**Java Encapsulation**

*Why do we need encapsulation?*

We need encapsulation in order to be able to **maintain the code**. Like maintaining your car. Periodically, you change car oil, air filters or wipers. So that your car performs better and lasts longer. Imagine if changing that humble windshield wiper could break down your car. Aww! What a nightmare if that were to happen. You wouldn’t trust that car any moment further.

Similarly, in order that, the java code performs better and lasts longer without breaking your entire application or other classes accessing your class, you need **maintainability** **and** **flexibility** to change the code.

**Maintainability and flexibility** are not inbuilt in any coding language. You need to design your code in such a way that it inherits these features.

*How do you achieve it?*

In Java, you achieve this by **restricting access to the data variables** in your class. In your public classes, declare all variables to be private (or protected) and provide public methods (**accessor methods**) to act on these variables. Doing so, you control how the data will be accessed and also determine how the code will behave, at the same time providing a consistent interface to the data. This gives you the flexibility to improve and maintain the code, and stops other classes accessing this code from breaking it.

*So, what is encapsulation in Java then?*

Encapsulation is how you structure your Java class. The class definition, where you declare you data variables, and provide methods that act on these data variables.



Photo by [julien Tromeur](https://unsplash.com/@julientromeur?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral)

Simply put, imagine it to be a capsule that contains the data variables and the methods that allow anyone with this capsule to access and work on that data.

*What else do I need to know?*

When one class extends another class, it violates encapsulation. When a subclass inherits a super class, it allows the encapsulation of the subclass to be breached, without event modifying the subclass. If you modify a method in super class in your next release, a subclass depending on this method, might break, without you even touching this subclass.

*How?*

Say in your next release, you decide to create a new method in superclass. However, you already have a method with same signature but different return type in your subclass. Puff, your subclass will not even compile now.

*What should be done then, avoid inheritance?*

Technically Yes. You should avoid extending one class from another unless you are 100% sure that your **subclass is-a type of your super class**. Say a class B should extend class A only if **every class B is really class A.**

*What else?*

Serialization limits encapsulation. As mentioned above encapsulation is needed for maintenance and evolution of your class without breaking the ecosystem around it. However, if default serialization is a norm and its later decided to change the class, JVM’s might use different version of serialization for serializing and deserializing the same class.