
TypeScript in frontend projects

Table of Contents

1. Structural typing vs nominal typing
2. Modeling types
3. Using types
4. React and TSX

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.

```ts
type MacBook = {
  name: string;
  colour: string;
  numberOfFeet: number;
  hasFans: boolean;
  canBeInARoom: boolean;
};

type Animal = {
  name: string;
  colour: string;
  numberOfFeet: number;
  hasFans: boolean;
  canBeInARoom: boolean;
};
```

```ts
const macBookPro: MacBook = {
  name: 'Mac',
  colour: 'gray',
  numberOfFeet: 4,
  hasFans: true,
  canBeInARoom: true,
};

const elephant: Animal = {
  name: 'Benjamin',
  colour: 'gray',
  numberOfFeet: 4,
  hasFans: true,
  canBeInARoom: true,
};

const listOfMacBooks: MacBook[] = [
  macBookPro,
  elephant, // no error
];
```

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.

```
public interface IMacBookAnimal {}
public record MacBook(String colour) implements IMacBookAnimal {}
public record Animal(String colour) implements IMacBookAnimal {}

public class NominalTypingExample {
  public static void main(String[] args) {
    var macBookPro = new MacBook("gray");
    var elephant = new Animal("gray");
    // List<MacBook> listOfMacBooks = Arrays.asList(macBookPro, elephant); // error
    List<IMacBookAnimal> listOfMacBooks = Arrays.asList(macBookPro, elephant); // no error
  }
}
```

Modeling types

Types & interfaces

Types and interfaces can be utilized to create shapes for data. Both can be used interchangeably in a lot of cases, but they differ in syntax and mutability. Interfaces are mutable, while types are not. Unlike interfaces, types can also be used for primitives like number or string.

```
type CustomerID = string;

type Customer1 = {
  id: CustomerID;
  firstName: string;
  lastName: string;
};

interface Customer2 {
  id: CustomerID;
  firstName: string;
  lastName: string;
}
```

Indexed Access Types

Keys/members of types and interfaces can be accessed via bracket notation in other types and interfaces or when creating variables. Type information of referenced types are preserved.

```
type CustomerID = string;

type Customer1 = {
  id: CustomerID;
  firstName: string;
  lastName: string;
};
const customerLastName:
  Customer1['lastName'] = 'Larry';

interface Customer2 {
  id: CustomerID;
  firstName: string;
  lastName: string;
}
const customerLastName2:
  Customer2['lastName'] = 'Peter';
```

Creating types from other types

Example payload from an API-endpoint

```
[
  {
    "id": "B122",
    "firstName": "Sherlock",
    "lastName": "Holmes",
    "house": "221B",
    "street": "Baker Street",
    "city": "London",
    "postcode": "NW16XE",
    "defaultedPayments": 1,
    "missedPayments": 50
  }
]
```

Corresponding monolithic type

```
type Customer = {
  id: string;
  firstName: string;
  lastName: string;
  house: string;
  street: string;
  city: string;
  postcode: string;
  defaultedPayments: number;
  missedPayments: number;
};

type CustomerList = Customer[];
```

Intersection types

Intersection types can be used to compose types from subtypes or other, unrelated types.

```
type CustomerDefaultFields = {
  id: string;
  firstName: string;
  lastName: string;
};
type CustomerAddress = {
  house: string;
  street: string;
  city: string;
  postcode: string;
};
type CustomerPaymentStatus = {
  defaultedPayments: number;
  missedPayments: number;
};
type Customer = CustomerDefaultFields
  & CustomerAddress;
type CustomerWithPaymentData =
  & CustomerDefaultFields
  & CustomerPaymentStatus;
```

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

```
interface CustomerDefaultFields {
  id: string;
  firstName: string;
  lastName: string;
}
interface CustomerAddress {
  house: string;
  street: string;
  city: string;
  postcode: string;
}
interface CustomerPaymentStatus {
  defaultedPayments: number;
  missedPayments: number;
}
interface Customer extends
  CustomerDefaultFields,
  CustomerAddress {}
interface CustomerWithPaymentData extends
  CustomerDefaultFields,
  CustomerAddress,
  CustomerPaymentStatus {}
```

Pick & Omit

Types can also be created by including or excluding certain keys/members from a complex base type via the Pick and Omit utility types.

```
type CustomerFields = {
  id: string;
  firstName: string;
  lastName: string;
  house: string;
  street: string;
  city: string;
  postcode: string;
  defaultedPayments: number;
  missedPayments: number;
};
type Customer1 = Pick<CustomerFields,
  'id' | 'firstName' | 'lastName' |
  'house' | 'street' | 'city' | 'postcode'
>;
type Customer2 = Omit<CustomerFields,
  'defaultedPayments' | 'missedPayments'
>;
```

Pick and Omit can also be used to change the type of certain keys, by removing the keys first and then adding them again with a different, e.g. narrower type. This comes in handy when dealing with incoming data that needs to be adapted to the type system of an application.

```
import currency from 'currency.js';

type LoanLooselyTyped = {
  id: string;
  title: string;
  tranche: string;
  available_amount: string;
  annualised_return: string;
  term_remaining: string;
  ltv: number;
  loan_value: string;
};
type Loan = Omit<LoanLooselyTyped,
  'available_amount' | 'annualised_return' | 'term_remaining' | 'ltv' | 'loan_value'
> & {
  available_amount: currency;
  annualised_return: currency;
  term_remaining: Date;
  ltv: BigInt;
  loan_value: currency;
};
```

Union types

Union types can be used to describe a set of different types, that are assignable to a key/member. Union types can consist of a range of other types like `string`, `boolean`, other custom types and also `undefined`.

```
type AnimalSound = 'Meow' | 'Woof' | 'Moo' |
  'Oink';

type Cat = {
  name: string;
  sound: AnimalSound;
  isCurrentChiefMouser: boolean;
};
type Dog = {
  name: string;
  sound: AnimalSound;
  canBeMistakenForAPony: boolean;
};
type Cow = {
  name: string;
  sound: AnimalSound;
};

type Animal = Cat | Dog | Cow;
type AnimalList = Animal[];
```

```
const cat: Cat = {
  name: 'Mittens',
  sound: 'Meow',
  isCurrentChiefMouser: false,
};

const dog: Dog = {
  name: 'Ben',
  sound: 'Woof',
  canBeMistakenForAPony: true,
};

const cow: Cow = {
  name: 'Gertie',
  sound: 'Meow', // no error
};

const animals: AnimalList = [cat, dog, cow];
```

Discriminated unions and discriminant property

When union types share a common field with literal string values, that field can be used to automatically differentiate types without explicit type assertion. These union types are called discriminated unions. The shared field is called discriminant property.

```ts
type AnimalSound = 'Meow' | 'Woof' | 'Moo' |
  'Oink';

type Cat = {
  type: 'Cat'; // discriminant property
  name: string;
  sound: AnimalSound;
  isCurrentChiefMouser: boolean;
};

type Dog = {
  type: 'Dog'; // discriminant property
  name: string;
  sound: AnimalSound;
  canBeMistakenForAPony: boolean;
};

type Animal = Cat | Dog;
```

```ts
// TS narrows Animal down to Cat
const cat: Animal = {
  type: 'Cat',
  name: 'Mittens',
  sound: 'Meow',
  isCurrentChiefMouser: false,
  // canBeMistakenForAPony: false,
  // Error ('canBeMistakenForAPony' does not exist
  // in type 'Cat')
};

// TS narrows Animal down to Dog
const dog: Animal = {
  type: 'Dog',
  name: 'Ben',
  sound: 'Woof',
  canBeMistakenForAPony: true,
};
```

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.

```
type DatePickerBase = {
  day: number;
  month: number;
  year: number;
};
type DatePickerRegular = {
  maxNumberOfDays?: number; // optional
  showTimePicker: boolean; // unique
};
type DatePickerDateOfBirthMode = {
  minAge?: number; // optional
  maxAge?: number; // optional
  showIAmOldEnoughBox: boolean; // unique
};
type DatePicker = DatePickerBase
  & (DatePickerRegular
    | DatePickerDateOfBirthMode);
```

Mapped types

Mapped types can be used to iterate over keys/members of multiple types to create new types/members in another type. This helps with DRY-ness. Additionally, mapped types also allow the adding of `readonly`- and `?` modifiers.

```ts
type CurrencyAbbreviations = 'GBP' | 'EUR' | 'USD'
  | 'CAD' | 'AUD';

type CurrencyNames = 'Pound' | 'Euro' | 'Dollar'
  | 'Canadian Dollar' | 'Australian Dollar';

type CurrencySymbols = '£' | '€' | '$';

type Currencies = {
  readonly [K in CurrencyAbbreviations]?: {
    name: CurrencyNames;
    symbol: CurrencySymbols;
  };
};
```

```ts
const currencies: Currencies = {
  GBP: {
    name: 'Pound',
    symbol: '£',
  },
  EUR: {
    name: 'Euro',
    symbol: '€',
  },
  USD: {
    name: 'Euro', // no error
    symbol: '$',
  },
};
```

Mapped types provide the ability to access the key of the current iteration. This can be used to lookup values by key, similar to looking up values in an object.

```
type CurrencyAbbreviations = 'GBP' | 'EUR' | 'USD'
  | 'CAD' | 'AUD';
type CurrencyNames = {
  GBP: 'Pound' | 'Pound Sterling';
  EUR: 'Euro';
  USD: 'Dollar' | 'US Dollar';
  CAD: 'Canadian Dollar';
  AUD: 'Australian Dollar' | 'Dollarydoos';
};
type CurrencySymbols = {
  GBP: '£';
  EUR: '€';
  USD: '$';
  CAD: '$';
  AUD: '$';
};
type Currencies = {
  readonly [K in CurrencyAbbreviations]?: [
    name: CurrencyNames[K],
    symbol: CurrencySymbols[K],
  ];
};
```

```
const currencies: Currencies = {
  EUR: ['Euro', '€'],
  GBP: ['Pound', '£'],
  USD: ['Dollar', '$'],
  CAD: ['Canadian Dollar', '$'],
  AUD: ['Dollarydoos', '$'],
  // CAD: ['Euro', '$'], // error
  // GBP: ['Pound Sterling', '$'], // error
};
```

Conditional types

Conditional types can can be used to conditionally output values by analyzing the input. They are often used in combination with mapped types and generics.

```ts
type Customer = {
  id: string;
  firstName: string;
  lastName: string;
  house: string;
  street: string;
  city: string;
  postCode: string;
  defaultedPayments: number;
  missedPayments: number;
};

type WithUppercaseValues<T> = {
  readonly [K in keyof T]?: T[K] extends string
    ? Uppercase<T[K]>
    : T[K];
};
```

```ts
const uppercaseCustomer: WithUppercaseValues<
  Customer
> = {
  // id: 'abcd', // error
  id: 'ABCD',
  firstName: 'LARRY',
};
```

Using types

Single types and optional keys

TypeScript can infer values via control flow analysis and may allow or disallow access for members of a type.

```
type Person = {
  firstName: string;
  lastName: string;
  address?: {
    house: string;
    street: string;
    postcode: string;
    city: string;
  };
};
```

```
function showPersonDetails(person: Person): void {
  const { firstName, lastName, address } = person;

  console.log(`First name: ${firstName}`);
  console.log(`Last name: ${lastName}`);

  if (address) {
    const { house, street, postcode,
    city } = address;

    console.log(`House: ${house}`);
    console.log(`Street: ${street}`);
    console.log(`Postcode: ${postcode}`);
    console.log(`City: ${city}`);
  }
}
```

Differentiating union types - manual

Narrowing down union types is slightly more complex since manually checking the type isn't possible as types are removed during compilation (type erasure).

```ts
type AnimalSound = 'Meow' | 'Woof' | 'Moo'|
  'Oink';

type Cat = {
  name: string;
  sound: AnimalSound;
  isCurrentChiefMouser: boolean;
};
type Dog = {
  name: string;
  sound: AnimalSound;
  canBeMistakenForAPony: boolean;
};
type Cow = {
  name: string;
  sound: AnimalSound;
};

type Animal = Cat | Dog | Cow;
```

```ts
function showAnimalDetails(animal: Animal): void {
  console.log(`Name: ${animal.name}`);
  console.log(`Sound: ${animal.sound}`);

  if (Object.hasOwn(animal,
    'isCurrentChiefMouser')) {
    const { isCurrentChiefMouser } = animal as Cat
    console.log(isCurrentChiefMouser);
    return;
  }

  if (Object.hasOwn(animal,
    'canBeMistakenForAPony')) {
    const { canBeMistakenForAPony } = animal as Dog
    console.log(canBeMistakenForAPony);
    return;
  }

  // Cow stuff
}
```

Differentiating union types - via type guards

TypeScript can narrow down multiple types in an union type via built-in type guards. Most common type guards are: `in`, `typeof` and `instanceof`.

```ts
type Person = {
  firstName: string;
  lastName: string;
};

type PersonWithAddress = Person & {
  address: {
    house: string;
    street: string;
    postcode: string;
    city: string;
  };
};

type People = Person | PersonWithAddress;
```

```ts
function showPersonDetails(person: People): void {
  const { firstName, lastName } = person;

  console.log(`First name: ${firstName}`);
  console.log(`Last name: ${lastName}`);

  if ('address' in person) {
    // PersonWithAddress
    const { house, street, postcode, city } =
      person.address;

    console.log(`House: ${house}`);
    console.log(`Street: ${street}`);
    console.log(`Postcode: ${postcode}`);
    console.log(`City: ${city}`);
  }
}
```

Differentiating union types - via discriminant union types

Since TypeScript is able to narrow down a discriminant union via a discriminant property, discriminant unions are widely used to ensure type security.

```
type ActionNamesTypes = 'UPDATE_COUNTER' |
  'UPDATE_DATETIME';

type State = {
  counter: Counter;
  dateTime: DateTime;
};

type CounterAction = {
  type: Extract<ActionNamesTypes,'UPDATE_COUNTER'>
  payload: Counter;
};

type DateTimeAction = {
  type: Extract<ActionNamesTypes,'UPDATE_DATETIME'>
  payload: DateTime;
};

type Actions = CounterAction | DateTimeAction;
```

```
const reducers = (state: State = initialState,
  action: Actions): State => {
  switch (action.type) {
    case ActionNames.UPDATE_COUNTER: {
      return {
        ...state,
        counter: action.payload, // Counter
      };
    }
    case ActionNames.UPDATE_DATETIME: {
      return {
        ...state,
        dateTime: action.payload, // DateTime
      };
    }
    default:
      return {
        ...state,
      };
  }
};
```

Differentiating union types - via discriminant union types with generics

Discriminant unions also work with generics to make code more reusable.

```ts
type Loading = {
  state: 'loading';
};
type Failed = {
  state: 'failed';
  statusCode: number;
  errorMessage?: string;
};
type Success<T> = {
  state: 'success';
  statusCode: number;
  data: T;
};

type APIRequestStatus<T> = Loading | Failed |
  Success<T>;

type Cat = {
  type: 'Cat';
  name: string;
};
```

```ts
async function sendAPIRequest<T>(): Promise<void> {
  const requestStatus = await getResponse<T>();
  switch (requestStatus.state) {
    case 'loading': {
      const { state } = requestStatus;
      console.log(state);
      return;
    }
    case 'failed': {
      const { errorMessage } = requestStatus;
      console.log(`${errorMessage
        ? errorMessage : '- no error message'}`);
      return;
    }
    case 'success': {
      const { data } = requestStatus;
      console.log(data);
      return;
    }
    default: {}
  }
}
```

React and TSX

Deprecate props

TypeScript's `never` type can be used to deprecate component props. 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 newProp instead`.

```
type CustomButtonProps = PropsWithChildren<
  Partial<Pick<HTMLButtonElement,
    'autofocus'| 'ariaDisabled'>
  > & {
    onClick: (event: MouseEvent<
      HTMLButtonElement
    >) => void,
    oldProp?: never, // is deprecated
    newProp: string,
  }
>;
```

```tsx
function CustomButton({children, onClick,
  newProp }: CustomButtonProps):
  ReactElement {
  return (
    <button type="button" onClick={onClick}
      data-testid={newProp}
    >
      {children}
    </button>
  );
}

function App() {
  function handleClick(): void {
    console.log('Click');
  }

  return (
    <CustomButton
      onClick={handleClick}
      newProp="test"
      // oldProp="test" // error
    >
      Button
    </CustomButton>
  );
}
```

Disallow prop combinations

TypeScript's `never` type can also be used to prohibit certain combinations of props.

```
function ButtonLink({
  onClick, as, disabled, href
}: PropsWithChildren<ButtonLinkProps>): {
  const Component = as;

  if (Component === 'button') {
    console.log(href); // undefined
    console.log(disabled);
  }
  if (Component === 'a') {
    console.log(href); // string
    console.log(disabled);
  }

  return (
    <Component className={style.ButtonLink}
      onClick={onClick}>
      Click
    </Component>
  );
}
```

The as-field is used as discriminant property to distinguish both types.

```
type ButtonLinkCommonProps = {
  onClick: () => void;
};

type ButtonLinkButtonProps = {
  as: 'button';
  disabled?: boolean;
  href?: never; // type button should
  // never have a href attribute
};

type ButtonLinkLinkProps = {
  as: 'a';
  disabled?: false; // type link should
  // never have a disabled attribute
  href: string;
};

type ButtonLinkProps = ButtonLinkCommonProps
  & (ButtonLinkButtonProps |
    ButtonLinkLinkProps);
```

Prohibit child components

The `never` type can also be used to prevent the creation of child components.

```tsx
type ChildProps = PropsWithChildren<{
  otherProp: string;
}>;

type ChildlessProps = {
  children?: never;
  otherProp: string;
};

function App(): ReactElement {
  return (
    <>
      <Child otherProp="test1">
        <h2>Can have child content</h2>
      </Child>
      <Childless otherProp="test2" />
    </>
  );
}
```