

The background of the slide is a blurred photograph of a sandy beach or dunes, with warm, golden tones.

TypeScript in React projects

Table of Contents

1. Structural typing vs nominal typing
2. Modeling types
3. Using types
4. React and TSX
5. Migrating to TypeScript



Structural typing vs nominal typing

Structural typing

Structural typing means that types with the same structure are compatible regardless of name. TypeScript uses structural typing by default.

```
1 type MacBook = {  
2   name: string;  
3   colour: string;  
4   number_of_feet: number;  
5   has_fans: boolean;  
6   can_be_in_a_room: boolean;  
7 };  
8  
9 type Animal = {  
10   name: string;  
11   colour: string;  
12   number_of_feet: number;  
13   has_fans: boolean;  
14   can_be_in_a_room: boolean;  
15 };
```

```
1 const macBookPro: MacBook = {  
2   name: 'Mac',  
3   colour: 'gray',  
4   number_of_feet: 4,  
5   has_fans: true,  
6   can_be_in_a_room: true,  
7 };  
8  
9 const elephant: Animal = {  
10   name: 'Benjamin',  
11   colour: 'gray',  
12   number_of_feet: 4,  
13   has_fans: true,  
14   can_be_in_a_room: true,  
15 };  
16  
17 const listOfMacBooks: MacBook[] = [  
18   macBookPro,  
19   elephant, // no error  
20 ];
```

Nominal typing

Nominal typing means that each type is unique and types can not be used interchangeably even when structurally identically. Compatibility can be created via interfaces or inheritance.

Nominal Typing is widely used in languages like Java or C#. Flow uses a mix of nominal and structural typing. Nominal Typing can be emulated in TypeScript via Branded Types.

```
1  public interface IMacBookAnimal {}  
2  public record MacBook(String colour) implements IMacBookAnimal {}  
3  public record Animal(String colour) implements IMacBookAnimal {}  
4  
5  public class NominalTypingExample {  
6      public static void main(String[] args) {  
7          var macBookPro = new MacBook("gray");  
8          var elephant = new Animal("gray");  
9          // List<MacBook> listOfMacBooks = Arrays.asList(macBookPro, elephant); // error  
10         List<IMacBookAnimal> listOfMacBooks = Arrays.asList(macBookPro, elephant); // no error  
11     }  
12 }
```

Modeling types

TypeScript provides a very powerful type system to model types that can be used to model parameters, return values, json files etc.

Example payload from an API-endpoint

```
1  [
2    {
3      "id": "B122",
4      "firstName": "Sherlock",
5      "lastName": "Holmes",
6      "house": "221B",
7      "street": "Baker Street",
8      "city": "London",
9      "postcode": "NW16XE",
10     "defaultedPayments": 1,
11     "missedPayments": 50
12   }
13 ]
```

Corresponding type

```
1  type Customer = {
2    id: string;
3    firstName: string;
4    lastName: string;
5    house: string;
6    street: string;
7    city: string;
8    postcode: string;
9    defaultedPayments: number;
10   missedPayments: number;
11 };
12
13 type CustomerList = Customer[];
```

When using types it is possible to split them up into logical units and composite new types from other types via **Intersection types**.

```
1 type CustomerDefaultFields = {  
2   id: string;  
3   firstName: string;  
4   lastName: string;  
5 };  
6 type CustomerAddress = {  
7   house: string;  
8   street: string;  
9   city: string;  
10  postcode: string;  
11};  
12 type CustomerPaymentStatus = {  
13  defaultedPayments: number;  
14  missedPayments: number;  
15};  
16 type Customer = CustomerDefaultFields  
17  & CustomerAddress;  
18 type CustomerWithPaymentData =  
19  & CustomerDefaultFields  
20  & CustomerPaymentStatus;
```



The same result can be achieved via `extend` when using interfaces.

```
1  interface CustomerDefaultFields {
2    id: string;
3    firstName: string;
4    lastName: string;
5  }
6  interface CustomerAddress {
7    house: string;
8    street: string;
9    city: string;
10   postcode: string;
11 }
12 interface CustomerPaymentStatus {
13   defaultedPayments: number;
14   missedPayments: number;
15 }
16 interface Customer extends
17   CustomerDefaultFields,
18   CustomerAddress {}
19 interface CustomerWithPaymentData extends
20   CustomerDefaultFields,
21   CustomerAddress,
22   CustomerPaymentStatus {}
```



Types can also be created by including or excluding certain fields from a complex base type via **Pick** or **Omit**.

```
1  type CustomerFields = {  
2    id: string;  
3    firstName: string;  
4    lastName: string;  
5    house: string;  
6    street: string;  
7    city: string;  
8    postcode: string;  
9    defaultedPayments: number;  
10   missedPayments: number;  
11 };  
12 type Customer = Pick<CustomerFields,  
13   'id' | 'firstName' | 'lastName' |  
14   'house' | 'street' | 'city' | 'postcode'  
15 >;  
16 type Customer2 = Omit<CustomerFields,  
17   'defaultedPayments' | 'missedPayments'  
18 >;
```



Pick and Omit can also be used when changing the type of certain fields, by removing the fields first and then adding them again with a different, e.g. narrower type. This comes in handy when working with JSON files.

```
1 import currency from 'currency.js';
2
3 type LoanLooselyTyped = {
4   id: string;
5   title: string;
6   tranche: string;
7   available_amount: string;
8   annualised_return: string;
9   term_remaining: string;
10  ltv: number;
11  loan_value: string;
12};
13
14 type Loan = Omit<LoanLooselyTyped,
15  'available_amount' | 'annualised_return' | 'term_remaining' | 'ltv' | 'loan_value'
16 > & {
17  available_amount: currency;
18  annualised_return: currency;
19  term_remaining: Date;
20  ltv: BigInt;
21  loan_value: currency;
22};
```

Union types can be used to allow a set of different values for a type. While often used as string unions, union types are not restricted to primitives. They are also used with objects that have common fields.

```
1  type AnimalSound = 'Meow' | 'Woof' | 'Moo' |
2    'Oink';
3
4  type Cat = {
5    name: string;
6    sound: AnimalSound;
7    isCurrentChiefMouser: boolean;
8  };
9
10 type Dog = {
11   name: string;
12   sound: AnimalSound;
13   canBeMistakenForAPony: boolean;
14 };
15
16 type Cow = {
17   name: string;
18   sound: AnimalSound;
19 };
20
21 type Animal = Cat | Dog | Cow;
22 type AnimalList = Animal[];
```

```
1  const cat: Cat = {
2    name: 'Mittens',
3    sound: 'Meow',
4    isCurrentChiefMouser: false,
5  };
6
7  const dog: Dog = {
8    name: 'Ben',
9    sound: 'Woof',
10   canBeMistakenForAPony: true,
11 };
12
13 const cow: Cow = {
14   name: 'Gertie',
15   sound: 'Meow', // no error
16 };
17
18 const animals: AnimalList = [cat, dog, cow];
```

Discriminated unions

When union types share a common field, that field can be used to automatically differentiate members of the union type. They are called **Discriminated unions**. The shared field is called **discriminant property**.

```
1  type AnimalSound = 'Meow' | 'Woof' | 'Moo' |  
2    'Oink';  
3  
4  type Cat = {  
5    type: 'Cat'; // discriminant property  
6    name: string;  
7    sound: AnimalSound;  
8    isCurrentChiefMouser: boolean;  
9  };  
10  
11 type Dog = {  
12  type: 'Dog';  
13  name: string;  
14  sound: AnimalSound;  
15  canBeMistakenForAPony: boolean;  
16};  
17  
18 type Animal = Cat | Dog;
```

```
1  const cat: Animal = { // TS narrows it down to Cat  
2    type: 'Cat',  
3    name: 'Mittens',  
4    sound: 'Meow',  
5    isCurrentChiefMouser: false,  
6    // canBeMistakenForAPony: false,  
7    // Error ('canBeMistakenForAPony' does not exist  
8    // in type 'Cat')  
9  };  
10  
11 const dog: Animal = {  
12  type: 'Dog',  
13  name: 'Ben',  
14  sound: 'Woof',  
15  canBeMistakenForAPony: true,  
16};
```

Logical union type combinations

Union types can also be modeled in such a way, that unique logical combinations are created.

TypeScript can use those to automatically differentiate members of the union type.

```
1  type DatePickerBase = {
2    day: number;
3    month: number;
4    year: number;
5  };
6  type DatePickerRegular = {
7    maxNumberOfDays?: number; // optional
8    showTimePicker: boolean; // unique
9  };
10 type DatePickerDateOfBirthMode = {
11   minAge?: number; // optional
12   maxAge?: number; // optional
13   showIAmOldEnoughBox: boolean; // unique
14 };
15 type DatePicker = DatePickerBase
16   & (DatePickerRegular
17     | DatePickerDateOfBirthMode);
```



Mapped types can be used to iterate over fields of one or more types to create a new type. This is useful with keeping the type declarations DRY.

```
1  type CustomerTypeBackend = {  
2      id: string;  
3      first_name: string;  
4      last_name: string;  
5      defaulted_payments: number;  
6      missed_payments: number;  
7      // additional_field_1: number[];  
8  }  
9  
10 type GlobalKeyMap = {  
11     id: 'id',  
12     first_name: 'firstName',  
13     last_name: 'lastName',  
14     house: 'house',  
15     street: 'street',  
16     city: 'city',  
17     post_code: 'postCode',  
18     defaulted_payments: 'defaultedPayments',  
19     missed_payments: 'missedPayments',  
20     additionalField2: 'additional_field_2';  
21 }
```

```
1  type CustomerTypeFrontendWithoutMismatchingData = {  
2      [K in keyof CustomerTypeBackend as  
3      GlobalKeyMap[K]]: CustomerTypeBackend[K]  
4  }
```

Output:

```
1  type CustomerTypeFrontendWithoutMismatchingData = {  
2      id: string;  
3      firstName: string;  
4      lastName: string;  
5      defaultedPayments: number;  
6      missedPayments: number;  
7  }
```

```
1 type CustomerTypeBackend = {  
2   id: string;  
3   first_name: string;  
4   last_name: string;  
5   defaulted_payments: number;  
6   missed_payments: number;  
7   additional_field_1: number[];  
8 };
```

```
1 type GlobalKeyMap = {  
2   id: 'id',  
3   first_name: 'firstName',  
4   last_name: 'lastName',  
5   ...  
6   defaulted_payments: 'defaultedPayments',  
7   missed_payments: 'missedPayments',  
8   additionalField2: 'additional_field_2';  
9 }
```

```
1 type CustomerTypeFrontend = {  
2   [K in keyof Pick<CustomerTypeBackend, Extract<keyof CustomerTypeBackend, keyof GlobalKeyMap>>]:  
3     as GlobalKeyMap[K]]: CustomerTypeBackend[K];  
4 };
```

Output

```
1 type CustomerTypeFrontend = {  
2   id: string;  
3   firstName: string;  
4   lastName: string;  
5   defaultedPayments: number;  
6   missedPayments: number;  
7 }
```

Mapped types can also iterate over fields of multiple types to create new types.

```
1 type CurrencyAbbreviations = 'GBP' | 'EUR' | 'USD'  
2   | 'CAD' | 'AUD';  
3  
4 type CurrencyNames = 'Pound' | 'Euro' | 'Dollar'  
5   | 'Canadian Dollar' | 'Australian Dollar';  
6  
7 type CurrencySymbols = '£' | '€' | '$';  
8  
9 type Currencies = {  
10   [K in CurrencyAbbreviations]?: {  
11     name: CurrencyNames;  
12     symbol: CurrencySymbols;  
13   };  
14 };  
15  
16 // Alternative approach using Record and Partial  
17 type CurrenciesAlternative = Partial<Record<  
18   CurrencyAbbreviations, {  
19     name: CurrencyNames;  
20     symbol: CurrencySymbols;  
21   }  
22 >>;
```

```
1 const currencies: Currencies = {  
2   GBP: {  
3     name: 'Pound',  
4     symbol: '£',  
5   },  
6   EUR: {  
7     name: 'Euro',  
8     symbol: '€',  
9   },  
10};
```

Since mapped types provide the current type within the map function, a type lookup can be performed, where the current type acts as the key, similar to looking up values in an object.

```
1  type CurrencyAbbreviations = 'GBP' | 'EUR' | 'USD'  
2    | 'CAD' | 'AUD';  
3  type CurrencyNames = {  
4    GBP: 'Pound' | 'Pound Sterling';  
5    EUR: 'Euro';  
6    USD: 'Dollar' | 'US Dollar';  
7    CAD: 'Canadian Dollar';  
8    AUD: 'Australian Dollar' | 'Dollarydoos';  
9  };  
10 type CurrencySymbols = {  
11   GBP: '£';  
12   EUR: '€';  
13   USD: '$';  
14   CAD: '$';  
15   AUD: '$';  
16 };  
17 type Currencies = {  
18   [K in CurrencyAbbreviations]?: [  
19     name: CurrencyNames[K],  
20     symbol: CurrencySymbols[K],  
21   ];  
22 };
```

```
1  const currencies: Currencies = {  
2    EUR: ['Euro', '€'],  
3    GBP: ['Pound', '£'],  
4    USD: ['Dollar', '$'],  
5    CAD: ['Canadian Dollar', '$'],  
6    AUD: ['Dollarydoos', '$'],  
7    // CAD: ['Euro', '$'], // Error  
8    // GBP: ['Pound Sterling', '$'], // Error  
9  };
```

Using types

Differentiating values

TypeScript can infer values via control flow analysis and allow or disallow certain operations at specific points within the code.

```
1  export type Person = {
2    firstName: string;
3    lastName: string;
4    address?: {
5      house: string;
6      street: string;
7      postcode: string;
8      city: string;
9    };
10 }
```

```
1  function showPersonDetails(person: Person): void {
2    const { firstName, lastName, address } = person;
3
4    console.log(`First name: ${firstName}`);
5    console.log(`Last name: ${lastName}`);
6
7    if (address) {
8      const { house, street, postcode,
9      city } = address;
10
11      console.log(`House: ${house}`);
12      console.log(`Street: ${street}`);
13      console.log(`Postcode: ${postcode}`);
14      console.log(`City: ${city}`);
15    }
16 }
```

Differentiating custom types

TypeScript can narrow down types similar to values via control flow analysis and.

```
1 type Person = {  
2   firstName: string;  
3   lastName: string;  
4 };  
5  
6 type PersonWithAddress = Person & {  
7   address: {  
8     house: string;  
9     street: string;  
10    postcode: string;  
11    city: string;  
12  };  
13};  
14  
15 type People = Person | PersonWithAddress;
```

```
1 function showPersonDetails(  
2   person: Person | PersonWithAddress): void {  
3   const { firstName, lastName } = person;  
4  
5   console.log(`First name: ${firstName}`);  
6   console.log(`Last name: ${lastName}`);  
7  
8   // in-operator narrows down union types  
9   if ('address' in person) {  
10      // PersonWithAddress  
11      const type = person;  
12      const { house, street, postcode, city } =  
13        person.address;  
14  
15      console.log(`House: ${house}`);  
16      console.log(`Street: ${street}`);  
17      console.log(`Postcode: ${postcode}`);  
18      console.log(`City: ${city}`);  
19    }  
20  }
```

Discriminated unions

Discriminate unions are a way to narrow down union types by using a single field called `discriminant property`.

```
1  type AnimalSound = 'Meow' | 'Woof';
2
3  type Cat = {
4    type: 'Cat';
5    name: string;
6    sound: AnimalSound;
7    isCurrentChiefMouser: boolean;
8  };
9
10 type Dog = {
11   type: 'Dog';
12   name: string;
13   sound: AnimalSound;
14   canBeMistakenForAPony: boolean;
15 };
16
17 type Animal = Cat | Dog;
18
19 type AnimalList = [Animal, Animal]; // Tuple
```

```
1  function showAnimalDetails(animal: Animal): void {
2    const { type, name, sound } = animal;
3
4    console.log(name);
5    console.log(sound);
6
7    if (type === 'Cat') {
8      console.log(animal.isCurrentChiefMouser);
9
10      return;
11    }
12
13    console.log(animal.canBeMistakenForAPony);
14  }
```

Type predicates & Type guards

Todo

```
1  function isCustomerWithDefaultedPayments(customer: Customer): customer is CustomerWithDefaultedPayments {
2      const { hasDefaultedPayments } = customer;
3
4      return hasDefaultedPayments;
5  }
6
7  function isCustomerWithMissedPayments(customer: Customer): customer is CustomerWithMissedPayments {
8      const { hasDefaultedPayments, hasMissedPayments } = customer;
9
10     if (hasDefaultedPayments) {
11         return false;
12     }
13
14     return hasMissedPayments;
15 }
16
17 function isCustomerRegular(customer: Customer): customer is CustomerRegular {
18     const { hasDefaultedPayments, hasMissedPayments } = customer;
19
20     return !hasDefaultedPayments && !hasMissedPayments;
21 }
```

React and TSX

Deprecate props

TypeScript's `never` type can be used to hard deprecate component props. This is most useful for library or styleguide maintainers. Trying to use the deprecated prop, will result in a type error. Alternatively a soft deprecation can be achieved by using the JSDoc deprecated tag, e.g. `@deprecated Please use x instead`.

Props

```
1  type CustomButtonProps = Partial<Pick<HTMLButt
2  {
3      onClick: (event?: MouseEvent<HTMLButtonEleme
4      oldProp?: never, // is deprecated
5      newProp: string,
6  };
```



```
1 import React, { PropsWithChildren } from 'react';
2 import { CustomButtonProps } from './library';
3
4 function CustomButton({ children, onClick, newProp }: PropsWithChildren<CustomButtonProps>) {
5   return (
6     <button type="button" onClick={onClick} data-testid={newProp}>
7       {children}
8     </button>
9   );
10 }
11
12 default function App() {
13   function handleClick(): void {}
14
15   return (
16     <CustomButton
17       onClick={() => handleClick()}
18       newProp="test"
19       // oldProp="test" // error
20     >
21       Button
22     </CustomButton>
23   );
24 }
```

Disallow prop combinations

TypeScript's `'never'` type can also be used to prohibit certain combination of props. The syntax can get a bit complex for certain type of props, e.g. `'boolean'` as those type of props can either be set to false or not passed at all.

Component

```
1  function ButtonLink({ onClick, type, disabled, href, children }: PropsWithChildren<ButtonLinkProps>) {
2    const TagType = type; // either a or button
3    if (type === 'button') {
4      console.log(href); // undefined
5      console.log(disabled);
6    }
7    if (type === 'a') {
8      console.log(href); // string
9      console.log(disabled);
10   }
11   return (
12     <TagType className={style.ButtonLink} onClick={onClick}>
13       {children}
14     </TagType>
15   );
16 }
```

We use the type field to distinguish between various types and allow TypeScript to narrow down the type.

Props

```
1  type ButtonLinkCommonProps = {
2    onClick: (() => void);
3  }
4
5  type ButtonLinkButtonProps = {
6    type: 'button';
7    disabled?: boolean;
8    href?: never; // button should never have a href attribute
9  }
10
11 type ButtonLinkLinkProps = {
12   type: 'a';
13   disabled?: false; // link should never have a disabled attribute
14   href: string;
15 }
16
17 type ButtonLinkProps = ButtonLinkCommonProps & (ButtonLinkButtonProps | ButtonLinkLinkProps);
```

Prohibit child components

TypeScript's `never` type can also be used to prevent child components. This is useful for library authors to prevent composition and use a component as is.

App

```
1 import Child from './Child';
2 import Childless from './Childless';
3
4 function App() {
5   return (
6     <div className="wrapper">
7       <h1>Restrict child elements</h1>
8       <Child>
9         <h2>Can have child content</h2>
10      </Child>
11      <Childless />
12    </div>
13  );
14}
```



Component with children

```
1 import { PropsWithChildren } from 'react';
2
3 type ChildProps = PropsWithChildren<{
4   // same as PropsWithChildren<
5   // children?: ReactNode | undefined;
6   otherProp: string;
7 }>;
```

Component without children

```
1 type ChildlessProps = {
2   children?: never;
3   otherProp: string;
4 };
```



Migrating to TypeScript

Soft migration

Soft migration allows a gradual migration in parallel to regular feature work, albeit at a slower pace.

1. Add a `jsconfig.json` file and set `checkJs` to `false` to disable type checking in JS files
2. Set `checkJs` to `true`, when working on the migration or enable it on a per file basis via adding a `// @ts-check` line at the start of a JS file
3. Fix all errors that are fixable without type annotations and disable the remaining ones by prepending each with `// @ts-ignore`
4. Install TypeScript dependencies and necessary type packages like `@types/react-dom`
5. Create a script, that runs the TypeScript compiler in no-emit mode to check remaining errors: `npx -`

```
1  {
2      "compilerOptions": {
3          "target": "ES2022", // ES3 is default
4          "checkJs": false,
5          "allowJs": true,
6          "noEmit": true,
7      },
8      "exclude": [
9          "node_modules"
10     ],
11     "include": [
12         ".//**/*.js"
13     ]
14 }
```

Example `jsconfig.json` in project root

Soft migration (part 2)

6. Add TypeScript to build pipelines
7. Create a `tsconfig.json` file via `npx -p typescript tsc --init` and disable strict mode, strict null checks and no implicit any
8. Rename all `*.js` files to `*.ts` and `*.tsx`
9. Reenable previously disabled errors and fix them
10. Add TypeScript-specific linter rules, f.e. `eslint-config-airbnb-typescript` and adapt code to new linter rules
11. Enable strict mode, strict null checks and no implicit any again
12. Fix TypeScript errors caused by stricter TypeScript settings

```
1  {
2    "compilerOptions": {
3      "target": "ES2022", // ES3 is default
4      "strict": false,
5      "strictNullChecks": false,
6      "noImplicitAny": false
7    },
8    "lib": [
9      "dom",
10     "dom.iterable"
11   ],
12   "exclude": [
13     "node_modules"
14   ],
15   "include": [
16     ".//**/*.tsx",
17     ".//**/*.ts"
18   ]
19 }
```

Example `tsconfig.json` in project root

Big bang migration

Allows a faster migration by converting all files at once at the expense of stopping feature work for a certain period of time. It might require freezing the code base for the duration of the migration.

1. Install TypeScript dependencies and necessary type packages like `@types/react-dom`
2. Create a `tsconfig.json` file via `npx -p typescript tsc --init` and disable strict mode, strict null checks and no implicit any
3. Rename all `*.js` files to `*.ts` and `*.tsx`
4. Fix all TypeScript errors
5. Add TypeScript to build pipelines
6. Enable strict mode, strict null checks and no implicit any again
7. Fix TypeScript errors caused by stricter TypeScript settings
8. Improve types for components, tests etc. further