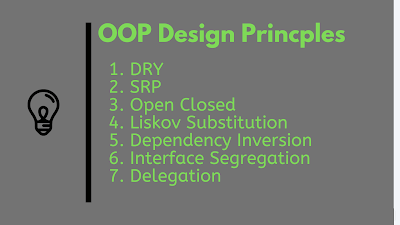
**10 Object-Oriented and SOLID Desing Principles for Programmers**

Though the best way of learning any design principle or pattern is a real-world example and understanding the consequences of violating that design principle, the subject of this article is Introducing *Object-oriented design principles* for Java Programmers, who are either not exposed to it or in the learning phase.

[](https://javarevisited.blogspot.com/2018/02/top-5-java-design-pattern-courses-for-developers.html)

I personally think each of these OOP and SOLID design principle needs an article to explain them clearly, and I will definitely try to do that here, but for now, just get yourself ready for a quick bike ride on design principle town :)

**1. 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 standard code to validate OrderID and SSN, it doesn’t mean they are the same, or they will remain the same in the future.

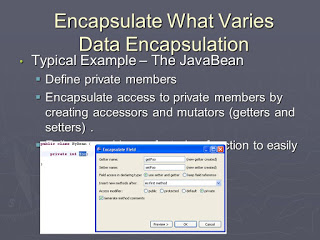
By using standard code for two different functionality or thing, you tightly couple them forever, and when your OrderId changes its format, your SSN validation code will break.  
  
So beware of such coupling and just don’t combine anything which uses similar code but is not related. You can further check out the

[](https://click.linksynergy.com/fs-bin/click?id=JVFxdTr9V80&subid=0&offerid=562016.1&type=10&tmpid=14538&RD_PARM1=https%3A%2F%2Fwww.udemy.com%2Fbasics-of-software-architecture-design-in-java%2F)

**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 the future. The benefit of this OOP Design principle is that It's easy to test and maintain proper encapsulated code.  
  
If you are coding in Java, then follow the principle of [making variables and methods private](http://javarevisited.blogspot.sg/2012/03/private-in-java-why-should-you-always.html#axzz559dyoLSA) by default and increasing access step by step, like from private to protected and not public.

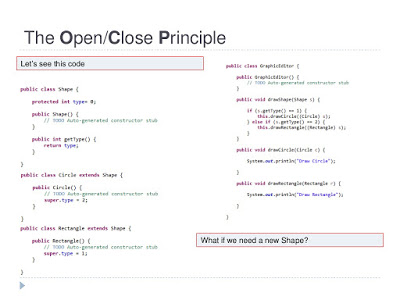
Several of the **design patterns in Java** uses Encapsulation, the [Factory design pattern](http://javarevisited.blogspot.com/2011/12/factory-design-pattern-java-example.html) is one example of Encapsulation that encapsulates object creation code and provides flexibility to introduce a new product later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

[](http://www.java67.com/2018/01/top-10-web-mobile-and-big-data-framework-libraries-technologies-programmers-should-learn-in-2018.html)

**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.

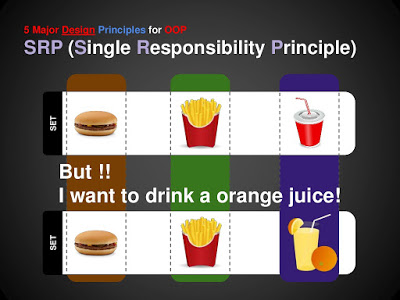
Ideally, if you are adding new functionality only, then your code should be tested, and that's the goal of the [Open Closed Design principle](http://javarevisited.blogspot.com/2011/11/great-example-of-open-closed-design.html). By the way, the Open-Closed principle is "O" from the SOLID acronym.

[](http://javarevisited.blogspot.sg/2015/07/strategy-design-pattern-and-open-closed-principle-java-example.html)

**4. Single Responsibility Principle (SRP)**

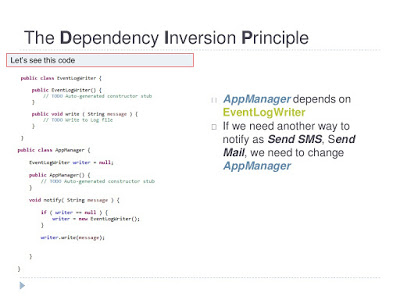
Single Responsibility Principle is another SOLID design principle, and represent  "S" on the SOLID acronym. As per SRP, there should not be more than one reason for a class to change, or a class should always handle single functionality.

If you put more than one functionality in one Class in Java, it introduces **coupling** between two functionality, and even if you change one feature, there is a chance you broke coupled functionality,  which requires another round of testing to avoid any surprise on the production environment.  
  
You can further see the [**SOLID Principles of Object-Oriented Design and Architecture**](https://click.linksynergy.com/deeplink?id=JVFxdTr9V80&mid=39197&murl=https%3A%2F%2Fwww.udemy.com%2Fcourse%2Fsolid-principles-object-oriented-design-architecture%2F) course on Udemy for more real-world examples and to learn about patterns that are based on this principle.

[](http://www.java67.com/2018/01/10-must-read-books-for-coders-of-all-level.html)

**5. Dependency Injection or Inversion principle**

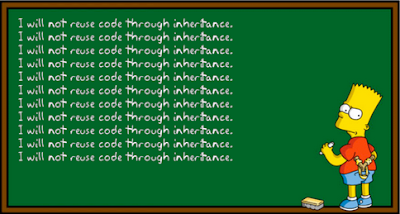
Don't ask for dependency; it will be provided to you by the framework. This has been very well implemented in the [Spring framework](http://www.java67.com/2017/11/top-5-free-core-spring-mvc-courses-learn-online.html), the beauty of this **design principle** is that any class which is injected by the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

[](http://www.java67.com/2016/02/5-books-to-improve-coding-skills-of.html)

There are multiple ways to implemented **Dependency injection** like using bytecode instrumentation, which some AOP (Aspect Oriented Programming) framework like AspectJ does, or by using proxies just like used in Spring. See this [example of the IOC and DI design pattern](http://javarevisited.blogspot.com/2012/12/inversion-of-control-dependency-injection-design-pattern-spring-example-tutorial.html) to learn more about this SOLID design principle. It represents "D" on the SOLID acronym.

**6. Favor Composition over Inheritance**

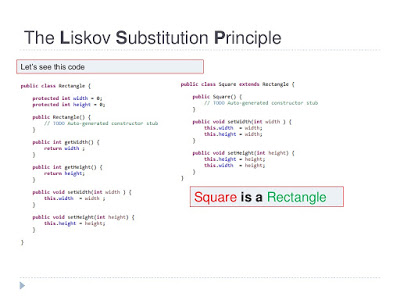
Always favor composition over inheritance, if possible. Some of you may argue this, but I found that Composition is a lot more flexible than [Inheritance](http://javarevisited.blogspot.sg/2012/10/what-is-inheritance-in-java-and-oops-programming.html).  
  
Composition allows changing the behavior of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

[](http://javarevisited.blogspot.sg/2015/06/difference-between-inheritance-and-Composition-in-Java-OOP.html#axzz4tUeeQOAU)

Even the classic book [Effective Java](http://javarevisited.blogspot.sg/2017/10/effective-java-3rd-edition-coming-soon.html) advises favoring composition over inheritance. See [here](http://javarevisited.blogspot.sg/2015/06/difference-between-inheritance-and-Composition-in-Java-OOP.html)to learn more about why your Composition is better than Inheritance for reusing code and functionality.

**7. Liskov Substitution Principle (LSP)**

According to the Liskov Substitution Principle, Subtypes must be substitutable for supertype, i.e. methods or functions which use superclass type must be able to work with the [object](http://javarevisited.blogspot.com/2012/12/what-is-object-in-java-or-oops-example.html)of subclass without any issue".  
  
 LSP is closely related **to the Single responsibility principle** and **Interface Segregation Principle**. If a class has more functionality, then the subclass might not support some of the functionality and does violate LSP.

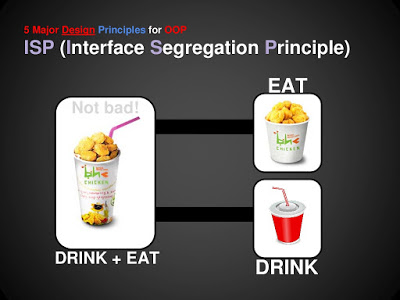
[](http://www.java67.com/2016/03/top-21-java-inheritance-interview-Questions-Answer-Programming.html)

In order to follow **LSP SOLID design principle**, derived class or subclass must enhance functionality, but not reduce them. LSP represents "L" on the SOLID acronym.  If you are interested in a more real-world example, then the [**SOLID Principles of Object-Oriented Design**](https://pluralsight.pxf.io/c/1193463/424552/7490?u=https%3A%2F%2Fwww.pluralsight.com%2Fcourses%2Fprinciples-oo-design) course on Pluralsight is an excellent course to start with.

Btw, you would need a Pluralsight membership to get access to this course, which costs around $29 per month or $299 annually (14% discount).  
  
If you don't have Pluralsight membership, I encourage you to get one because it allows you to access their 5000+ online courses on all the latest topics like front-end and back-end development, machine learning, etc. It also includes interactive quizzes, exercises, and the most recent certification material.  
  
It's more like Netflix for Software Developers, and Since learning is an integral part of our job, Pluralsight membership is a great way to stay ahead of your competition.  
  
They also provide a [10-day free trial](http://pluralsight.pxf.io/c/1193463/424552/7490?u=https%3A%2F%2Fwww.pluralsight.com%2Flearn) without any commitment, which is a great way to not just access this course for free but also to check the quality of courses before joining Pluralsight.

**8. Interface Segregation Principle (ISP)**

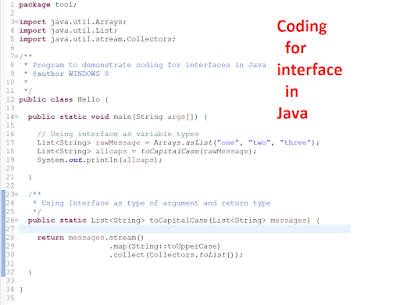
The Interface Segregation Principle states that a client should not implement an [interface](http://javarevisited.blogspot.com/2012/04/10-points-on-interface-in-java-with.html) if it doesn't use that. This happens mostly when one interface contains more than one functionality, and the client only needs one functionality and no other.  
  
Interface design is a tricky job because once you release your interface, you can not change it without breaking all implementation.

[](http://javarevisited.blogspot.sg/2014/09/top-6-books-to-learn-programming-coding.html)

Another benefit of this design principle in Java is, the interface has the disadvantage of implementing all methods before any class can use it, so having single functionality means less method to implement. If you don't get the benefit of the interface in coding, then I suggest you read my blog post, the [real usage of an interface in Java](http://www.java67.com/2014/02/what-is-actual-use-of-interface-in-java.html) to learn more.

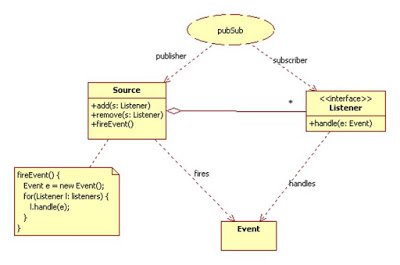
**9. Programming for Interface not implementation**

Always *program for the interface and not for application; this* will lead to flexible code that can work with any new implementation of the interface.  
  
So use interface type on variables, return types of method, or argument type of methods in Java.  
  
  
This has been advised in many Java books, including in [Effective Java](http://www.java67.com/2018/01/effective-java-3rd-edition-by-joshua-bloch-must-read-book-for-java-develoeprs.html) and [Head First design pattern](http://www.java67.com/2016/10/top-5-object-oriented-analysis-and-design-patterns-book-java.html) book.

[](https://javarevisited.blogspot.com/2018/02/10-courses-to-prepare-for-programming-job-interviews.html)

**10. Delegation principles**

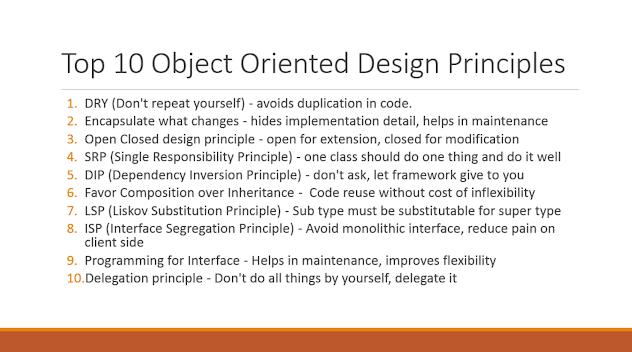
Don't do all stuff by yourself,  delegate it to the respective class. Classical example of delegation design principle is [equals() and hashCode() method in Java](http://javarevisited.blogspot.com/2011/02/how-to-write-equals-method-in-java.html). In order to compare two objects for equality, we ask the class itself to make a comparison instead of the Client class doing that check.  
  
The key benefit of this design principle is no duplication of code and pretty easy to modify behavior. Event delegation is another example of this principle, where an event is delegated to handlers for handling.

[](http://www.java67.com/2017/12/10-ways-to-learn-new-technology-programming-language-or-framework.html)

**Summary**

All these **object-oriented design principles**help you write flexible and better code by striving for high cohesion and low coupling. The theory is the first step, but what is most important is to *develop the ability to find out when to apply these design principles*.  
  
Once you get hold of that, the next step is to learn Design patterns in Java, which uses these design patterns to solve common problems of application development and software engineering.

If you are looking for an excellent course to start with, I suggest you join the [**Software Design Patterns: Best Practices for Software Developers**](https://www.educative.io/courses/software-design-patterns-best-practices?affiliate_id=5073518643380224)course on Educative, a text-based interactive online learning portal where you can practice patterns right on your browser.   
  
Anyway, here is an excellent summary of all these OOP design principles.

[](https://medium.com/javarevisited/7-best-online-courses-to-learn-object-oriented-design-pattern-in-java-749b6399af59)

Find out whether we are violating any design principle and compromising the flexibility of code, but again as nothing is perfect in this world, don't always try to solve the problem with **design patterns and design principles. They** are mostly for large enterprise projects which have a longer maintenance cycle.  
  
The bottom line is, professionals programmers should always strive for a highly cohesive and loosely couple solution, code, or design. Looking at open source code from Apache and Google are some excellent ways of learning Java and OOP design principles.  
  
They show us how design principles should be used in coding and Java programs. Java Development Kit follows many design principles like Factory Pattern in BorderFactory class,  Singleton pattern in Runtime class, Decorator pattern on various java.io classes.  
  
If you are interested in learning object-oriented principles and patterns, then you can look at my other personal favorite [**Head First Object-Oriented Analysis and Design**](http://www.amazon.com/dp/0596008678/?tag=javamysqlanta-20), an excellent book and probably the best material available in object-oriented analysis and design.  
  
Not many programmers know this book because it is often shadowed by its more famous cousin Head First Design Pattern by Eric Freeman, which is more about how these principles come together to create a pattern you can use directly to solve known problems.  
  
These books help a lot to write better code, taking full advantage of various Object-oriented and SOLID design principles.  
  
Btw, if you really interested more in Java coding practices, then read [**Effective Java 3rd Edition**](https://www.amazon.com/Effective-Java-3rd-Joshua-Bloch/dp/0134685997/?tag=javamysqlanta-20) by Joshua Bloch, a gem by the guy who wrote Java Collection API.

Other **Java and Programming Resource**s you may like

* [10 Things Java Programmer should learn in 2021](https://javarevisited.blogspot.com/2017/12/10-things-java-programmers-should-learn.html#axzz5atl0BngO)
* [10 Books Every Programmer Must Read](http://www.java67.com/2015/03/10-books-every-programmer-and-software-engineer-read.html)
* [10 Courses to learn DevOps in Depth](https://javarevisited.blogspot.com/2018/09/10-devops-courses-for-experienced-java-developers.html)
* [The 2021 Java Developer RoadMap](https://javarevisited.blogspot.com/2019/10/the-java-developer-roadmap.html#123)
* [My favorite courses to learn Software Architecture](https://medium.com/javarevisited/top-5-courses-to-learn-software-architecture-in-2020-best-of-lot-5d34ebc52e9)
* [10 Tips to Improve Your Programming skill](http://javarevisited.blogspot.sg/2014/01/10-tips-to-improve-programming-skill-become-better-programmer.html#axzz553pz1hYh)
* [10 Tools Every Software Developer should know](http://javarevisited.blogspot.sg/2018/01/10-tools-every-software-developer-know.html#axzz559dyoLSA)
* [21 Tech skills Java developers should learn in 2021](https://javarevisited.blogspot.com/2020/03/top-20-skills-java-developers-can-learn.html#axzz6k4XBgTw4)
* [5 Courses to Learn Software Architecture in Depth](https://javarevisited.blogspot.com/2019/03/5-courses-programmers-can-join-to-learn.html#axzz5jKqbGRcg)
* [20 Libraries and APIS Java Programmer Should Know](https://javarevisited.blogspot.com/2018/01/top-20-libraries-and-apis-for-java-programmers.html)
* [7 Best books to learn about Design Patterns](https://medium.com/javarevisited/7-best-books-to-learn-design-patterns-for-java-programmers-5627b93eefdb)
* [Top 10 Programming languages to Learn in 2021](http://www.java67.com/2017/12/10-programming-languages-to-learn-in.html)
* [10 Articles Every Programmer Should Read](http://javarevisited.blogspot.sg/2014/05/10-articles-every-programmer-must-read.html)
* [10 Framework and Library Java and Web Developer Should Learn](http://javarevisited.blogspot.sg/2018/01/10-frameworks-java-and-web-developers-should-learn.html)

Thanks for reading this article. If you find these *object-oriented design principles* useful, then please share them with your friends and colleagues. If you have any questions or feedback, then please drop a note.  
  
**P. S. -** If you are serious about improving your coding skills then as a next step you can also learn design patterns, they are the patterns that use these principles to solve common problems of software development. If you need resources, check out these [**Design Patterns in Java**](https://click.linksynergy.com/deeplink?id=JVFxdTr9V80&mid=39197&murl=https%3A%2F%2Fwww.udemy.com%2Fcourse%2Fdesign-patterns-java%2F) courses by Dmitri Nestruk on Udemy to learn more about the modern implementation of classic design patterns in Java.