**<https://www.geeksforgeeks.org/oops-object-oriented-design/>**

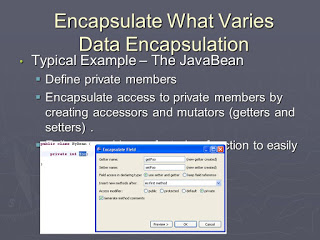
**object-oriented-design-principles**

**DRY (Don't repeat yourself)**

Our first object-oriented design principle is DRY, as the name suggests **DRY (don't repeat yourself)** means don't write duplicate code, instead use [Abstraction](http://javarevisited.blogspot.com/2010/10/abstraction-in-java.html) to abstract common things in one place. If you have a block of code in more than two places consider making it a separate method, or if you use a hard-coded value more than one time make them [public final constant](http://javarevisited.blogspot.com/2011/12/final-variable-method-class-java.html).  
  
The benefit of this Object oriented design principle is in maintenance. It's important not to abuse it, duplication is not for code, but for functionality. It means if you used common code to validate OrderID and SSN it doesn’t mean they are same or they will remain same in future.

### By using common code for two different functionality or thing you closely couple them forever and when your OrderId changes its format, your SSN validation code will break. 2. Encapsulate What Changes

Only one thing is constant in the software field and that is "Change", So encapsulate the code you expect or suspect to be changed in future. The benefit of this OOP Design principle is that It's easy to test and maintain proper encapsulated code.



Several of the **design patterns in Java** uses Encapsulation, the [Factory design pattern](http://javarevisited.blogspot.com/2011/12/factory-design-pattern-java-example.html) is one example of Encapsulation which encapsulates object creation code and provides flexibility to introduce a new product later with no impact on existing code.

**3. Open Closed Design Principle**

Classes, methods or functions should be Open for extension (new functionality) and Closed for modification. This is another beautiful SOLID design principle, which prevents someone from changing already tried and tested code.

If you put more than one functionality in one Class in Java it introduces **coupling** between two functionality and even if you change one functionality there is a chance you broke coupled functionality,  which requires another round of testing to avoid any surprise on the production environment.

**5. Dependency Injection or Inversion principle**

Don't ask for dependency it will be provided to you by the framework. This has been very well implemented in [Spring framework](http://www.java67.com/2017/11/top-5-free-core-spring-mvc-courses-learn-online.html), the beauty of this **design principle** is that any class which is injected by DI framework is easy to test with the mock object and easier to maintain because object creation code is centralized in the framework and client code is not littered with that.

**6. Favor Composition over Inheritance**

Always favor composition over inheritance, if possible. Some of you may argue this, but I found that Composition is the lot more flexible than [Inheritance](http://javarevisited.blogspot.sg/2012/10/what-is-inheritance-in-java-and-oops-programming.html).

Composition allows changing the behavior of a class at run-time by setting property during run-time and by using Interfaces to compose a class we use [polymorphism](http://javarevisited.blogspot.com/2011/08/what-is-polymorphism-in-java-example.html) which provides flexibility to replace with better implementation any time.