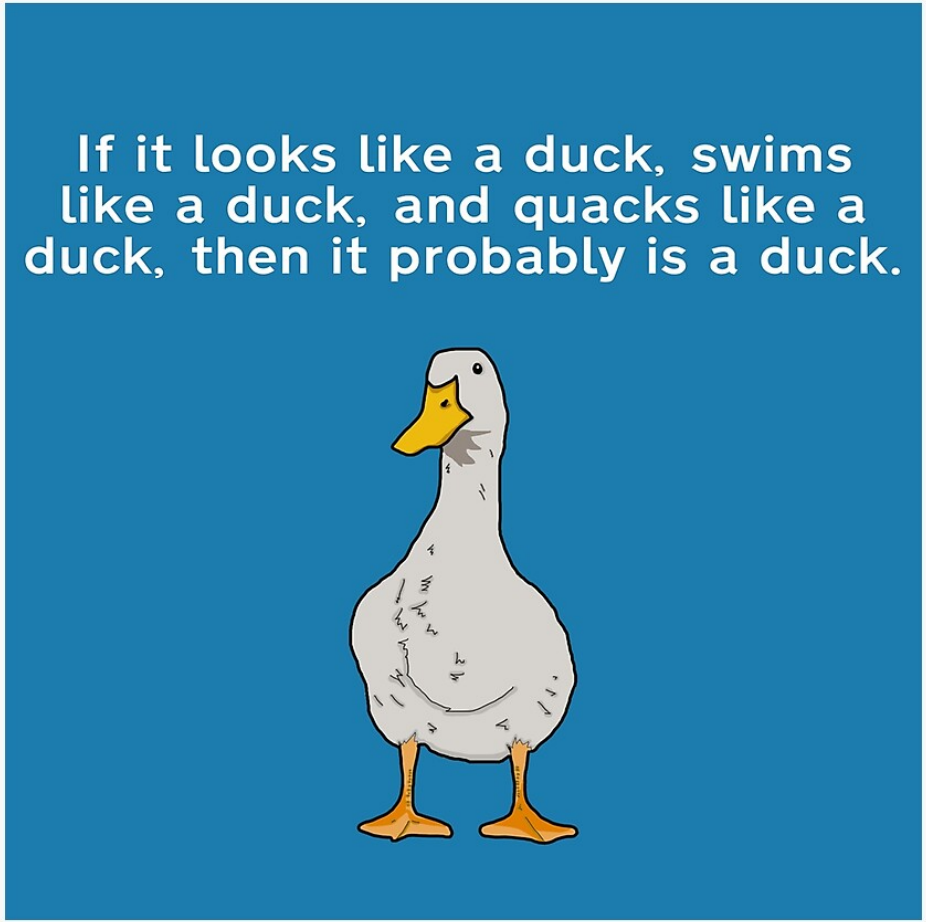


Duck typing

Also called *structural typing*

Typescript is meant to prevent runtime errors, if you pass in extra properties in an object to a function, Javascript will not care. And neither does Typescript.

```
1   interface User {
2     name: string;
3   }
4
5   const myFunction = (user: User) => {
6     doSomething(user.name);
7   }
8
9   myFunction({
10    name: 'Per',
11    address: 'Home'
12  })
```



Duck typing

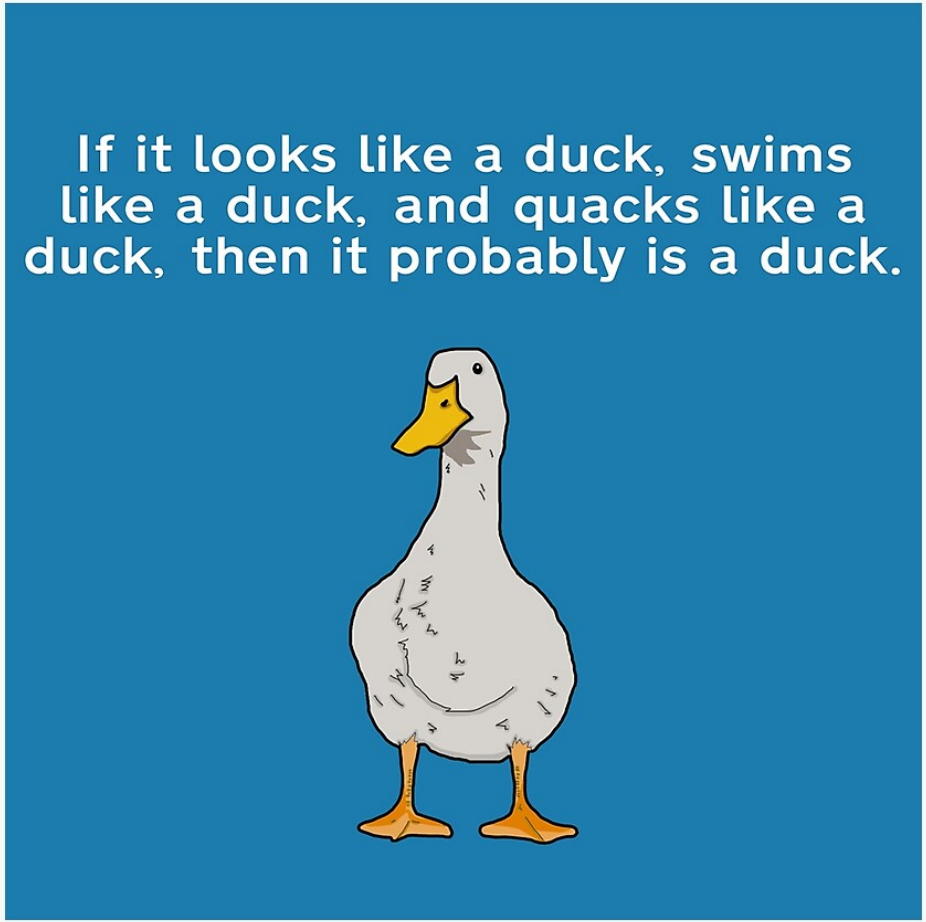
Also called *structural typing*

Typescript is meant to prevent runtime errors, if you pass in extra properties in an object to a function, Javascript will not care. And neither does Typescript.

```
interface User {
  name: string;
}

const myFunction = (user: User) => {
  doSomething(user.name);
}

myFunction({
  name: 'Per',
  address: 'Home'
})
```

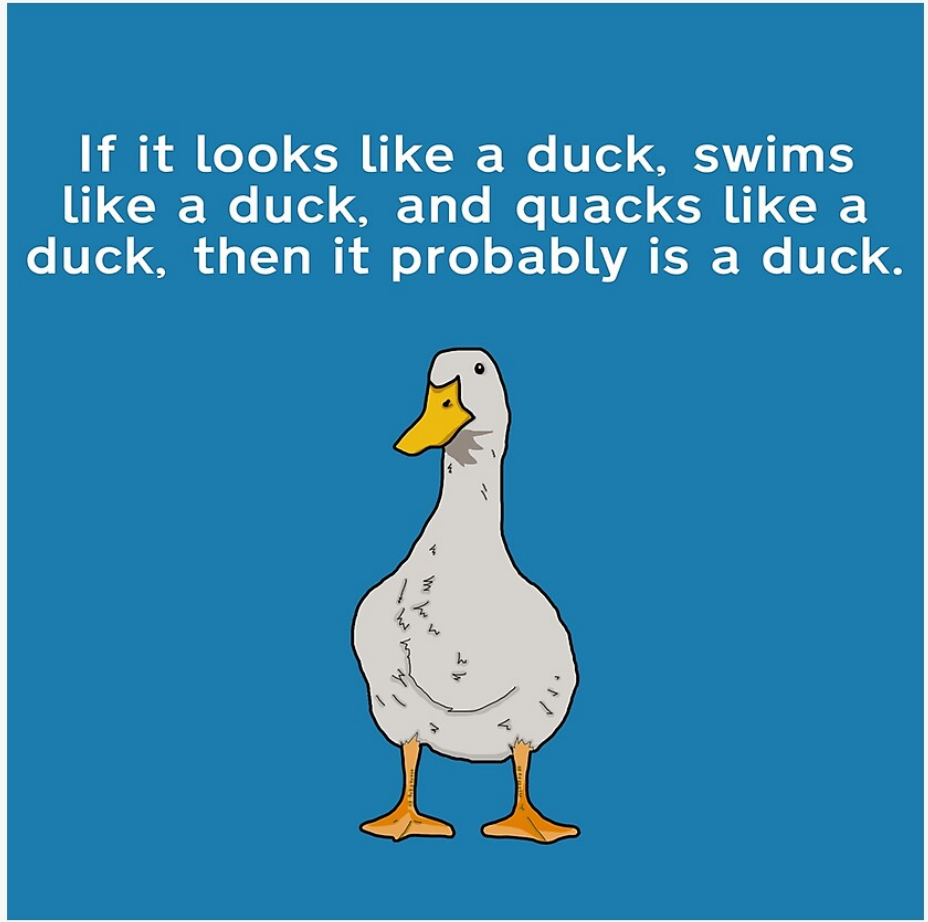


Duck typing

Also called *structural typing*

Typescript is meant to prevent runtime errors, if you pass in extra properties in an object to a function, Javascript will not care. And neither does Typescript.

```
1   interface User {
2     name: string;
3   }
4
5   const myFunction = (user: User) => {
6     doSomething(user.name);
7   }
8
9   myFunction({
10    name: 'Per',
11    address: 'Home'
12  })
```

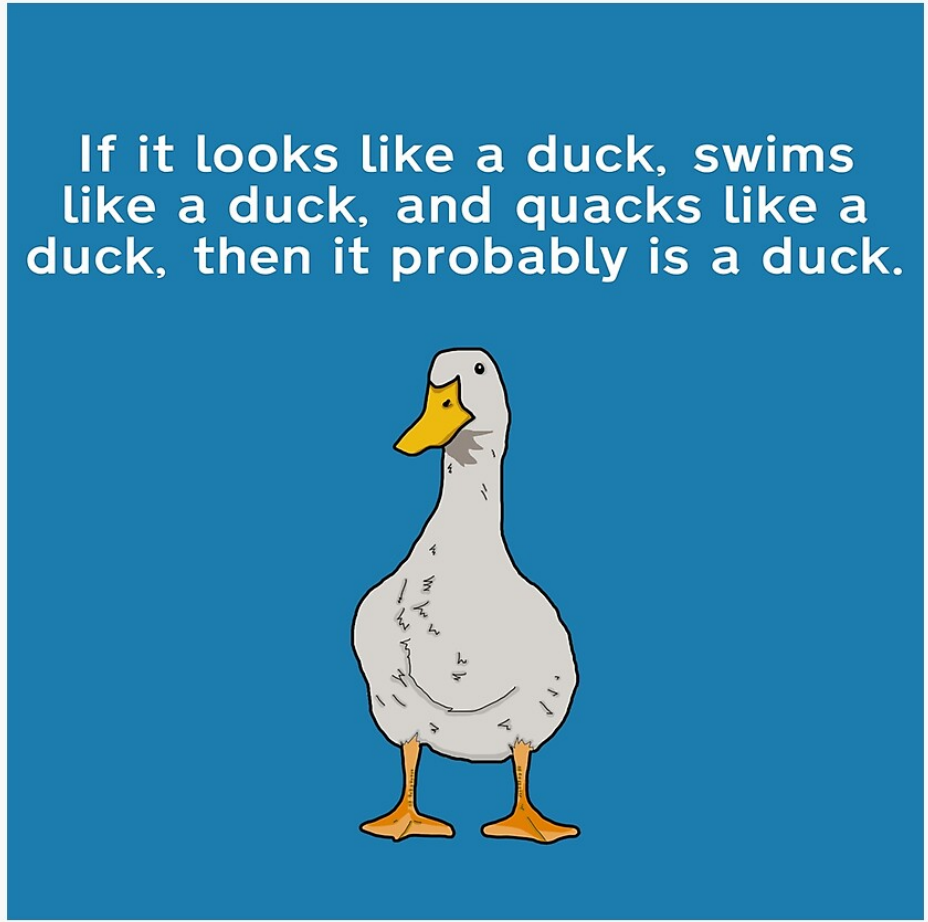


Duck typing

Also called *structural typing*

Typescript is meant to prevent runtime errors, if you pass in extra properties in an object to a function, Javascript will not care. And neither does Typescript.

```
1   interface User {
2     name: string;
3   }
4
5   const myFunction = (user: User) => {
6     doSomething(user.name);
7   }
8
9   myFunction({
10    name: 'Per',
11    address: 'Home'
12  })
```



Duck typing

Also called *structural typing*

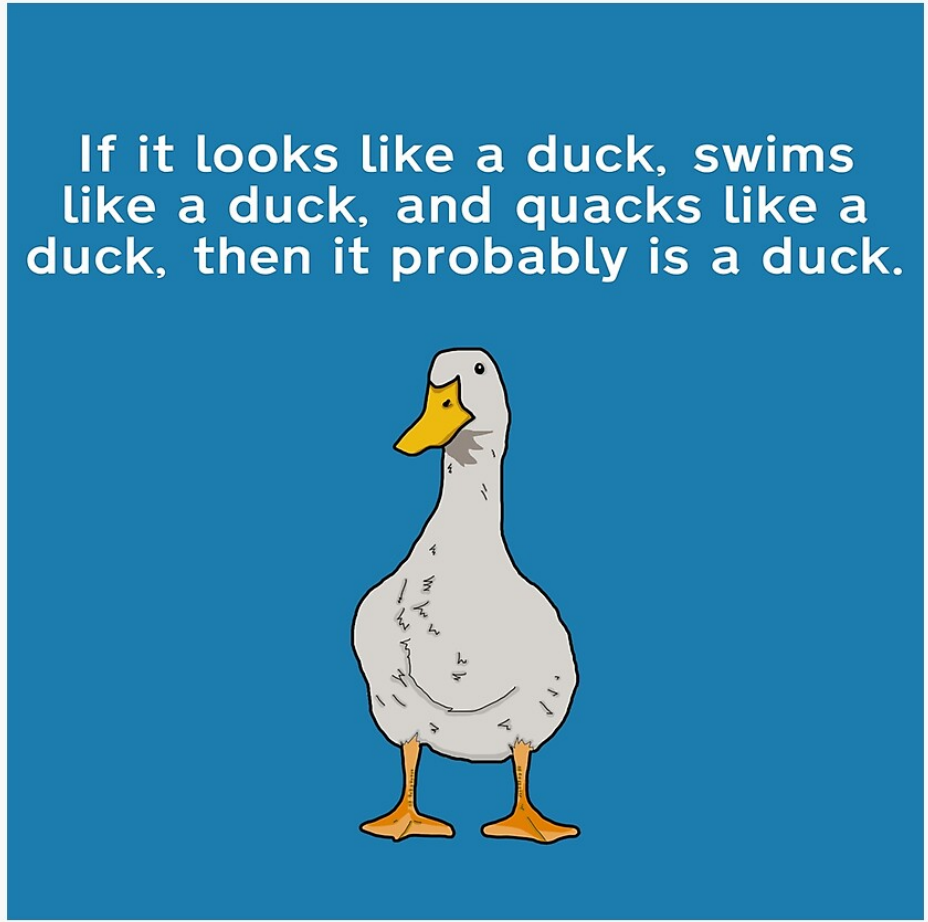
Means you sometimes have to rethink your code.

And this is a *good* thing.

```ts
interface User {
  name: string;
}

const myFunction = (user: User) => {
  callExternalApi(user);
}

myFunction({
  name: 'Per',
  address: 'Home'
})
// Might lead to unknown
// errors in external API
```



You don’t need to – and shouldn’t – type everything

Inferring types

Typescript is great at figuring out what types are.

```
1  const myArr = [1, 2, 3, 4, 5];
2  //     ^? const myArr: number[]
3
4  const concatenate = (numbers: number[]) => (
5    numbers.join('-')
6  );
7
8  const result = concatenate(myArr);
9  //     ^? const result: string
```

You don’t need to – and shouldn’t – type everything

Inferring types

Typescript is great at figuring out what types are.

```
1  const myArr = [1, 2, 3, 4, 5];
2  //     ^? const myArr: number[]
3
4  const concatenate = (numbers: number[]) => (
5    numbers.join('-')
6  );
7
8  const result = concatenate(myArr);
9  //     ^? const result: string
```

We do however need to type the input `numbers` to the function in this case

Enough talk, let’s get to some syntax

Specifying types

```
const myVar: number = 6; // Note, no need to specify type here, will be inferred

const myObject: MyObjectType = {
  property1: 'hello'
}

const myFunction = (input: MyType): MyReturnType => {
  // do stuff
  // return something of type MyReturnType
}
```

Enough talk, let's get to some syntax

Specifying types

```
const myVar: number = 6; // Note, no need to specify type here, will be inferred

const myObject: MyObjectType = {
  property1: 'hello'
}

const myFunction = (input: MyType): MyReturnType => {
  // do stuff
  // return something of type MyReturnType
}
```

Enough talk, let's get to some syntax

Specifying types

```
const myVar: number = 6; // Note, no need to specify type here, will be inferred

const myObject: MyObjectType = {
  property1: 'hello'
}

const myFunction = (input: MyType): MyReturnType => {
  // do stuff
  // return something of type MyReturnType
}
```

Typing functions

```
const myFunction = (input: MyType): MyReturnType => {
  // do stuff
  // return something of type MyReturnType
}

type MyFunctionType = (input: MyType) => MyReturnType;
const myFunction2: MyFunctionType = (input) => {
  // do stuff
  // return something of type MyReturnType
}
```

Typing functions

```
const myFunction = (input: MyType): MyReturnType => {
  // do stuff
  // return something of type MyReturnType
}

type MyFunctionType = (input: MyType) => MyReturnType;
const myFunction2: MyFunctionType = (input) => {
  // do stuff
  // return something of type MyReturnType
}
```

Typing functions

```
const myFunction = (input: MyType): MyReturnType => {
  // do stuff
  // return something of type MyReturnType
}

type MyFunctionType = (input: MyType) => MyReturnType;
const myFunction2: MyFunctionType = (input) => {
  // do stuff
  // return something of type MyReturnType
}
```

Basic types

Primitives

```
number
string
boolean
```

Arrays and tuples

```
const myArray: number[] = [ 1, 2, 3 ];
const myTuple: [number, number, number] = [ 1, 2, 3 ];
```

Basic types

Primitives

```
number
string
boolean
```

Arrays and tuples

```
const myArray: number[] = [ 1, 2, 3 ];
const myTuple: [number, number, number] = [ 1, 2, 3 ];
```

```
const myTuple = [ 1, 2, 3 ] as const;
//     ^? const myTuple: readonly [1, 2, 3]
```

Basic types

Primitives

```
number
string
boolean
```

Arrays and tuples

```
const myArray: number[] = [ 1, 2, 3 ];
const myTuple: [number, number, number] = [ 1, 2, 3 ];
```

```
const myTuple = [ 1, 2, 3 ] as const;
//     ^? const myTuple: readonly [1, 2, 3]
```

Objects

```
interface MyObjectType {
  property1: string;
  optionalProperty?: number;
}

type MyObjectType2 = {
  property1: string;
  optionalProperty?: number;
}

const myObject: MyObjectType = {
  // ...
}
```

Objects

```
interface MyObjectType {
  property1: string;
  optionalProperty?: number;
}

type MyObjectType2 = {
  property1: string;
  optionalProperty?: number;
}

const myObject: MyObjectType = {
  // ...
}
```

Objects

```
interface MyObjectType {
  property1: string;
  optionalProperty?: number;
}

type MyObjectType2 = {
  property1: string;
  optionalProperty?: number;
}

const myObject: MyObjectType = {
  // ...
}
```

Objects

```
interface MyObjectType {
  property1: string;
  optionalProperty?: number;
}

type MyObjectType2 = {
  property1: string;
  optionalProperty?: number;
}

const myObject: MyObjectType = {
  // ...
}
```

Objects

```
interface MyObjectType {
  property1: string;
  optionalProperty?: number;
}

type MyObjectType2 = {
  property1: string;
  optionalProperty?: number;
}

const myObject: MyObjectType = {
  // ...
}
```

Objects

```
interface MyObjectType {
  property1: string;
  optionalProperty?: number;
}

type MyObjectType2 = {
  property1: string;
  optionalProperty?: number;
}

const myObject: MyObjectType = {
  // ...
}
```

What's the difference you might ask?

Basically nothing, I tend to use `interface`;

Union types

Specify multiple types

```
interface User {
  id: string | number;
  name: string;
}

const myArray = [4, "hello"];
//     ^? const myArray: (string | number)[]
```

Union types

Specify multiple types

```ts
interface User {
  id: string | number;
  name: string;
}

const myArray = [4, "hello"];
//     ^? const myArray: (string | number)[]
```

Subtle but different:

```ts
type MyMixedArrayType1 = number[] | string[];
type MyMixedArrayType2 = (number | string)[];

const myArray1: MyMixedArrayType1 = [1, 2, 3, 4];
const myArray2: MyMixedArrayType1 = ["a", "b", "c", "d"];
const myArray3: MyMixedArrayType1 = [1, "b", 3, "d"]; // ❗Error

const myArray4: MyMixedArrayType2 = [1, 2, 3, 4];
const myArray5: MyMixedArrayType2 = ["a", "b", "c", "d"];
const myArray6: MyMixedArrayType2 = [1, "b", 3, "d"]; // ✅
```

Union types

Specify multiple types

```ts
interface User {
  id: string | number;
  name: string;
}

const myArray = [4, "hello"];
//     ^? const myArray: (string | number)[]
```

Subtle but different:

```ts
type MyMixedArrayType1 = number[] | string[];
type MyMixedArrayType2 = (number | string)[];

const myArray1: MyMixedArrayType1 = [1, 2, 3, 4];
const myArray2: MyMixedArrayType1 = ["a", "b", "c", "d"];
const myArray3: MyMixedArrayType1 = [1, "b", 3, "d"]; // ❗Error

const myArray4: MyMixedArrayType2 = [1, 2, 3, 4];
const myArray5: MyMixedArrayType2 = ["a", "b", "c", "d"];
const myArray6: MyMixedArrayType2 = [1, "b", 3, "d"]; // ✅
```

Union types

Specify multiple types

```ts
interface User {
  id: string | number;
  name: string;
}

const myArray = [4, "hello"];
//     ^? const myArray: (string | number)[]
```

Subtle but different:

```ts
type MyMixedArrayType1 = number[] | string[];
type MyMixedArrayType2 = (number | string)[];

const myArray1: MyMixedArrayType1 = [1, 2, 3, 4];
const myArray2: MyMixedArrayType1 = ["a", "b", "c", "d"];
const myArray3: MyMixedArrayType1 = [1, "b", 3, "d"]; // ❗Error

const myArray4: MyMixedArrayType2 = [1, 2, 3, 4];
const myArray5: MyMixedArrayType2 = ["a", "b", "c", "d"];
const myArray6: MyMixedArrayType2 = [1, "b", 3, "d"]; // ✅
```

Union types

Specify multiple types

```ts
interface User {
  id: string | number;
  name: string;
}

const myArray = [4, "hello"];
//     ^? const myArray: (string | number)[]
```

Subtle but different:

```ts
type MyMixedArrayType1 = number[] | string[];
type MyMixedArrayType2 = (number | string)[];

const myArray1: MyMixedArrayType1 = [1, 2, 3, 4];
const myArray2: MyMixedArrayType1 = ["a", "b", "c", "d"];
const myArray3: MyMixedArrayType1 = [1, "b", 3, "d"]; // ❗Error

const myArray4: MyMixedArrayType2 = [1, 2, 3, 4];
const myArray5: MyMixedArrayType2 = ["a", "b", "c", "d"];
const myArray6: MyMixedArrayType2 = [1, "b", 3, "d"]; // ✅
```

Literal types

Constrain stuff!

When only certain values are valid.

```
interface User {
  id: string | number;
  name: string;
  state: "subscribed" | "inactive";
}

const userPer: User = {
  id: 1,
  name: "Per",
  state: "inactive",
};

userPer.state = "subscribed"; // Autocomplete for both property and state

userPer.state = "inatcive";
// ! Type '"inatcive"' is not assignable to type '"subscribed" | "inactive"'. Did you mean '"inactive"'?
```

Literal types

Constrain stuff!

When only certain values are valid.

What about enums?

Short answer, they don’t behave as you might expect. My recommendation is to stay away from them.

For a more in depth explanation, look at the video *Enums considered harmful* by *Matt Pocock* on Youtube.

Type Narrowing

A way to make a broad type more specific.

Even more restricted type

```ts
interface User {
  id: string | number;
  name: string;
  type: "user";
}

interface Company {
  id: string;
  legalName: string;
  publicName: string;
  type: "company";
}

type Member = User | Company;
```

```ts
const printName = (member: Member) => {
  console.log(member.name);
  // ! Error: Property 'name' does not
  //          exist on type 'Company'.
}

const printName2 = (member: Member) => {
  if (member.type === "user") {
    console.log(member.name);
    //          ^? member: User
  }

  else {
    console.log(member.publicName);
    //          ^? member: Company
  }
};
```

Type Narrowing

A way to make a broad type more specific.

Even more restricted type

```ts
interface User {
  id: string | number;
  name: string;
  type: "user";
}

interface Company {
  id: string;
  legalName: string;
  publicName: string;
  type: "company";
}

type Member = User | Company;
```

```ts
const printName = (member: Member) => {
  console.log(member.name);
  // ! Error: Property 'name' does not
  //          exist on type 'Company'.
}

const printName2 = (member: Member) => {
  if (member.type === "user") {
    console.log(member.name);
    //            ^? member: User
  }

  else {
    console.log(member.publicName);
    //            ^? member: Company
  }
};
```

Type Narrowing

A way to make a broad type more specific.

Even more restricted type

```ts
interface User {
  id: string | number;
  name: string;
  type: "user";
}

interface Company {
  id: string;
  legalName: string;
  publicName: string;
  type: "company";
}

type Member = User | Company;
```

```ts
const printName = (member: Member) => {
  console.log(member.name);
  // ! Error: Property 'name' does not
  //          exist on type 'Company'.
}

const printName2 = (member: Member) => {
  if (member.type === "user") {
    console.log(member.name);
    //            ^? member: User
  }

  else {
    console.log(member.publicName);
    //            ^? member: Company
  }
};
```

Type Narrowing

A way to make a broad type more specific.

Even more restricted type

```ts
interface User {
  id: string | number;
  name: string;
  type: "user";
}

interface Company {
  id: string;
  legalName: string;
  publicName: string;
  type: "company";
}

type Member = User | Company;
```

```ts
const printName = (member: Member) => {
  console.log(member.name);
  // ! Error: Property 'name' does not
  //          exist on type 'Company'.
}

const printName2 = (member: Member) => {
  if (member.type === "user") {
    console.log(member.name);
    //          ^? member: User
  }

  else {
    console.log(member.publicName);
    //          ^? member: Company
  }
};
```

Type Narrowing

A way to make a broad type more specific.

Even more restricted type

```ts
interface User {
  id: string | number;
  name: string;
  type: "user";
}

interface Company {
  id: string;
  legalName: string;
  publicName: string;
  type: "company";
}

type Member = User | Company;
```

```ts
const printName = (member: Member) => {
  console.log(member.name);
  // ! Error: Property 'name' does not
  //          exist on type 'Company'.
}

const printName2 = (member: Member) => {
  if (member.type === "user") {
    console.log(member.name);
    //            ^? member: User
  }

  else {
    console.log(member.publicName);
    //            ^? member: Company
  }
};
```

Type Narrowing

A way to make a broad type more specific.

Even more restricted type

```ts
interface User {
  id: string | number;
  name: string;
  type: "user";
}

interface Company {
  id: string;
  legalName: string;
  publicName: string;
  type: "company";
}

type Member = User | Company;
```

```ts
const printName = (member: Member) => {
  console.log(member.name);
  // ! Error: Property 'name' does not
  //          exist on type 'Company'.
}

const printName2 = (member: Member) => {
  if (member.type === "user") {
    console.log(member.name);
    //            ^? member: User
  }

  else {
    console.log(member.publicName);
    //            ^? member: Company
  }
};
```

Type Narrowing

A way to make a broad type more specific.

Even more restricted type

```ts
interface User {
  id: string | number;
  name: string;
  type: "user";
}

interface Company {
  id: string;
  legalName: string;
  publicName: string;
  type: "company";
}

type Member = User | Company;
```

```ts
const printName = (member: Member) => {
  console.log(member.name);
  // ! Error: Property 'name' does not
  //          exist on type 'Company'.
}

const printName2 = (member: Member) => {
  if (member.type === "user") {
    console.log(member.name);
    //           ^? member: User
  }

  else {
    console.log(member.publicName);
    //           ^? member: Company
  }
};
```

Escape hatches

If your ship is sinking, use these as a last resort

- Any
- as Type

The `any` type

Turns off Typescript's checks completely

- Often used in frustration when Typescript is giving developers hard to understand error messages.
- There's *always* a better option
  - Fix your upstream types to be what you actually mean
  - Generics
  - The `unknown` type

```
1   const myUnsafeFunction = (input: any) => {
2     const result: number = input + 6;
3
4     return result;
5   }
6
7   const result = myUnsafeFunction(5) // returns 11
8   const result2 = myUnsafeFunction(true) // returns 7
9
10  const result3 = myUnsafeFunction({prop: "bar"}) // ! Will crash
```

Type declarations with `as Type`

When you *actually* know better than Typescript.

```
const getData = async () => {
  const result = await fetch('https://my.external.api');
  const data = await result.json();
  //     ^? const data: any

  return data as UserDetails
}

const myMap = new Map<string, number>();
const iterator = myMap.entries();

const value = iterator.next().value() as [string, number];
```

Type declarations with `as Type`

When you *actually* know better than Typescript.

```
const getData = async () => {
  const result = await fetch('https://my.external.api');
  const data = await result.json();
  //     ^? const data: any

  return data as UserDetails
}

const myMap = new Map<string, number>();
const iterator = myMap.entries();

const value = iterator.next().value() as [string, number];
```

Resources

- Typescript Handbook – typescriptlang.org/docs/handbook/intro.html
- Matt Pocock – www.youtube.com/@mattpocockuk
- These slides (made with slidev) – per.fyi/talks