

# Advanced TypeScript types for fun and reliability

Maurice de Beijer  
@mauricedb



- Maurice de Beijer
- The Problem Solver
- Microsoft MVP
- Freelance developer/instructor
- Twitter: [@mauricedb](https://twitter.com/mauricedb)
- Web: <http://www.TheProblemSolver.nl>
- E-mail: [maurice.de.beijer@gmail.com](mailto:maurice.de.beijer@gmail.com)



# Workshop goal

- Use TypeScript's **strict settings** to catch as many errors as possible
- How to **validate data** against its type definition at runtime
- Using **mapped types** to create better type definitions
- Using **custom type mapping** definitions
- When to use the **unknown and any types**
- **Opaque types** for better type checking
- Using **type predicates and assertions**

Type it out  
by hand?

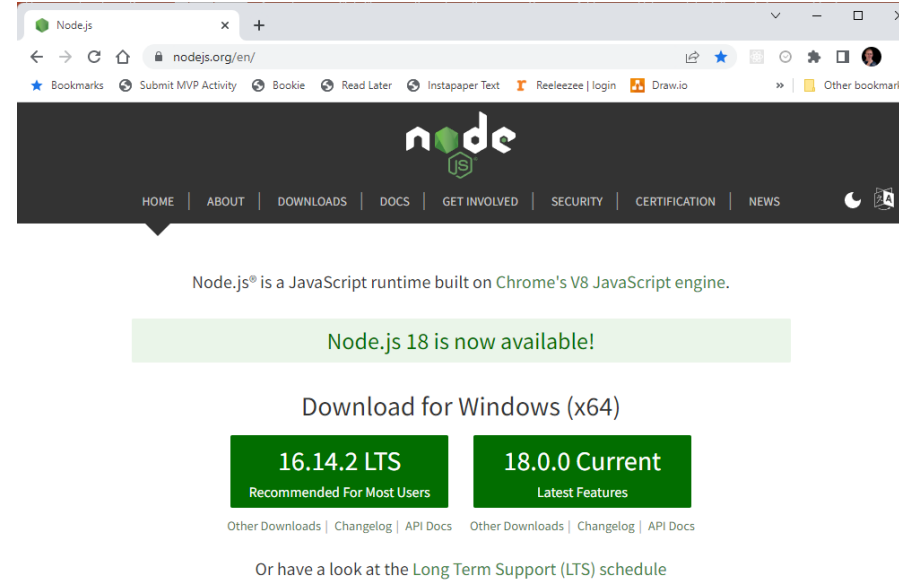
*"Typing it drills it into your brain much better than simply copying and pasting it. You're forming new neuron pathways. Those pathways are going to help you in the future. Help them out now!"*

# Prerequisites

Install Node & NPM

Install the GitHub repository

# Install Node.js & NPM



```
Windows PowerShell
PS C:\Temp> node --version
v12.18.3
PS C:\Temp> npm --version
6.14.8
PS C:\Temp> |
```

# Following Along

```
File Edit Selection View Go Run Terminal Help
TS main.ts IM TS types.ts IM X
src > TS types.ts > ...
37 declare const _type: unique symbol;
38
39 type Opaque<A, B> = A & {
40   readonly [_type]: B;
41 };
42
43 export type Amount = Opaque<number, 'Amount'>;
44 export type Account = Opaque<number, 'Account'>;
```

- Repository: <https://github.com/mauricedb/ts-advanced>
- Slides: <http://theproblemsolver.nl/docs/ts-advanced-workshop.pdf>

# The changes



Commits · mauricedb/ts-advance		
github.com/mauricedb/ts-advanced/commits/main		
main		
Commits on Apr 30, 2022		
Exhaustiveness Checking	mauricedb committed 1 hour ago	4b04e88
Type Assertion Functions	mauricedb committed 1 hour ago	45350f9
Type predicate functions	mauricedb committed 2 hours ago	5c82c4e
Commits on Apr 24, 2022		
Opaque types	mauricedb committed 6 days ago	34b1f61
Displaying Types	mauricedb committed 6 days ago	2146490
Type Mapping with Omit and Pick	mauricedb committed 6 days ago	ec6a9dc
Indexed Access Types	mauricedb committed 6 days ago	114140d
DeepReadonly<T>	mauricedb committed 6 days ago	9902b7e
Using Readonly<T>	mauricedb committed 6 days ago	b6b11ac
Unknown In Catch	mauricedb committed 6 days ago	b0d27f9
Inferring TS types	mauricedb committed 6 days ago	7b38bc2
Validating data at the boundary	mauricedb committed 6 days ago	020826b
More strict features	mauricedb committed 6 days ago	66c0ee8
Enabling strict mode		e161746



# Clone the GitHub Repository

```
PS C:\Temp> git clone git@github.com:mauricedb/ts-advanced.git
Cloning into 'ts-advanced'...
remote: Enumerating objects: 90, done.
remote: Counting objects: 100% (90/90), done.
remote: Compressing objects: 100% (50/50), done.
Receiving objects: 34% (31/90), 516.00 KiB | 923.00 KiB/sremote: Total 90 (delta 44), reused 84 (delta 38), pack-reused 0
Receiving objects: 100% (90/90), 807.90 KiB | 1.09 MiB/s, done.
Resolving deltas: 100% (44/44), done.
```

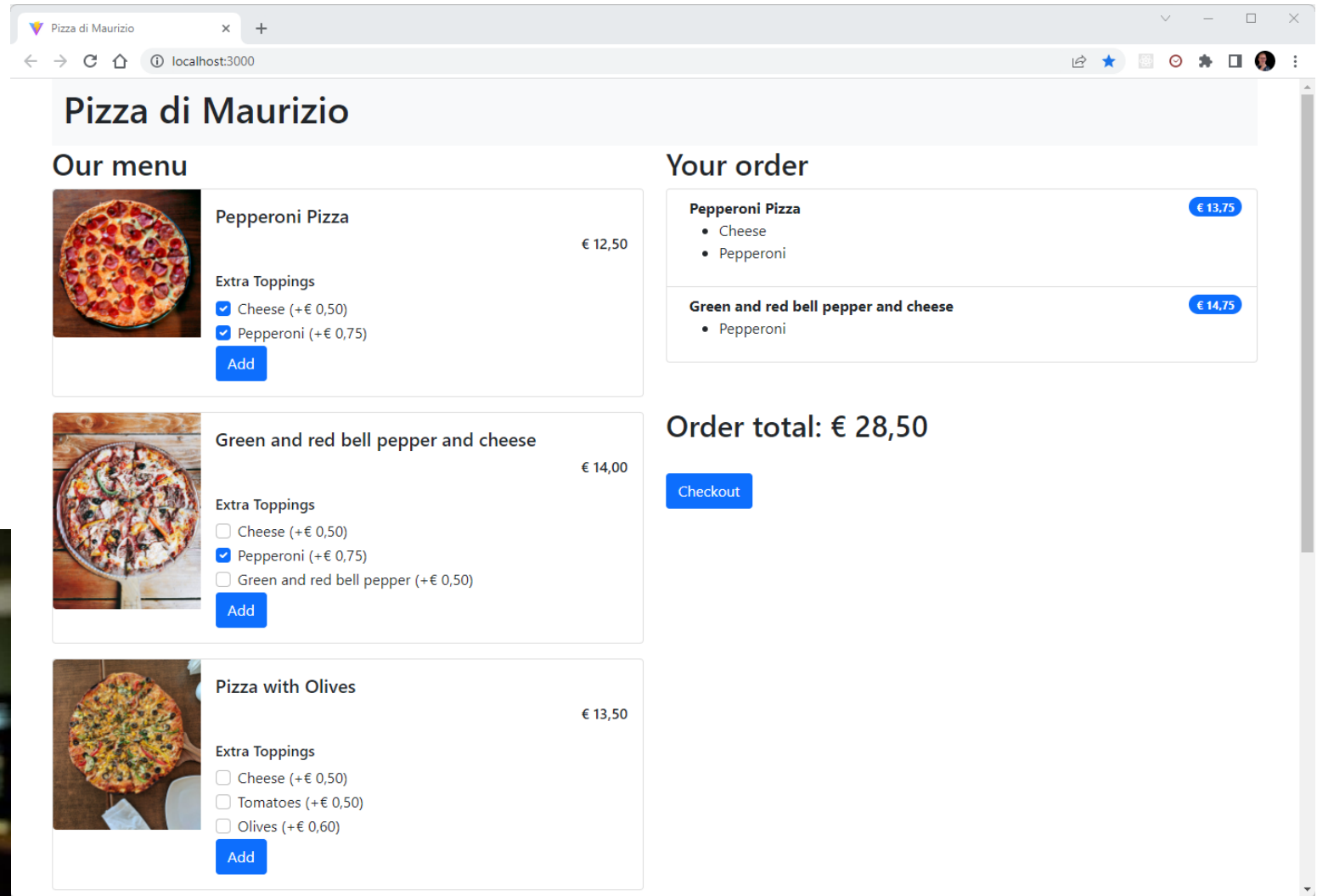
# Install NPM Packages

```
PS C:\Temp> cd .\ts-advanced\  
PS C:\Temp\ts-advanced> npm install  
  
added 18 packages, and audited 19 packages in 2s  
  
7 packages are looking for funding  
  run `npm fund` for details  
  
found 0 vulnerabilities
```

# Start the application

```
PS C:\Temp\ts-advanced> npm run dev  
  
> ts-advanced@0.0.0 dev  
> vite  
  
vite v2.9.5 dev server running at:  
  
> Local: http://localhost:3000/  
> Network: use `--host` to expose  
  
ready in 127ms.
```

# The application





Enabling strict mode

# Strict mode

- Start with the **00-start** branch
- Set the **strict compiler option** to **true**
  - Usually in the tsconfig.json
- The **"?."** **optional chaining operator** helps
  - Only use properties when the parent is defined
- The **"??"** **nullish coalescing operator**
  - Like the "||" or operator but only for "null" and "undefined"
  - Great for default values
- The **"!"** **non-null assertion** operator
  - Removes "null" and "undefined" from a type

# tsconfig.json

```
tsconfig.json M x TS main.ts 6
tsconfig.json > ...
You, 3 minutes ago | 1 author (You)
1 {
2   "compilerOptions": {
3     "esModuleInterop": true,
4     "lib": ["ESNext", "DOM"],
5     "module": "ESNext",
6     "moduleResolution": "Node",
7     "noEmit": true,
8     "resolveJsonModule": true,
9     "sourceMap": true,
10    "strict": true,
11    "target": "ESNext",
12    "useDefineForClassFields": true
13  },
14  "include": ["src"]
15 }
```

# main.ts

```
tsconfig.json M TS main.ts M X
src > TS main.ts > ...
13 function addPizzaToOrder(e: SubmitEvent) {
14     e.preventDefault();
15
16     const form = e.target as HTMLFormElement;
17     const pizza = formToPizzaMap.get(form);
18     if (!pizza) {
19         throw new Error('Could not find pizza');
20     }
21     const formElements = Array.from(form.elements) as HTMLInputElement[];
```



main.ts



```
tsconfig.json M TS main.ts 1, M X
src > TS main.ts > ...
50 function renderOrderTotal(recalculate: boolean) {
51   const totalPriceEl = document.getElementById('order-total');
52   const totalPrice = order.reduce((sum, item) => sum + item.price, 0);
53   totalPriceEl!.innerHTML = formatCurrency(totalPrice);
54 }
55
56 function renderOrder() {
57   const recalculate = true;
58   const orderEl = document.getElementById('order')!;
```



More strict features

# More Strict Features

- There are **many more strict settings** not enabled by “strict”
  - allowUnreachableCode
  - allowUnusedLabels
  - exactOptionalPropertyTypes
  - noFallthroughCasesInSwitch
  - noImplicitOverride
  - noImplicitReturns
  - noPropertyAccessFromIndexSignature
  - noUncheckedIndexedAccess
  - noUnusedLocals
  - noUnusedParameters

# tsconfig.json

```
tsconfig.json M x TS main.ts M
tsconfig.json > ...
You, 4 minutes ago | 1 author (You)
1 {
2   "compilerOptions": {
3     "allowUnreachableCode": false,
4     "allowUnusedLabels": false,
5     "esModuleInterop": true,
6     "exactOptionalPropertyTypes": true,
7     "lib": ["ESNext", "DOM"],
8     "module": "ESNext",
9     "moduleResolution": "Node",
10    "noEmit": true,
11    "noFallthroughCasesInSwitch": true,
12    "noImplicitOverride": true,
13    "noImplicitReturns": true,
14    "noPropertyAccessFromIndexSignature": true,
15    "noUncheckedIndexedAccess": true,
16    "noUnusedLocals": true,
17    "noUnusedParameters": true,
18    "resolveJsonModule": true,
19    "sourceMap": true,
20    "strict": true,
21    "target": "ESNext",
22    "useDefineForClassFields": true
23  },
```

# Unchecked Indexed Access

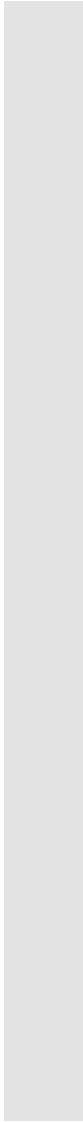

main.ts

```
tsconfig.json M TS main.ts M X
src > TS main.ts > ...
24   const extras = formElements
25     .filter((element) => element.type === 'checkbox' && element.checked)
26     .map((element) => element.value)
27     .map((name) => {
28       const extra = extraIngredients[name];
29       return extra ?? { name, price: 0 };
30     });
31
32   const price = extras.reduce(
33     (sum, extraIngredient) => sum + extraIngredient.price,
34     pizza.price
35   );
```

# Unused Parameters



```
tsconfig.json M TS main.ts M X
src > TS main.ts > ...
54 function renderOrderTotal() {
55   const totalPriceEl = document.getElementById('order-total');
56   const totalPrice = order.reduce((sum, item) => sum + item.price, 0);
57   totalPriceEl!.innerHTML = formatCurrency(totalPrice);
58 }
```



Validating data at the  
boundary

# Validating Data

- The type definitions are **used at compile time**
- They **might not match the runtime** behavior
  - Specially when doing AJAX requests or reading JSON files



# package.json

```
package.json X
package.json > ...
10  "devDependencies": {
11    "typescript": "^4.6.3",
12    "vite": "^2.9.5"
13  },
14  "dependencies": {
15    "bootstrap": "^5.1.3",
16    "zod": "^3.14.4"
17  }
```

main.ts

```
TS schemas.ts U x TS main.ts M
src > TS schemas.ts > ...
1  import { z } from 'zod';
2
3  export const pizzaSchema = z.object({
4    name: z.string().nonempty(),
5    price: z.number().positive(),
6    imageUrl: z.string(),
7    imageCredit: z.string(),
8    extras: z.array(z.string()),
9  });
10
11 export const pizzaArraySchema = z.array(pizzaSchema);
12
13 export const extraIngredientSchema = z.object({
14   name: z.string().nonempty(),
15   price: z.number().nonnegative(),
16 });
17
18 export const extraIngredientsSchema = z.record(extraIngredientSchema);
```

main.ts



```
TS schemas.ts U TS main.ts M X
src > TS main.ts > ...
148  async function loadPizzas(): Promise<Pizza[]> {
149      const rsp = await fetch('/api/pizzas.json');
150      const data = await rsp.json();
151      return pizzaArraySchema.parse(data);
152  }
153
154  async function loadExtras(): Promise<ExtraIngredients> {
155      const rsp = await fetch('/api/extra-ingredients.json');
156      const data = await rsp.json();
157      return extraIngredientsSchema.parse(data);
158  }
```



# Inferring TypeScript types

# Inferring Types

- Maintaining a **Zod schema** and a **TypeScript interface** is tedious
  - Both have to be kept in sync
- The TypeScript types can be **inferred** from the Zod schema
  - Using “z.infer(typeof schema)”

types.ts



```
TS types.ts M X
src > TS types.ts > ...
You, 24 seconds ago | 1 author (You)
1 import { z } from 'zod';
2 import {
3   extraIngredientSchema,
4   extraIngredientsSchema,
5   pizzaSchema,
6 } from './schemas';
7
8 export type Pizza = z.infer<typeof pizzaSchema>;
9
10 export type PizzaArray = Pizza[];
11
12 export type ExtraIngredient = z.infer<typeof extraIngredientSchema>;
13
14 export type ExtraIngredients = z.infer<typeof extraIngredientsSchema>;
```



Unknown In Catch

# Unknown Catch

- In ECMAScript **any type of variable can be thrown**
  - The default for the variable in the catch is “any”
- With **useUnknownInCatchVariables** set to true it will be “unknown”
  - Use a type guard to check the actual type
  - Can still be explicitly defined as “any” when needed



main.ts



```
TS main.ts M x
src > TS main.ts > ...

168 async function init() {
169   try {
170     renderOrder();
171     const pizzasPromise = loadPizzas();
172     const extrasPromise = loadExtras();
173     renderMenu(await pizzasPromise, await extrasPromise);
174
175     document
176       .getElementById('checkout-button')
177       ?.addEventListener('click', () => {
178         const account = 1234567890;
179         const amount = order.reduce((sum, item) => sum + item.price, 0);
180
181         checkout(account, amount);
182       });
183   } catch (error) {
184     if (error instanceof Error) {
185       console.error(
186         `%cError loading data: ${error.name} - ${error.message}`,
187         'font-weight: bold; font-size: 1.5rem;'
188       );
189     } else {
190       console.error(error);
191     }
192   }
193 }
```



# Mapped Types

# Mapped Types

- Mapped types are **very flexible and powerful**
- Many **build in mapped types**
  - Readonly<Type>
  - Omit<Type, Keys>
  - Pick<Type, Keys>
  - Partial<Type>
- Easy to **create custom mapped types**
  - Zod infer<typeof schema>
  - DeepReadonly<Type>



Using Readonly<T>

# Readonly<T>

- The Readonly<T> mapped type **creates a read-only mapped type**
  - Can't change properties anymore
  - Or use "array.push()" etc.
- ⚠️ Readonly<T> is **not recursive** ⚠️
  - Only the first level of properties becomes read-only
- 💡 Recommended for **function arguments** to show intent 💡
  - And AJAX responses etc.

# types.ts

```
TS main.ts M TS types.ts M X
src > TS types.ts > ...
You, 2 minutes ago | 1 author (You)
1 import { z } from 'zod';
2 import {
3   extraIngredientSchema,
4   extraIngredientsSchema,
5   pizzaSchema,
6 } from './schemas';
7
8 export type Pizza = Readonly<z.infer<typeof pizzaSchema>>;
9
10 export type PizzaArray = ReadonlyArray<Pizza>;
11
12 export type ExtraIngredient = Readonly<z.infer<typeof extraIngredientSchema>>;
13
14 export type ExtraIngredients = Readonly<z.infer<typeof extraIngredientsSchema>>;
```

main.ts

```
TS main.ts M x TS types.ts M
src > TS main.ts > ...
93 function renderMenu(pizzas: PizzaArray, extras: ExtraIngredients) {
94     extraIngredients = extras;
95
96     const main = document.getElementById('menu');
97
98     for (const pizza of pizzas) {
99         const extraToppings = pizza.extras
100             .map((key) => {
101                 const id = crypto.randomUUID();
102                 const extraTopping = extras[key] ?? { name: key, price: 0 };
            })
    }
```

main.ts



```
TS main.ts M x TS types.ts M
src > TS main.ts > ...
44   order.push(itemOrdered);
45
46   itemOrdered.extras.forEach((extra) => {
47     if (extra.price === 1) {
48       // console.log(`${extra.name} has a price of € 1`);
49     }
50   });
51
52   renderOrder();
53 }
```





DeepReadOnly<T>

# DeepReadonly<T>

- Make a whole **nested object structure read-only**
  - Recursive mapped types are very powerful
  - An improvement over the default Readonly<T>
- Source:  
<https://gist.github.com/basarat/1c2923f91643a16agode638e76bceoab>

types.ts



```
TS types.ts M X
src > TS types.ts > ...
1 import { z } from 'zod';
2 import {
3   extraIngredientSchema,
4   extraIngredientsSchema,
5   pizzaSchema,
6 } from './schemas';
7
8 type DeepReadonly<T> = {
9   readonly [P in keyof T]: DeepReadonly<T[P]>;
10 };
11
12 export type Pizza = DeepReadonly<z.infer<typeof pizzaSchema>>;
13 export type PizzaArray = DeepReadonly<Pizza[]>;
14
15 export type ExtraIngredient = DeepReadonly<
16   z.infer<typeof extraIngredientSchema>
17 >;
18
19 export type ExtraIngredients = DeepReadonly<
20   z.infer<typeof extraIngredientsSchema>
21 >;
```



# Indexed Access Types

# Indexed Access Types

- Sometimes you want to **access the type of a specific property**
  - To avoid manually duplicating the type
  - But the type is not exposed
- Some **other very useful mapped types**:
  - `Parameters< typeof someFunction >`
  - `ReturnType< typeof someFunction >`
  - `ConstructorParameters< T >`

types.ts



```
TS types.ts M x
src > TS types.ts > ...
23 export interface ItemOrdered {
24     name: Pizza['name'];
25     price: Pizza['price'];
26     extras: ExtraIngredient[];
27 }
```



# Type Mapping with Omit<> and Pick<>

# Omit<> Pick<>

- Use Omit<T> and Pick<T> to **build custom types based on others**
  - Pick<T> lets you specify all the properties you want to copy
  - Omit<T> lets you take all properties except the listed ones

```
// type NamePrice2 = { name: string; price: number; }  
type NamePrice = Pick<Pizza, 'name' | 'price'>;
```

- 💡 Use the **Exclude<T>** and **Extract<T>** to mutate types 💡

```
type StringOrNumber = string | number;  
// type AlwaysString = string  
type AlwaysString = Exclude<StringOrNumber, number>;
```



types.ts



```
TS types.ts M X
src > TS types.ts > ...
23 // export type ItemOrdered = Omit<Pizza, 'imageUrl' | 'imageCredit' | 'extras'> & {
24 //   extras: ExtraIngredient[];
25 // };
26
• 27 export type ItemOrdered = Pick<Pizza, 'name' | 'price'> & {
28   extras: ExtraIngredient[];
29 };
```



# Displaying Types

# Displaying Types

- A **disadvantage of mapped types** is that the type definition in tooltips becomes **hard to read**
  - It shows how a type is constructed instead of the resulting type
- The `Resolve<T>` turns this into **the resulting type** instead
  - Source:  
<https://effectivetypescript.com/2022/02/25/gentips-4-display/>

main.ts

```
const extra = e (alias) type ItemOrdered = Pick<DeepReadonly<{  
  return extra ??  
});  
name: string;  
price: number;  
imageUrl: string;  
imageCredit: string;  
extras: string[];  
const price = extra (sum, extraIngredient pizza.price }>, "name" | "price"> & {  
  extras: ExtraIngredient[];  
  }  
);  
const itemOrdered: ItemOrdered = {
```

types.ts

```
TS main.ts TS types.ts M X
src > TS types.ts > ...
28 // Taken from https://effectivetypescript.com/2022/02/25/gentips-4-display/
29 type Resolve<T> = T extends Function ? T : { [K in keyof T]: T[K] };
30
31 export type ItemOrdered = Resolve<
32   Pick<Pizza, 'name' | 'price'> & {
33     readonly extras: ExtraIngredient[];
34   }
35 >;
```

main.ts



```
4 import { ExtraIngredients, ItemOrdered, Pizza, PizzaArray } from './types';
5 (alias) type ItemOrdered = {
6   readonly name: string;
7   readonly price: number;
8   readonly extras: ExtraIngredient[];
9 }
10 You, 1 import ItemOrdered
11 const order: ItemOrdered[] = [];
```



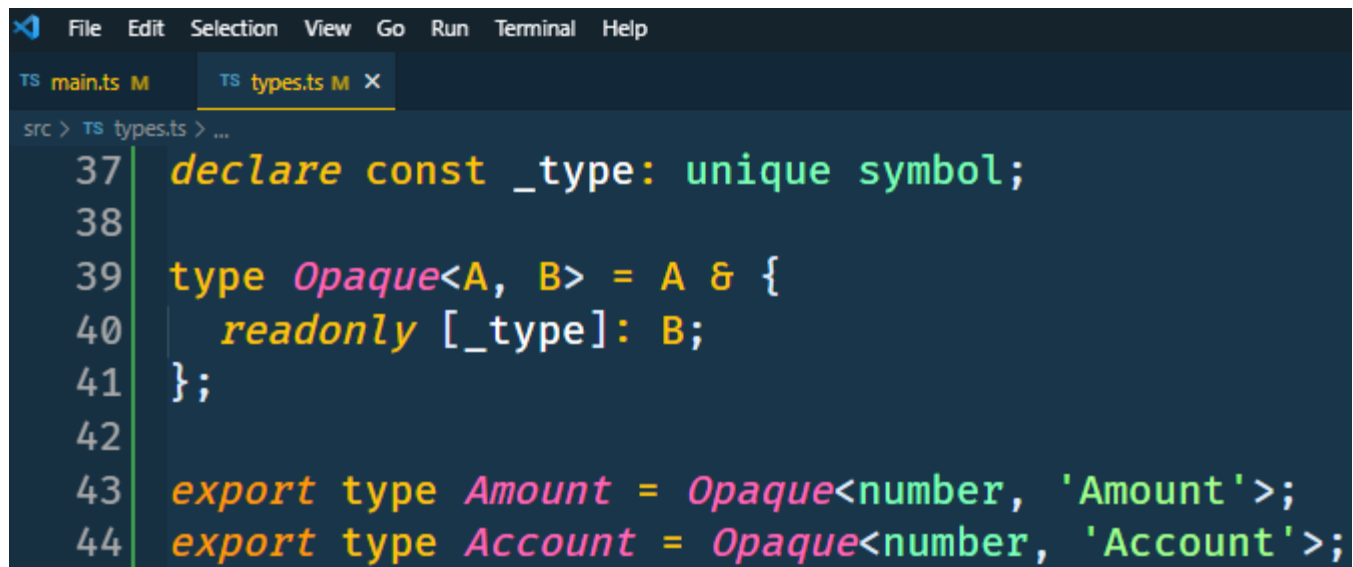
# Opaque Types

# Opaque Types

- A lot of business data ultimately end up as a **primitive data type**
  - They are all modeled as string, number etc.
- The **compiler doesn't know the difference** between them
  - A PO box number and invoice total amount are both type "number"
    - The same for the compiler
    - Very different for the business case
- **Opaque types can make it easier to reason about code**
  - By providing distinct types and a clear separation



types.ts



The image shows a screenshot of a Visual Studio Code editor window. The title bar at the top includes the VS Code logo and menu items: File, Edit, Selection, View, Go, Run, Terminal, and Help. Below the title bar, there are two tabs: 'TS main.ts M' and 'TS types.ts M X', with the latter being the active tab. The editor content shows the following TypeScript code:

```
src > TS types.ts > ...
37 declare const _type: unique symbol;
38
39 type Opaque<A, B> = A & {
40   readonly [_type]: B;
41 };
42
43 export type Amount = Opaque<number, 'Amount'>;
44 export type Account = Opaque<number, 'Account'>;
```

main.ts

```
TS main.ts M X TS types.ts M
src > TS main.ts > ...
156 function checkout(amount: Amount, account: Account) {
157     document.getElementById('checkout-amount')!.innerText =
158         formatCurrency(amount);
159     document.getElementById('checkout-account')!.innerText =
160         account.toLocaleString('nl-NL');
161     (document.getElementById('checkout-dialog') as any)?.showModal();
162 }
```

main.ts



```
TS main.ts M x TS types.ts M
src > TS main.ts > ...
182     document
183     .getElementById('checkout-button')
184     ?.addEventListener('click', () => {
185         const account = 1234567890 as Account;
186         const amount = order.reduce(
187             (sum, item) => sum + item.price,
188             0
189         ) as Amount;
190
191         checkout(amount, account);
192     });
```



# Type Predicate Functions

# Type Predicate Functions

- Often a **TypeScript cast** is used when types don't quite line up
  - But that is **just silencing the compiler**
    - ⚠ Casting via "unknown" will even allow any (invalid) type cast ⚠
  - There is no runtime checking or guarantee
- A **type predicate** can do a cast in a runtime safe manner
  - 💡 Checks **both at runtime and compile time** 💡
  - A **function that returns a "boolean"** to indicate if the type matches

types.ts

```
TS main.ts M TS types.ts M X
src > TS types.ts > ...
43 export type Amount = Opaque<number, 'Amount'>;
44 export type Account = Opaque<number, 'Account'>;
45
46 export function isAccount(value: unknown): value is Account {
47     return typeof value === 'number' && value.toString().length === 10;
48 }
```

main.ts



```
TS main.ts M x TS types.ts M
src > TS main.ts > ...
183     document
184     .getElementById('checkout-button')
185     ?.addEventListener('click', () => {
186         const account = 1234567890;
187         const amount = order.reduce(
188             (sum, item) => sum + item.price,
189             0
190         ) as Amount;
191
192         if (isAccount(account)) {
193             checkout(amount, account);
194         } else {
195             throw new Error('Account is not valid');
196         }
197     });
```



# Type Assertion Functions



# Type Assertion Functions

- **Type assertion functions** can be even easier
  - Throw an error if the type doesn't match
- Often a **better alternative** then a cast
  - The code will not continue if the assumption is wrong

# types.ts

```
TS main.ts 1, M TS types.ts M X
src > TS types.ts > ...
43 export type Amount = Opaque<number, 'Amount'>;
44 export type Account = Opaque<number, 'Account'>;
45
46 export function isAccount(value: unknown): value is Account {
47     return typeof value === 'number' && value.toString().length === 10;
48 }
49
50 export function assertAccount(value: unknown): asserts value is Account {
51     if (!isAccount(value)) {
52         throw new Error(`Expected account, got ${value}`);
53     }
54 }
55
56 export function assertAmount(value: unknown): asserts value is Amount {
57     if (typeof value !== 'number') {
58         throw new Error(`Expected amount, got ${value}`);
59     }
60 }
```

main.ts



```
TS main.ts 1, M x TS types.ts M
src > TS main.ts > ...
185     document
186     .getElementById('checkout-button')
187     ?.addEventListener('click', () => {
188         const account = 1234567890;
189         const amount = order.reduce((sum, item) => sum + item.price, 0);
190
191         assertAmount(amount);
192         assertAccount(account);
193
194         checkout(amount, account);
195     });
```



# Exhaustiveness Checking

# Exhaustiveness Checking

- The TypeScript compiler doesn't tell us if every case is provided
  - It's easy to forget to add a switch case when an enumeration is expanded
- The “**never**” type is a great way to make sure
  - A compile error if the default case can be reached
  - 💡 Make sure to add an exception or error logging at runtime 💡

# exhaustive.ts



```
TS exhaustive.ts U x
src > TS exhaustive.ts > ...
1  export enum Animal {
2      Dog = 'dog',
3      Cat = 'cat',
4      Bird = 'bird',
5  }
6
7  export function assertNever(value: never): never {
8      throw new Error('Unexpected value: ' + value);
9  }
10
11 export function feedAnimal(animal: Animal) {
12     switch (animal) {
13         case Animal.Dog:
14             console.log('The dog eats meat');
15             break;
16         case Animal.Cat:
17             console.log('The cat eats fish');
18             break;
19         case Animal.Bird:
20             console.log('The bird eats seeds');
21             break;
22         default:
23             assertNever(animal);
24     }
25 }
```

# Conclusion

- TypeScript's **strict settings** help catch many errors
  - Make sure to turn on the additional strict features as well
- **Validate all data at boundaries**
  - Not just from the user, also from API's
- Use **type predicates and assertions** both at compile and run-time
  - Instead of just casting at compile time
- Use **mapped types** to create new types
  - The possibilities are almost endless
- Enable **exhaustiveness checking** with the "never" type
  - Make sure to log unexpected cases at runtime

Maurice de Beijer

@mauricedb

maurice.de.beijer  
@gmail.com

