Top 5 techniques in TypeScript to bring your code to the next level.

Write better code with refactoring and functional approaches.

Oleh Baranovskyi 3 days ago · 4 min read

With time and practice, we all deliver code with better quality. However, there is always room for improvement. Whenever I look at the code that I wrote half of the year ago, and I don't know how to improve it, I think about two things. Either I didn't grow, or it is already in good shape. If you're like me, and the code quality is essential to you, then that makes two of us. Most likely, this article will discover at least a few new techniques you didn't know and might use on a regular basis.

. . .

Topics to cover:

- Better validations with `includes` and selector function
- Use callbacks to encapsulate code that changes
- Consider using predicate combinators
- Even better predicate combinators with factories
- Encapsulate algorithm in the new class

. . .

Better validations with `includes` and selector function

Problem:

We have a function that compares the same object property with multiple values of the same type and returns `true` in case if at least one statement is positive.

```
enum UserStatus {
  Administrator = 1,
  Author = 2,
  Contributor = 3,
  Editor = 4,
  Subscriber = 5,
}

interface User {
  firstName: string;
  lastName: string;
  status: UserStatus;
}

function isEditActionAvailable(user: User): boolean {
  return (
    user.status === UserStatus.Administrator ||
    user.status === UserStatus.Author ||
    user.status === UserStatus.Editor
  );
}
```

cr-same-value-checks.ts hosted with ♥ by GitHub view raw

Solution:

We can use an array method `includes`. With this approach, an if statement will look cleaner than before.

```
const EDIT_ROLES = [
  UserStatus.Administrator,
  UserStatus.Author,
  UserStatus.Editor,
```

```
];

function isEditActionAvailable(user: User): boolean {
  return EDIT_ROLES.includes(user.status);
}
```

cr-same-value-better.ts hosted with by GitHub view raw

But there is a catch.

First, we have hard-coded data inside the function, or to put it in another way we have an implicit input (`EDIT_ROLES`).

Second, what if we want to make another role guard function that will be checking different action roles?

We can provide a factory function that will take a selector function and data responsible for describing a user role. The function itself returns another function that takes a user and compares his status with the previously passed role list.

```
function roleCheck<D, T>(selector: (data: D) => T, roles: T[]): (value: D) => boolean {
    return (value: D) => roles.includes(selector(value));
}

const isEditActionAvailable = roleCheck((user: User) => user.status, EDIT_ROLES);
```

first-value-even-better.ts hosted with by GitHub view raw

In this way, we've split the data from the function, what is good from the functional standpoint, and made it reusable.

Here is an example of how easy we can add another role guard function:

```
const ADD_ROLES = [
  UserStatus.Administrator,
  UserStatus.Author
];

const isAddActionAvailable = roleCheck((user: User) => user.status, ADD_ROLES);
```

cr-same-value-other-action.ts hosted with by GitHub view raw

But wait a minute. You probably think about the selector function. Why do we need it?

With the help of the selector function, it's possible to select different fields.

Suppose a user has a team status role field, and we have to check whether he is a team lead or manager. It's effortless to develop a guard function that follows mentioned requirements by using an already implemented function.

```
// ...

enum TeamStatus {
    Lead = 1,
    Manager = 2,
    Developer = 3
}

interface User {
  firstName: string;
  lastName: string;
  status: UserStatus;
```

```
  teamStatus: TeamStatus;
}


function roleCheck<D, T>(selector: (data: D) => T, roles: T[]): (value: D) => boolean {
    return (value: D) => roles.includes(selector(value));
}

const MANAGER_OR_LEAD = [
    TeamStatus.Lead,
    TeamStatus.Manager
]

const isManagerOrLead = roleCheck((user: User) => user.teamStatus, MANAGER_OR_LEAD);
```

cr-same-value-selector.ts hosted with by GitHub view raw

. . .

Use callbacks to encapsulate code that changes

Problem:

We have multiple functions that are pretty similar, with a minor difference. So it would be great to get rid of the duplicated code.

```
async function createUser(user: User): Promise<void> {
  LoadingService.startLoading();
  await userHttpClient.createUser(user);
  LoadingService.stopLoading();
  UserGrid.reloadData();
}

async function updateUser(user: User): Promise<void> {
  LoadingService.startLoading();
  await userHttpClient.updateUser(user);
  LoadingService.stopLoading();
```

```
    UserGrid.reloadData();
}
```

cr-encapsulate-changes.ts hosted with by GitHub view raw

Solution:

We can extract the code that changes and pass it through the callback. In such a manner, we'll remove duplicate code.

```
async function makeUserAction(fn: Function): Promise<void> {
  LoadingService.startLoading();
  await fn();
  LoadingService.stopLoading();
  UserGrid.reloadData();
}

async function createUser2(user: User): Promise<void> {
  makeUserAction(() => userHttpClient.createUser(user));
}

async function updateUser2(user: User): Promise<void> {
  makeUserAction(() => userHttpClient.updateUser(user));
}
```

cr-encapsulate-changes-refactored.ts hosted with by GitHub view raw

. . .

Consider using predicate combinators

Problem:

Our predicate functions are checking too much and are in charge of more than should.

```ts
enum UserRole {
  Administrator = 1,
  Editor = 2,
  Subscriber = 3,
  Writer = 4,
}

interface User {
  username: string;
  age: number;
  role: UserRole;
}

const users = [
  { username: "John", age: 25, role: UserRole.Administrator },
  { username: "Jane", age: 7, role: UserRole.Subscriber },
  { username: "Liza", age: 18, role: UserRole.Writer },
  { username: "Jim", age: 16, role: UserRole.Editor },
  { username: "Bill", age: 32, role: UserRole.Editor },
];

const greaterThen17AndWriterOrEditor = users.filter((user: User) => {
  return (
    user.age > 17 &&
    (user.role === UserRole.Writer || user.role === UserRole.Editor)
  );
});

const greaterThen5AndSubscriberOrWriter = users.filter((user: User) => {
    return user.age > 5 && user.role === UserRole.Writer;
});
```

cr-predicate-combinators-bad.ts hosted with by GitHub view raw

Solution:

We have to start using the predicate combinators. It will increase code readability and reusability.

```ts
type PredicateFn = (value: any, index?: number) => boolean;
```

```
type ProjectionFn = (value: any, index?: number) => any;

function or(...predicates: PredicateFn[]): PredicateFn {
  return (value) => predicates.some((predicate) => predicate(value));
}

function and(...predicates: PredicateFn[]): PredicateFn {
  return (value) => predicates.every((predicate) => predicate(value));
}

function not(...predicates: PredicateFn[]): PredicateFn {
  return (value) => predicates.every((predicate) => !predicate(value));
}
```

cr-combinator-predicates.ts hosted with by GitHub view raw

Let's take a look at the combinator predicates in action:

```
const isWriter = (user: User) => user.role === UserRole.Writer;
const isEditor = (user: User) => user.role === UserRole.Editor;
const isGreaterThan17 = (user: User) => user.age > 17;
const isGreaterThan5 = (user: User) => user.age > 5;

const greaterThan17AndWriterOrEditor = users.filter(
    and(isGreaterThan17, or(isWriter, isEditor))
);

const greaterThan5AndSubscriberOrWriter = users.filter(
    and(isGreaterThan5, isWriter)
);
```

cr-combinator-predicates-result.ts hosted with by GitHub view raw

. . .

Even better predicate combinators with factories

Problem:

Predicate combinators create too many variables, so it's easy to get lost between these functions. If we use the combinator predicate function only once, then it's better to have something more generic.

```
const isWriter = (user: User) => user.role === UserRole.Writer;
const isEditor = (user: User) => user.role === UserRole.Editor;
const isGreaterThan17 = (user: User) => user.age > 17;
const isGreaterThan5 = (user: User) => user.age > 5;

const greaterThan17AndWriterOrEditor = users.filter(
    and(isGreaterThan17, or(isWriter, isEditor))
);

const greaterThan5AndSubscriberOrWriter = users.filter(
    and(isGreaterThan5, isWriter)
);
```

cr-combinator-predicates-result.ts hosted with by GitHub view raw

Solution:

We have to start using the combinator predicate factories. Let's add a few:

```
const isRole = (role: UserRole) =>
    (user: User) => user.role === role;

const isGreaterThan = (age: number) =>
    (user: User) => user.role === age;


const greaterThan17AndWriterOrEditor = users.filter(
    and(isGreaterThan(17), or(isRole(UserRole.Writer), isRole(UserRole.Editor)))
);

const greaterThan5AndSubscriberOrWriter = users.filter(
    and(isGreaterThan(5), isRole(UserRole.Writer))
```

```
14  );
```

cr-combinator-predicate-factories.ts hosted with ♡ by GitHub view raw

You've probably noticed that some function invocation is repeated with the same arguments. The best option will be to mix predicate factories with combinator predicates together. Thus we'll have the best of both worlds.

```
1   const isRole = (role: UserRole) =>
2       (user: User) => user.role === role;
3
4   const isGreaterThan = (age: number) =>
5       (user: User) => user.role === age;
6
7   const isWriter = isRole(UserRole.Writer)
8
9   const greaterThan17AndWriterOrEditor = users.filter(
10      and(isGreaterThan(17), or(isWriter, isRole(UserRole.Editor)))
11  );
12
13  const greaterThan5AndSubscriberOrWriter = users.filter(
14      and(isGreaterThan(5), isWriter)
15  );
```

cr-predicates-and-predicate-factories.ts hosted with ♡ by GitHub view raw

In this way, we do fewer repeats and keep the code clean and neat.

. . .

Encapsulate algorithm in the new class

Problem:

We have a class responsible for too many things. It's bound with algorithm logic, but it shouldn't.

```
class User {
  constructor(
    public firstName: string,
    public lastName: string,
    public signUpDate: Date
  ) {}

  getFormattedUserDetails(): string {
    const formattedSignUpDate = `${this.signUpDate.getFullYear()}-${this.signUpDate.getMonth() +
    const username = `${this.firstName.charAt(0)}${this.lastName}`.toLowerCase();

    return `
        First name: ${this.firstName},
        Last name: ${this.lastName},
        Sign up date: ${formattedSignUpDate},
        Username: ${username}
    `;
  }
}

const user = new User("John", "Doe", new Date());
console.log(user.getFormattedUserDetails());
```

cr-encapsulate-algorithm.ts hosted with ♥ by GitHub view raw

Solution:

It's fair to say this method shouldn't reside in the user data model. Therefore, our mission is to split the responsibility. We have to extract the algorithm and encapsulate it in the new class to do this.

```
interface User {
```

```
    firstName: string,
    lastName: string,
    signUpDate: Date
}

class UserDetailsFormatter {
  constructor(private user: User) {}

  format(): string {
    const { firstName, lastName } = this.user;

    return `
        First name: ${firstName},
        Last name: ${lastName},
        Sign up date: ${this.getFormattedSignUpDate()},
        Username: ${this.getUsername()}
    `;
  }

  private getUsername(): string {
    const { firstName, lastName } = this.user;

    return `${firstName.charAt(0)}${lastName}`.toLowerCase();
  }

  private getFormattedSignUpDate(): string {
    const signUpDate = this.user.signUpDate;

    return [
      signUpDate.getFullYear(),
      signUpDate.getMonth() + 1,
      signUpDate.getDate(),
    ].join("-");
  }
}

const user = { firstName: "John", lastName: "Doe", signUpDate: new Date() };
const userFormatter = new UserDetailsFormatter(user);
console.log(userFormatter.format());
```

cr-encapsulate-algorithm-right-way.ts hosted with ♡ by GitHub view raw

. . .

Further reading

If you liked the article, you might find these articles interesting as well:

TypeScript: Understand Built-In/Utility Types

Nowadays TypeScript becomes more and more popular. Many people are using it for different purposes and not only for...

obaranovskyi.medium.com

Code Refactoring: Polymorphism instead of the switch and other conditionals.

Purpose of this article is to explain why switch statements are bad and how to avoid them by using polymorphic nature...

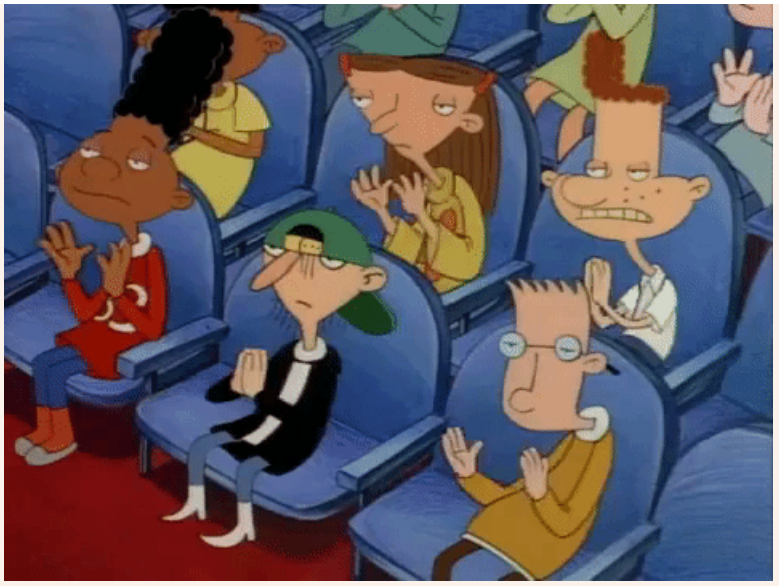
obaranovskyi.medium.com

. . .

Conclusion

I hope you found this article interesting and came across new techniques.

Please feel free to reach out if you have any questions.



Hit the clap button and subscribe, which will motivate me to write more articles.

Typescript Frontend JavaScript Web Development Refactoring