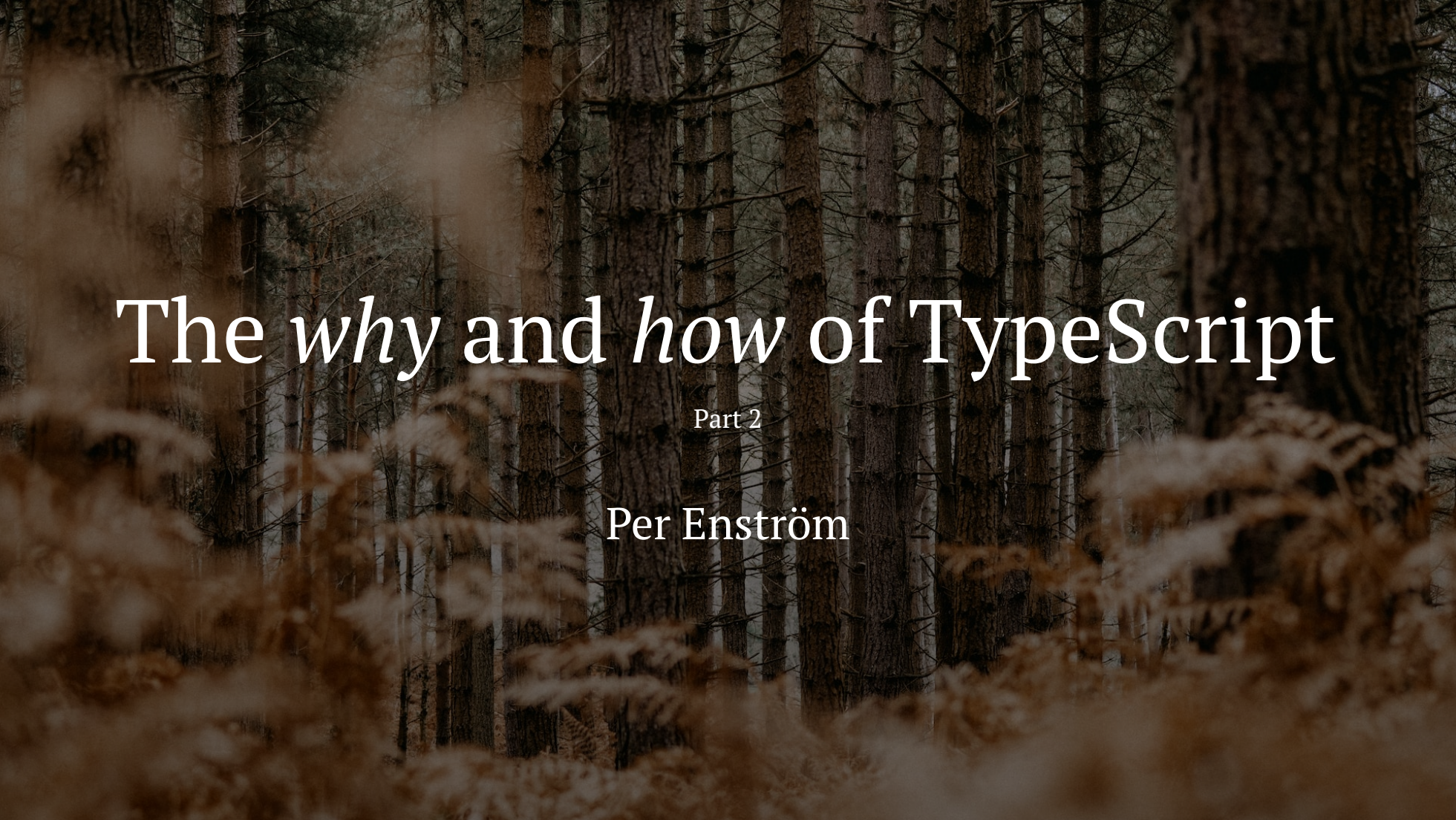


The *why* and *how* of TypeScript

Part 2

Per Enström

Recap

Typing stuff

```
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
}
```

You don't *need* to type everything. Inferring types based on usage is really strong.

Recap

Typing stuff

```
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
}
```

You don't *need* to type everything. Inferring types based on usage is really strong.

Recap

Typing stuff

```
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
}
```

You don’t *need* to type everything. Inferring types based on usage is really strong.

Recap

Typing stuff

```
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
}
```

You don't *need* to type everything. Inferring types based on usage is really strong.

Recap

Discriminated unions

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

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

type Member = User | Company;
```

Recap

Discriminated unions

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

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

type Member = User | Company;
```

Recap

Discriminated unions

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

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

type Member = User | Company;
```

Recap

Discriminated unions

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

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

type Member = User | Company;
```

Recap

Discriminated unions

"Can’t you just switch on the Type without having a literal property?"

Recap

Discriminated unions

"Can’t you just switch on the Type without having a literal property?"

```
const printName = (member: Member) => {
  if (typeof member === "User") {
    console.log(member.name);
    //           ^? member: User
  }

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

Recap

Discriminated unions

"Can’t you just switch on the Type without having a literal property?"

```
const printName = (member: Member) => {
  if (typeof member === "User") {
    console.log(member.name);
    //           ^? member: User
  }

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

No.

Typescript needs to be able to transpile down to plain Javascript, and there is no way of doing that in this case.

Type predicates

How would you do in javascript?

Type predicates

How would you do in javascript?

```
const myEvent = {
  eventType: 'payment',
  data: {
    payee: '07011111111'
  }
}

const myPayment = {
  amount: 123,
  message: 'Hello'
}
```

Type predicates

How would you do in javascript?

```
const myEvent = {
  eventType: 'payment',
  data: {
    payee: '07011111111'
  }
}

const myPayment = {
  amount: 123,
  message: 'Hello'
}
```

```
const isEvent = (eventOrPayment) => {
  // If key eventType exists in object eventOrPayment
  return "eventType" in eventOrPayment;
};

const doSomething = (eventOrPayment) => {
  if(isEvent(myEvent)){
    console.log(myEvent.data);
  }
}
```

Type predicates

How do you do in Typescript?

```ts
interface EventType {
  eventType: string;
  data: { payee: string; };
}

const myEvent: EventType = {
  eventType: "payment",
  data: { payee: "07011111111" }
};

interface PaymentType {
  amount: number;
  message: string;
}

const myPayment: PaymentType = {
  amount: 123,
  message: 'Hello'
}
```

```ts
const isEvent =
  (e: EventType | PaymentType) => {
    // if key eventType exists in object e
    return "eventType" in e;
  };

const doStuff = (e: EventType | PaymentType) => {
  if(isEvent(e)){
    console.log(e.data); // ! Error
    //          ^? const e: EventType | PaymentType
  }
}
```

Type predicates

How do you do in Typescript?

```typescript
interface EventType {
  eventType: string;
  data: { payee: string; };
}

const myEvent: EventType = {
  eventType: "payment",
  data: { payee: "07011111111" }
};

interface PaymentType {
  amount: number;
  message: string;
}

const myPayment: PaymentType = {
  amount: 123,
  message: 'Hello'
}
```

```typescript
const isEvent =
  (e: EventType | PaymentType) => {
    // if key eventType exists in object e
    return "eventType" in e;
  };

const doStuff = (e: EventType | PaymentType) => {
  if(isEvent(e)){
    console.log(e.data); // ! Error
    //          ^? const e: EventType | PaymentType
  }
}
```

Type predicates

How do you do in Typescript?

```ts
interface EventType {
  eventType: string;
  data: { payee: string; };
}

const myEvent: EventType = {
  eventType: "payment",
  data: { payee: "07011111111" }
};

interface PaymentType {
  amount: number;
  message: string;
}

const myPayment: PaymentType = {
  amount: 123,
  message: 'Hello'
}
```

```ts
const isEvent =
  (e: EventType | PaymentType) => {
    // if key eventType exists in object e
    return "eventType" in e;
  };

const doStuff = (e: EventType | PaymentType) => {
  if(isEvent(e)){
    console.log(e.data); // ! Error
    //          ^? const e: EventType | PaymentType
  }
}
```

Type predicates

How do you do in Typescript?

```ts
interface EventType {
  eventType: string;
  data: { payee: string; };
}

const myEvent: EventType = {
  eventType: "payment",
  data: { payee: "07011111111" }
};

interface PaymentType {
  amount: number;
  message: string;
}

const myPayment: PaymentType = {
  amount: 123,
  message: 'Hello'
}
```

```ts
const isEvent =
  (e: EventType | PaymentType) => {
    // if key eventType exists in object e
    return "eventType" in e;
  };

const doStuff = (e: EventType | PaymentType) => {
  if(isEvent(e)){
    console.log(e.data); // ! Error
    //          ^? const e: EventType | PaymentType
  }
}
```

Type predicates

How do you do in Typescript?

```typescript
interface EventType {
  eventType: string;
  data: { payee: string; };
}

const myEvent: EventType = {
  eventType: "payment",
  data: { payee: "07011111111" }
};

interface PaymentType {
  amount: number;
  message: string;
}

const myPayment: PaymentType = {
  amount: 123,
  message: 'Hello'
}
```

```typescript
const isEvent =
  (e: EventType | PaymentType) => {
    // if key eventType exists in object e
    return "eventType" in e;
  };

const doStuff = (e: EventType | PaymentType) => {
  if(isEvent(e)){
    console.log(e.data); // ! Error
    //          ^? const e: EventType | PaymentType
  }
}
```

Type predicates

How do you do in Typescript?

```typescript
interface EventType {
  eventType: string;
  data: { payee: string; };
}

const myEvent: EventType = {
  eventType: "payment",
  data: { payee: "07011111111" }
};

interface PaymentType {
  amount: number;
  message: string;
}

const myPayment: PaymentType = {
  amount: 123,
  message: 'Hello'
}
```

```typescript
const isEvent =
  (e: EventType | PaymentType) => {
    // if key eventType exists in object e
    return "eventType" in e;
  };

const doStuff = (e: EventType | PaymentType) => {
  if(isEvent(e)){
    console.log(e.data); // ! Error
    //          ^? const e: EventType | PaymentType
  }
}
```

Type predicates

How do you do in Typescript?

```typescript
interface EventType {
  eventType: string;
  data: { payee: string; };
}

const myEvent: EventType = {
  eventType: "payment",
  data: { payee: "07011111111" }
};

interface PaymentType {
  amount: number;
  message: string;
}

const myPayment: PaymentType = {
  amount: 123,
  message: 'Hello'
}
```

```typescript
const isEvent =
  (e: EventType | PaymentType) => {
    // if key eventType exists in object e
    return "eventType" in e;
  };

const doStuff = (e: EventType | PaymentType) => {
  if(isEvent(e)){
    console.log(e.data); // ! Error
    //          ^? const e: EventType | PaymentType
  }
}
```

Type predicates

How do you do in Typescript?

```ts
interface EventType {
  eventType: string;
  data: { payee: string; };
}

const myEvent: EventType = {
  eventType: "payment",
  data: { payee: "07011111111" }
};

interface PaymentType {
  amount: number;
  message: string;
}

const myPayment: PaymentType = {
  amount: 123,
  message: 'Hello'
}
```

```ts
const isEvent =
  (e: EventType | PaymentType) => {
    // if key eventType exists in object e
    return "eventType" in e;
  };

const doStuff = (e: EventType | PaymentType) => {
  if(isEvent(e)){
    console.log(e.data); // ! Error
    //          ^? const e: EventType | PaymentType
  }
}
```

Type predicates

How do you do in Typescript?

```typescript
interface EventType {
  eventType: string;
  data: { payee: string; };
}

const myEvent: EventType = {
  eventType: "payment",
  data: { payee: "07011111111" }
};

interface PaymentType {
  amount: number;
  message: string;
}

const myPayment: PaymentType = {
  amount: 123,
  message: 'Hello'
}
```

```typescript
const isEvent =
  (e: EventType | PaymentType): e is MyEvent => {
    // if key eventType exists in object e
    return "eventType" in e;
  };

const doStuff = (e: EventType | PaymentType) => {
  if(isEvent(e)){
    console.log(e.data);
    //          ^? const e: EventType
  }
}
```

Type predicates

How do you do in Typescript?

```typescript
interface EventType {
  eventType: string;
  data: { payee: string; };
}

const myEvent: EventType = {
  eventType: "payment",
  data: { payee: "07011111111" }
};

interface PaymentType {
  amount: number;
  message: string;
}

const myPayment: PaymentType = {
  amount: 123,
  message: 'Hello'
}
```

```typescript
const isEvent =
  (e: EventType | PaymentType): e is MyEvent => {
    // if key eventType exists in object e
    return "eventType" in e;
  };

const doStuff = (e: EventType | PaymentType) => {
  if(isEvent(e)){
    console.log(e.data);
    //          ^? const e: EventType
  }
}
```

Hard to read types, how to parse?

The following is what you get when hovering over an array map function:

```
(method) Array<number>.map<number>(callbackfn: (value: number, index: number, array: number[]) => number, thisArg?
```

Hard to read types, how to parse?

The following is what you get when hovering over an array map function:

```
(method) Array<number>.map<number>(callbackfn: (value: number, index: number, array: number[]) => number, thisArg?
```

Let's break it down

```
// Remove category (only a visual representation)
Array<number>.map<number>(callbackfn: (value: number, index: number, array: number[]) =>
  number, thisArg?: any): number[]
// Remove generics for now (assume an array of numbers, and that map returns a number)
Array.map(callbackfn: (value: number, index: number, array: number[]) => number, thisArg?: any): number[]
// Move types to separate definitions
type CallBackFn = (
  value: number,
  index: number,
  array: number[]
) => number

Array.map(callbackfn: CallBackFn, thisArg?: any): number[]
```

Hard to read types, how to parse?

The following is what you get when hovering over an array map function:

```
(method) Array<number>.map<number>(callbackfn: (value: number, index: number, array: number[]) => number, thisArg?
```

Let's break it down

```
// Remove category (only a visual representation)
Array<number>.map<number>(callbackfn: (value: number, index: number, array: number[]) =>
  number, thisArg?: any): number[]
// Remove generics for now (assume an array of numbers, and that map returns a number)
Array.map(callbackfn: (value: number, index: number, array: number[]) => number, thisArg?: any): number[]
// Move types to separate definitions
type CallBackFn = (
  value: number,
  index: number,
  array: number[]
) => number

Array.map(callbackfn: CallBackFn, thisArg?: any): number[]
```

Hard to read types, how to parse?

The following is what you get when hovering over an array map function:

```
(method) Array<number>.map<number>(callbackfn: (value: number, index: number, array: number[]) => number, thisArg?
```

Let's break it down

```
// Remove category (only a visual representation)
Array<number>.map<number>(callbackfn: (value: number, index: number, array: number[]) =>
  number, thisArg?: any): number[]
// Remove generics for now (assume an array of numbers, and that map returns a number)
Array.map(callbackfn: (value: number, index: number, array: number[]) => number, thisArg?: any): number[]
// Move types to separate definitions
type CallBackFn = (
  value: number,
  index: number,
  array: number[]
) => number

Array.map(callbackfn: CallBackFn, thisArg?: any): number[]
```

Generics

A way of *passing arguments* to a type

Generics

A way of *passing arguments* to a type

Passing parameters to a function

```
const myFunction = (input: number) => {
  return input + 1;
}

const addedNumber = myFunction(5);
```

Generics

A way of *passing arguments* to a type

Passing parameters to a function

```
const myFunction = (input: number) => {
  return input + 1;
}

const addedNumber = myFunction(5);
```

Passing arguments to a type

```
const myFunction = <T>(input: T) => {
  if (typeof input === "number") return input + 1;
  else return "Not a number";
};

myFunction<number>(1); // returns 2
myFunction<string>("1"); // returns "Not a number"

myFunction(true); // returns "Not a number"
// ^? const myFunction: <boolean>(input: boolean) => number | "Not a number"
```

Generics

A way of *passing arguments* to a type

Passing parameters to a function

```
const myFunction = (input: number) => {
  return input + 1;
}

const addedNumber = myFunction(5);
```

Passing arguments to a type

```
const myFunction = <T>(input: T) => {
  if (typeof input === "number") return input + 1;
  else return "Not a number";
};

myFunction<number>(1); // returns 2
myFunction<string>("1"); // returns "Not a number"

myFunction(true); // returns "Not a number"
// ^? const myFunction: <boolean>(input: boolean) => number | "Not a number"
```

Generics

A way of *passing arguments* to a type

Passing parameters to a function

```
const myFunction = (input: number) => {
  return input + 1;
}

const addedNumber = myFunction(5);
```

Passing arguments to a type

```
const myFunction = <T>(input: T) => {
  if (typeof input === "number") return input + 1;
  else return "Not a number";
};

myFunction<number>(1); // returns 2
myFunction<string>("1"); // returns "Not a number"

myFunction(true); // returns "Not a number"
// ^? const myFunction: <boolean>(input: boolean) => number | "Not a number"
```

Generics

```
interface ErrorMessage {
  code: string;
  message: string;
}

type SuccessResult<T> = {
  success: true;
  data: T;
};

type ErrorResult = {
  success: false;
  error: ErrorMessage;
};

type Maybe<T> = SuccessResult<T> | ErrorResult;
```

Generics

```
interface ErrorMessage {
  code: string;
  message: string;
}

type SuccessResult<T> = {
  success: true;
  data: T;
};

type ErrorResult = {
  success: false;
  error: ErrorMessage;
};

type Maybe<T> = SuccessResult<T> | ErrorResult;
```

Generics

```
interface ErrorMessage {
  code: string;
  message: string;
}

type SuccessResult<T> = {
  success: true;
  data: T;
};

type ErrorResult = {
  success: false;
  error: ErrorMessage;
};

type Maybe<T> = SuccessResult<T> | ErrorResult;
```

Generics

```
interface ErrorMessage {
  code: string;
  message: string;
}

type SuccessResult<T> = {
  success: true;
  data: T;
};

type ErrorResult = {
  success: false;
  error: ErrorMessage;
};

type Maybe<T> = SuccessResult<T> | ErrorResult;
```

Back to the complicated map type

This is the actual definition

```
interface Array<T> {
  map<U>(callbackfn: (value: T, index: number, array: T[]) => U, thisArg?: any): U[];
}
```

Back to the complicated map type

This is the actual definition

```
interface Array<T> {
  map<U>(callbackfn: (value: T, index: number, array: T[]) => U, thisArg?: any): U[];
}
```

Let’s break it down

```
interface Array {
  map(arguments): any[];
}

interface Array {
  map<U>(arguments): U[];
}

interface Array {
  map<U>(callbackfn: CallbackFn): U[];
}
```

Back to the complicated map type

This is the actual definition

```
interface Array<T> {
  map<U>(callbackfn: (value: T, index: number, array: T[]) => U, thisArg?: any): U[];
}
```

Let’s break it down

```
interface Array {
  map(arguments): any[];
}

interface Array {
  map<U>(arguments): U[];
}

interface Array {
  map<U>(callbackfn: CallbackFn): U[];
}
```

Back to the complicated map type

This is the actual definition

```
interface Array<T> {
  map<U>(callbackfn: (value: T, index: number, array: T[]) => U, thisArg?: any): U[];
}
```

Let’s break it down

```
interface Array {
  map(arguments): any[];
}

interface Array {
  map<U>(arguments): U[];
}

interface Array {
  map<U>(callbackfn: CallbackFn): U[];
}
```

Back to the complicated map type

This is the actual definition

```
interface Array<T> {
  map<U>(callbackfn: (value: T, index: number, array: T[]) => U, thisArg?: any): U[];
}
```

Let’s break it down

```
interface Array {
  map<U>(
    callbackfn: CallbackFn
  ): U[];
}

interface Array<T> {
  map<U>(
    callbackfn: (value: T, index: number, array: T[]) => U
  ): U[];
}
```

Back to the complicated map type

This is the actual definition

```
interface Array<T> {
  map<U>(callbackfn: (value: T, index: number, array: T[]) => U, thisArg?: any): U[];
}
```

Let’s break it down

```
interface Array {
  map<U>(
    callbackfn: CallbackFn
  ): U[];
}

interface Array<T> {
  map<U>(
    callbackfn: (value: T, index: number, array: T[]) => U
  ): U[];
}
```

Back to the complicated map type

This is the actual definition

```
interface Array<T> {
  map<U>(callbackfn: (value: T, index: number, array: T[]) => U, thisArg?: any): U[];
}
```

Let’s break it down

```
interface Array {
  map<U>(
    callbackfn: CallbackFn
  ): U[];
}

interface Array<T> {
  map<U>(
    callbackfn: (value: T, index: number, array: T[]) => U
  ): U[];
}
```

Back to the complicated map type

This is the actual definition

```
interface Array<T> {
  map<U>(callbackfn: (value: T, index: number, array: T[]) => U, thisArg?: any): U[];
}
```

Let’s break it down

```
interface Array {
  map<U>(
    callbackfn: CallbackFn
  ): U[];
}

interface Array<T> {
  map<U>(
    callbackfn: (value: T, index: number, array: T[]) => U
  ): U[];
}
```

Back to the complicated map type

This is the actual definition

```
interface Array<T> {
  map<U>(callbackfn: (value: T, index: number, array: T[]) => U, thisArg?: any): U[];
}
```

Let’s break it down

```
interface Array {
  map<U>(
    callbackfn: CallbackFn
  ): U[];
}

interface Array<T> {
  map<U>(
    callbackfn: (value: T, index: number, array: T[]) => U
  ): U[];
}
```

Error messages

We had this example before:

```
type MyMixedArrayType1 = number[] | string[];

const myArray3: MyMixedArrayType1 = [1, "b", 3, "d"]; // ! Error
```

What the hell does this mean?

```
Type '(string | number)[]' is not assignable to type 'MyMixedArrayType1'.
  Type '(string | number)[]' is not assignable to type 'number[]'.
    Type 'string | number' is not assignable to type 'number'.
      Type 'string' is not assignable to type 'number'.
```

Error messages

We had this example before:

```
type MyMixedArrayType1 = number[] | string[];

const myArray3: MyMixedArrayType1 = [1, "b", 3, "d"]; // ! Error
```

What the hell does this mean?

```
Type '(string | number)[]' is not assignable to type 'MyMixedArrayType1'.
  Type '(string | number)[]' is not assignable to type 'number[]'.
    Type 'string | number' is not assignable to type 'number'.
      Type 'string' is not assignable to type 'number'.
```

Error messages

We had this example before:

```
type MyMixedArrayType1 = number[] | string[];

const myArray3: MyMixedArrayType1 = [1, "b", 3, "d"]; // ! Error
```

What the hell does this mean?

```
Type '(string | number)[]' is not assignable to type 'MyMixedArrayType1'.
  Type '(string | number)[]' is not assignable to type 'number[]'.
    Type 'string | number' is not assignable to type 'number'.
      Type 'string' is not assignable to type 'number'.
```

Error messages

We had this example before:

```
type MyMixedArrayType1 = number[] | string[];

const myArray3: MyMixedArrayType1 = [1, "b", 3, "d"]; // ! Error
```

What the hell does this mean?

```
Type '(string | number)[]' is not assignable to type 'MyMixedArrayType1'.
  Type '(string | number)[]' is not assignable to type 'number[]'.
    Type 'string | number' is not assignable to type 'number'.
      Type 'string' is not assignable to type 'number'.
```

Error messages

We had this example before:

```
type MyMixedArrayType1 = number[] | string[];

const myArray3: MyMixedArrayType1 = [1, "b", 3, "d"]; // ! Error
```

What the hell does this mean?

```
Type '(string | number)[]' is not assignable to type 'MyMixedArrayType1'.
  Type '(string | number)[]' is not assignable to type 'number[]'.
    Type 'string | number' is not assignable to type 'number'.
      Type 'string' is not assignable to type 'number'.
```

Utility functions

Create types from other types

```ts
type Fruits = "apple" | "banana" | "mango";

type TastyFruits = Exclude<Fruits, "mango">;        // "apple" | "banana"
type ActuallyNotAFruit = Extract<Fruits, "banana">; // "mango"

interface Fruit {
  name: string;
  color: string;
  tasty: boolean;
}

type BasicFruit = Omit<Fruit, 'tasty'>; // { name: string; color: string; }
type FruitName = Pick<Fruit, 'name'>;   // { name: string; }
```

Utility functions

Create types from other types

```ts
type Fruits = "apple" | "banana" | "mango";

type TastyFruits = Exclude<Fruits, "mango">;        // "apple" | "banana"
type ActuallyNotAFruit = Extract<Fruits, "banana">; // "mango"

interface Fruit {
  name: string;
  color: string;
  tasty: boolean;
}

type BasicFruit = Omit<Fruit, 'tasty'>; // { name: string; color: string; }
type FruitName = Pick<Fruit, 'name'>;   // { name: string; }
```

Utility functions

Create types from other types

```ts
type Fruits = "apple" | "banana" | "mango";

type TastyFruits = Exclude<Fruits, "mango">;        // "apple" | "banana"
type ActuallyNotAFruit = Extract<Fruits, "banana">; // "mango"

interface Fruit {
  name: string;
  color: string;
  tasty: boolean;
}

type BasicFruit = Omit<Fruit, 'tasty'>; // { name: string; color: string; }
type FruitName = Pick<Fruit, 'name'>;   // { name: string; }
```

Utility functions

Create types from other types

```ts
type Fruits = "apple" | "banana" | "mango";

type TastyFruits = Exclude<Fruits, "mango">;        // "apple" | "banana"
type ActuallyNotAFruit = Extract<Fruits, "banana">; // "mango"

interface Fruit {
  name: string;
  color: string;
  tasty: boolean;
}

type BasicFruit = Omit<Fruit, 'tasty'>; // { name: string; color: string; }
type FruitName = Pick<Fruit, 'name'>;   // { name: string; }
```

Utility functions

Create types from other types

```ts
type Fruits = "apple" | "banana" | "mango";

type TastyFruits = Exclude<Fruits, "mango">;        // "apple" | "banana"
type ActuallyNotAFruit = Extract<Fruits, "banana">; // "mango"

interface Fruit {
  name: string;
  color: string;
  tasty: boolean;
}

type BasicFruit = Omit<Fruit, 'tasty'>; // { name: string; color: string; }
type FruitName = Pick<Fruit, 'name'>;   // { name: string; }
```

Utility functions

Create types from other types

```ts
type Fruits = "apple" | "banana" | "mango";

type TastyFruits = Exclude<Fruits, "mango">;        // "apple" | "banana"
type ActuallyNotAFruit = Extract<Fruits, "banana">; // "mango"

interface Fruit {
  name: string;
  color: string;
  tasty: boolean;
}

type BasicFruit = Omit<Fruit, 'tasty'>; // { name: string; color: string; }
type FruitName = Pick<Fruit, 'name'>;   // { name: string; }
```

Working with APIs

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

  return data as UserDetails
}
```

Working with APIs

Enter Zod

```ts
import { z } from "zod";

const UserSchema = z.object({
  username: z.string()
});

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

  const parsedUser = UserSchema.safeParse(data);

  if (parsedUser.success === false) {
    console.log(parsedUser.error);
    throw new Error(parsedUser.error.message)
  } else {
    return parsedUser.data;
    //                ^? data: { username: string; }
  }
}
```

Working with APIs

Enter Zod

```
import { z } from "zod";

const UserSchema = z.object({
  username: z.string()
});

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

  const parsedUser = UserSchema.safeParse(data);

  if (parsedUser.success === false) {
    console.log(parsedUser.error);
    throw new Error(parsedUser.error.message)
  } else {
    return parsedUser.data;
    //                ^? data: { username: string; }
  }
}
```

Working with APIs

Enter Zod

```ts
import { z } from "zod";

const UserSchema = z.object({
  username: z.string()
});

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

  const parsedUser = UserSchema.safeParse(data);

  if (parsedUser.success === false) {
    console.log(parsedUser.error);
    throw new Error(parsedUser.error.message)
  } else {
    return parsedUser.data;
    //                ^? data: { username: string; }
  }
}
```

Working with APIs

Enter Zod

```
import { z } from "zod";

const UserSchema = z.object({
  username: z.string()
});

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

  const parsedUser = UserSchema.safeParse(data);

  if (parsedUser.success === false) {
    console.log(parsedUser.error);
    throw new Error(parsedUser.error.message)
  } else {
    return parsedUser.data;
    //                ^? data: { username: string; }
  }
}
```

Working with APIs

Enter Zod

```ts
import { z } from "zod";

const UserSchema = z.object({
  username: z.string()
});

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

  const parsedUser = UserSchema.safeParse(data);

  if (parsedUser.success === false) {
    console.log(parsedUser.error);
    throw new Error(parsedUser.error.message)
  } else {
    return parsedUser.data;
    //                ^? data: { username: string; }
  }
}
```

Working with APIs

Enter Zod

```
import { z } from "zod";

const UserSchema = z.object({
  username: z.string()
});

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

  const parsedUser = UserSchema.safeParse(data);

  if (parsedUser.success === false) {
    console.log(parsedUser.error);
    throw new Error(parsedUser.error.message)
  } else {
    return parsedUser.data;
    //                ^? data: { username: string; }
  }
}
```

Working with APIs

Enter Zod

```
import { z } from "zod";

const UserSchema = z.object({
  username: z.string()
});

const parsedUser = UserSchema.safeParse(data);

type ParseResult = SuccessUser | ErrorUser;

type SuccessUser = {
  success: true;
  data: {
    username: string;
  }
}

type ErrorUser = {
  success: false;
  error: ZodError;
}
```

Working with APIs

Enter Zod

```
import { z } from "zod";

const UserSchema = z.object({
  username: z.string()
});

const parsedUser = UserSchema.safeParse(data);

type ParseResult = SuccessUser | ErrorUser;

type SuccessUser = {
  success: true;
  data: {
    username: string;
  }
}

type ErrorUser = {
  success: false;
  error: ZodError;
}
```

Working with APIs

Enter Zod

```
import { z } from "zod";

const UserSchema = z.object({
  username: z.string()
});

const parsedUser = UserSchema.safeParse(data);

type ParseResult = SuccessUser | ErrorUser;

type SuccessUser = {
  success: true;
  data: {
    username: string;
  }
}

type ErrorUser = {
  success: false;
  error: ZodError;
}
```

Working with APIs

Enter Zod

```
import { z } from "zod";

const UserSchema = z.object({
  username: z.string()
});

const parsedUser = UserSchema.safeParse(data);

type ParseResult = SuccessUser | ErrorUser;

type SuccessUser = {
  success: true;
  data: {
    username: string;
  }
}

type ErrorUser = {
  success: false;
  error: ZodError;
}
```

Working with APIs

Enter Zod

```ts
import { z } from "zod";

const UserSchema = z.object({
  username: z.string()
});

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

  const parsedUser = UserSchema.safeParse(data);

  if (parsedUser.success === false) {
    console.log(parsedUser.error);
    throw new Error(parsedUser.error.message)
  } else {
    return parsedUser.data;
    //                ^? data: { username: string; }
  }
}
```

Working with APIs

Enter Zod

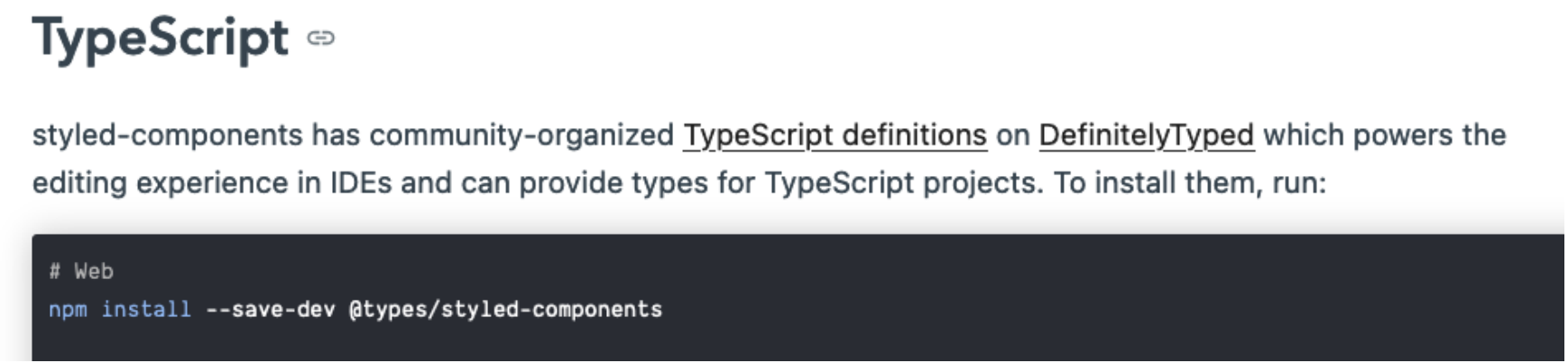
```ts
import { z } from "zod";

const UserSchema = z.object({
  username: z.string()
});

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

  const parsedUser = UserSchema.safeParse(data);

  if (parsedUser.success === false) {
    console.log(parsedUser.error);
    throw new Error(parsedUser.error.message)
  } else {
    return parsedUser.data;
    //                ^? data: { username: string; }
  }
}
```

Libraries and their types

Most libraries have specified types. Either built in or as a separate package.

Libraries and their types

Most libraries have specified types. Either built in or as a separate package.



Don’t be afraid to cmd-click into the definitions!

Libraries and their types

If we cmd-click into `safeParse` in the example from before

```
// src/node_modules/zod/lib/types.d.ts
safeParse(data: unknown, params?: Partial<ParseParams>): SafeParseReturnType<Input, Output>;

type SafeParseReturnType<Input, Output> = SafeParseSuccess<Output> | SafeParseError<Input>;

type SafeParseSuccess<Output> = {
    success: true;
    data: Output;
};

type SafeParseError<Input> = {
    success: false;
    error: ZodError<Input>;
};

class ZodError<T = any> extends Error {
    issues: ZodIssue[];
    // ...
}
```

Libraries and their types

If we cmd-click into `safeParse` in the example from before

```
// src/node_modules/zod/lib/types.d.ts
safeParse(data: unknown, params?: Partial<ParseParams>): SafeParseReturnType<Input, Output>;

type SafeParseReturnType<Input, Output> = SafeParseSuccess<Output> | SafeParseError<Input>;

type SafeParseSuccess<Output> = {
    success: true;
    data: Output;
};

type SafeParseError<Input> = {
    success: false;
    error: ZodError<Input>;
};

class ZodError<T = any> extends Error {
    issues: ZodIssue[];
    // ...
}
```

Libraries and their types

If we cmd-click into `safeParse` in the example from before

```ts
// src/node_modules/zod/lib/types.d.ts
safeParse(data: unknown, params?: Partial<ParseParams>): SafeParseReturnType<Input, Output>;

type SafeParseReturnType<Input, Output> = SafeParseSuccess<Output> | SafeParseError<Input>;

type SafeParseSuccess<Output> = {
    success: true;
    data: Output;
};

type SafeParseError<Input> = {
    success: false;
    error: ZodError<Input>;
};

class ZodError<T = any> extends Error {
    issues: ZodIssue[];
    // ...
}
```

Libraries and their types

If we cmd-click into `safeParse` in the example from before

```
// src/node_modules/zod/lib/types.d.ts
safeParse(data: unknown, params?: Partial<ParseParams>): SafeParseReturnType<Input, Output>;

type SafeParseReturnType<Input, Output> = SafeParseSuccess<Output> | SafeParseError<Input>;

type SafeParseSuccess<Output> = {
    success: true;
    data: Output;
};

type SafeParseError<Input> = {
    success: false;
    error: ZodError<Input>;
};

class ZodError<T = any> extends Error {
    issues: ZodIssue[];
    // ...
}
```

Libraries and their types

If we cmd-click into `safeParse` in the example from before

```
// src/node_modules/zod/lib/types.d.ts
safeParse(data: unknown, params?: Partial<ParseParams>): SafeParseReturnType<Input, Output>;

type SafeParseReturnType<Input, Output> = SafeParseSuccess<Output> | SafeParseError<Input>;

type SafeParseSuccess<Output> = {
    success: true;
    data: Output;
};

type SafeParseError<Input> = {
    success: false;
    error: ZodError<Input>;
};

class ZodError<T = any> extends Error {
    issues: ZodIssue[];
    // ...
}
```

Integrating with frameworks, vue, react

React

```
interface Props {
  nomination: Nomination;
  film: Film;
  bets: Bet[];
  players: Nullable<NormalizedPlayers>;
}

const NominatedFilmComponent: React.FC<Props> = ({
  nomination,
  film,
  bets,
  players
}) => {
  return (
    <Wrapper winner={nomination.won}>
      <Poster alt={film.name} src={poster} />
      {nomination.nominee && <p>{nomination.nominee}</p>}
      {bettingPlayers}
    </Wrapper>
  );
};
```

Integrating with frameworks, vue, react

React

```
interface Props {
  nomination: Nomination;
  film: Film;
  bets: Bet[];
  players: Nullable<NormalizedPlayers>;
}

const NominatedFilmComponent: React.FC<Props> = ({
  nomination,
  film,
  bets,
  players
}) => {
  return (
    <Wrapper winner={nomination.won}>
      <Poster alt={film.name} src={poster} />
      {nomination.nominee && <p>{nomination.nominee}</p>}
      {bettingPlayers}
    </Wrapper>
  );
};
```

Integrating with frameworks, vue, react

React

```
interface Props {
  nomination: Nomination;
  film: Film;
  bets: Bet[];
  players: Nullable<NormalizedPlayers>;
}

const NominatedFilmComponent: React.FC<Props> = ({
  nomination,
  film,
  bets,
  players
}) => {
  return (
    <Wrapper winner={nomination.won}>
      <Poster alt={film.name} src={poster} />
      {nomination.nominee && <p>{nomination.nominee}</p>}
      {bettingPlayers}
    </Wrapper>
  );
};
```

Integrating with frameworks, vue, react

React

```
interface Props {
  nomination: Nomination;
  film: Film;
  bets: Bet[];
  players: Nullable<NormalizedPlayers>;
}

const NominatedFilmComponent: React.FC<Props> = ({
  nomination,
  film,
  bets,
  players
}) => {
  return (
    <Wrapper winner={nomination.won}>
      <Poster alt={film.name} src={poster} />
      {nomination.nominee && <p>{nomination.nominee}</p>}
      {bettingPlayers}
    </Wrapper>
  );
};
```

Integrating with frameworks, vue, react

Vue

```vue
<script setup lang="ts">
export interface Props {
  title?: string;
  text?: string;
  linkText?: string;
  linkUrl?: string;
}

defineProps<Props>();
</script>

<template>
  <div class="box">
    <h2 v-if="title" class="title">{{ title }}</h2>
    <p v-if="text" class="text">{{ text }}</p>
    <a v-if="linkText && linkUrl" class="link" :href="linkUrl">
      <SwLinkButton :label="linkText" />
    </a>
  </div>
</template>
```

Integrating with frameworks, vue, react

Vue

```vue
<script setup lang="ts">
export interface Props {
  title?: string;
  text?: string;
  linkText?: string;
  linkUrl?: string;
}

defineProps<Props>();
</script>

<template>
  <div class="box">
    <h2 v-if="title" class="title">{{ title }}</h2>
    <p v-if="text" class="text">{{ text }}</p>
    <a v-if="linkText && linkUrl" class="link" :href="linkUrl">
      <SwLinkButton :label="linkText" />
    </a>
  </div>
</template>
```

Integrating with frameworks, vue, react

Vue

```vue
<script setup lang="ts">
export interface Props {
  title?: string;
  text?: string;
  linkText?: string;
  linkUrl?: string;
}

defineProps<Props>();
</script>

<template>
  <div class="box">
    <h2 v-if="title" class="title">{{ title }}</h2>
    <p v-if="text" class="text">{{ text }}</p>
    <a v-if="linkText && linkUrl" class="link" :href="linkUrl">
      <SwLinkButton :label="linkText" />
    </a>
  </div>
</template>
```

Integrating with frameworks, vue, react

Vue

```vue
<script setup lang="ts">
export interface Props {
  title?: string;
  text?: string;
  linkText?: string;
  linkUrl?: string;
}

defineProps<Props>();
</script>

<template>
  <div class="box">
    <h2 v-if="title" class="title">{{ title }}</h2>
    <p v-if="text" class="text">{{ text }}</p>
    <a v-if="linkText && linkUrl" class="link" :href="linkUrl">
      <SwLinkButton :label="linkText" />
    </a>
  </div>
</template>
```

Resources

- Typescript Handbook – typescriptlang.org/docs/handbook/intro.html
- Zod – zod.dev
- These slides (made with slidev) – per.fyi/talks