**What Is the Open-Closed Principle?**

*‘Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.’”*

This principle tells you to write code in such a way that you will be able to add additional functionality without changing the existing code.

Let’s see where we can apply this principle.

**Let’s Start With an Example**

Say we have a User component where we pass a user's details and the main purpose of this class is to show the details of that particular user.

This is simple enough to start with. But our life is not so simple. After a few days, our manager tells us that there are three types of users in our system: SuperAdmin, Admin, etc.

And each of them will have different information and functionalities.

**A Bad Solution**

Well, the first and obvious solution is to have a conditional inside our component and render different information based on the different user types.

User.js

Do you see what’s wrong here?

Firstly, our code is messy now.

Secondly, what if we need another type of user? We would then need to go into User.js and add another condition for that particular type of user.

This is a *clear violation* of the Open-Closed principle because we are not allowed to alter the code inside the User component.

**What’s The Solution?**

OK, so there are two main techniques that we can apply in this scenario:

1. Higher-order component
2. Component composition

It’s better to go the second route whenever possible, but there can be cases where using a HOC is necessary.

For now, we will use a technique recommended by Facebook that is called the [composition of components](https://reactjs.org/docs/composition-vs-inheritance.html).

**Let’s Create Separate User Components**

Now we need to design our code in such a way that we don’t need to add a conditional inside the User.js component. Let’s create a separate component for SuperAdmin:

SuperAdmin.js

Similarly, another one for Admin users:

Admin.js

And now our App.js file becomes

App.js

Now we can create as many user types as we need. Our logic for particular users is encapsulated and we don’t need to revisit our code for any additional modifications.

Some might argue we are increasing the number of files unnecessarily.  
Sure, you can leave it as-is for now, but you will definitely feel the pain as the complexity of the application grows.

**Caution**

SOLID is a set of principles. They are not mandatory for you to apply in every scenario. As a seasoned developer, you should find a good balance between code length and readability.

Don’t obsess too much over these principles. In fact, there is a famous phrase to explain these scenarios:

“Too Much SOLID.”

So knowing these principles is good, but you have to keep a balance. You may not need these compositions for one or two extra fields, but keeping them separate will definitely help in the long run.

**Conclusion**

Knowing these principles will take you a long way because at the end of the day, a good piece of code is what matters and there is no single way of doing things.