

The problems when classes aren't properly encapsulated

Allowing direct access to data on an object can bypass checks and operations.

It encourages an interdependency or coupling between the calling code and the class.

For the previous example, I showed that changing a field name broke the calling code.

And I also showed that the calling code had to take on the responsibility for properly initializing a new Player.

Benefits of Encapsulation

That's really one of the huge benefits of encapsulation. Changes made do not affect any other code.

It's like a black box in many ways.

But the enhanced player class has full control over its data.

Staying in Control

This is why you want to use encapsulation.

We protect the members of the class and some methods from external access.

This prevents calling code from bypassing the rules and constraints we've built into the class.

When I create a new instance, it's initialized with valid data.

But likewise, I'm also making sure that there's no direct access to the fields.

That's why you want to always use encapsulation.

It's something that you should really get used to and use.

Encapsulation Principles

To create an encapsulated class, you want to:

- Create constructors for object initialization, which enforces that only objects with valid data will get created.
- Use the private access modifier for your fields.
- Use setter methods sparingly and only as needed.
- Use access modifiers that aren't private, only for the methods that the calling code needs to use.