<http://javarevisited.blogspot.com/2012/03/10-object-oriented-design-principles.html>

10 Object Oriented Design Principles Java Programmer should know

**Object Oriented Design Principles** are core of OOP programming, but I have seen most of the Java programmers chasing design patterns like [Singleton pattern](http://javarevisited.blogspot.sg/2013/03/difference-between-singleton-pattern-vs-static-class-java.html), [Decorator pattern](http://java67.blogspot.sg/2013/07/decorator-design-pattern-in-java-real-life-example-tutorial.html) or Observer pattern, and not putting enough attention on learning *Object oriented analysis and design*. It's important to learn basics of Object oriented programming like Abstraction, Encapsulation, Polymorphism and Inheritance. But, at the same time, it's equally important to know object oriented design principles, to create clean and modular design. I have regularly seen Java programmers and developers of various experience level, who either doesn't heard about these **OOP** and **SOLID design principle,** or simply doesn't know what **benefits a particular design principle** offers, or how to apply these design principle in coding.

Bottom line is, always strive for highly cohesive and loosely couple solution, code or design. Looking open source code from Apache and Sun are good examples of learning Java and OOPS design principles. They show us,  how design principles should be used in coding and Java programs. Java Development Kit follows several design principle like Factory Pattern in BorderFactory class,  Singleton pattern in Runtime class, Decorator pattern on various java.io classes. By the way if you really interested more on Java coding practices then read Effective Java by Joshua Bloch , a gem by the guy who wrote Java Collection API.

If you are interested in learning object oriented principles and patterns, then you can look at my another personal favorite [Head First Object Oriented Analysis and Design](http://www.amazon.com/dp/0596008678/?tag=javamysqlanta-20). This an excellent book and probably the best material available in object oriented analysis and design, but it often shadowed by its more popular cousin Head First Design Pattern by Kathy Sierra. Later is more about how these principle comes together to create pattern you can use directly to solve known problems. These books helps a lot to write better code, taking full advantage of various Object oriented and SOLID design principles.

Though best way of learning any design principle or pattern is real world example and understanding the consequences of violating that design principle, subject of this article is Introducing *Object oriented design principles* for Java Programmers, who are either not exposed to it or in learning phase. I personally think each of these OOPS and SOLID design principle need an article to explain them clearly, and I will definitely try to do that here, but for now just get yourself ready for quick bike ride on design principle town :)

## DRY (Don't repeat yourself)

Our first object oriented design principle is DRY, as name suggest **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 block of code in more than two place 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). 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. So beware of such coupling and just don’t combine anything which uses similar code but are not related.

## Encapsulate What Changes

Only one thing is constant in software field and that is "Change", So encapsulate the code you expect or suspect to be changed in future. Benefit of this OOPS Design principle is that Its easy to test and maintain proper encapsulated code. If you are coding in Java then follow principle of making variable and methods private by default and increasing access step by step e.g. from private to protected and not public. Several of **design pattern in Java** uses Encapsulation, [Factory design pattern](http://javarevisited.blogspot.com/2011/12/factory-design-pattern-java-example.html) is one example of Encapsulation which encapsulate object creation code and provides flexibility to introduce new product later with no impact on existing code.

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

## Single Responsibility Principle (SRP)

Single Responsibility Principle is another SOLID design principle, and represent  "S" on 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](http://javarevisited.blogspot.com/2011/10/class-in-java-programming-general.html)  it introduce **coupling** between two functionality and even if you change one functionality there is chance you broke coupled functionality,  which require another round of testing to avoid any surprise on production environment.

## Dependency Injection or Inversion principle

Don't ask for dependency it will be provided to you by framework. This has been very well implemented in Spring framework, beauty of this **design principle** is that any class which is injected by DI framework is easy to test with mock object and easier to maintain because object creation code is centralized in framework and client code is not littered with that.There are multiple ways to  implemented **Dependency injection** like using  byte code instrumentation which some AOP (Aspect Oriented programming) framework like AspectJ does or by using proxies just like used in Spring. See this [example of 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 represent "D" on SOLID acronym.

## Favor Composition over Inheritance

Always *favor composition over inheritance* ,if possible. Some of you may argue this, but I found that Composition is lot more flexible than [Inheritance](http://javarevisited.blogspot.sg/2012/10/what-is-inheritance-in-java-and-oops-programming.html). Composition allows to change 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 of to replace with better implementation any time. Even Effective Java advise to favor 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 you Composition is better than Inheritance for reusing code and functionality.

## Liskov Substitution Principle (LSP)

According to Liskov Substitution Principle, Subtypes must be substitutable for super type i.e. methods or functions which uses super class type must be able to work with [object](http://javarevisited.blogspot.com/2012/12/what-is-object-in-java-or-oops-example.html) of sub class without any issue". LSP is closely related **to Single responsibility principle** and **Interface Segregation Principle**. If a class has more functionality than subclass might not support some of the functionality ,and does violated LSP. In order to follow **LSP SOLID design principle**, derived class or sub class must enhance functionality, but not reduce them. LSP represent  "L" on SOLID acronym.

## Interface Segregation principle (ISP)

Interface Segregation Principle stats 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 client only need one functionality and not other.Interface design is tricky job because once you release your interface you can not change it without breaking all implementation. Another benefit of this design principle in Java is, interface has disadvantage to implement all method before any class can use it so having single functionality means less method to implement.

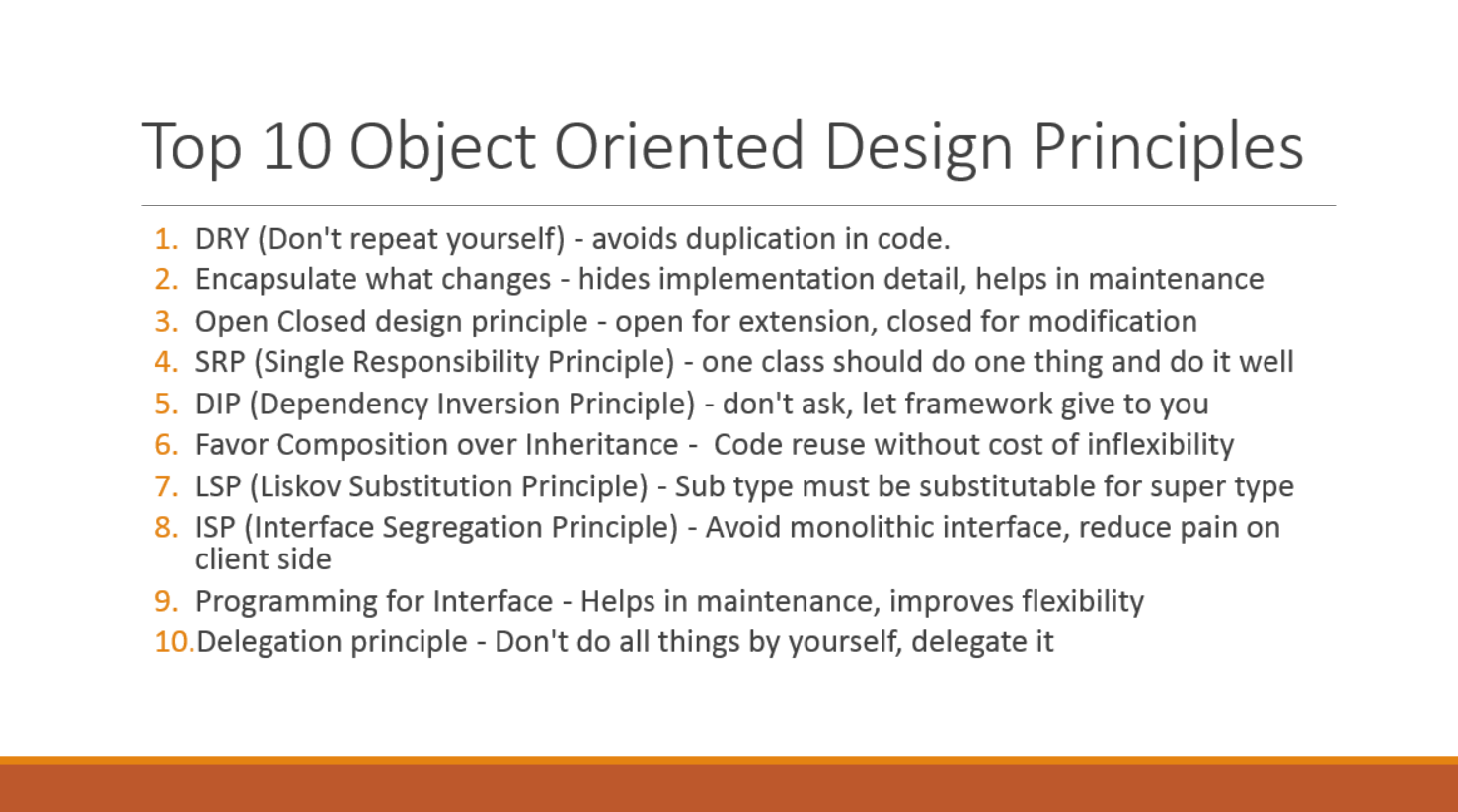
## Programming for Interface not implementation

Always *program for interface and not for implementation* this will lead to flexible code which can work with any new implementation of interface. So use interface type on variables, return types of method or argument type of methods in Java. This has been advised by many Java programmer including in Effective Java and Head First design pattern book.

## Delegation principle

Don't do all stuff  by yourself,  delegate it to 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 object for equality we ask class itself to do comparison instead of Client class doing that check. Benefit of this design principle is no duplication of code and pretty easy to modify behavior.

Here is nice summary of all these OOP design principles :



All these **object oriented design principle** helps you write flexible and better code by striving high cohesion and low coupling. Theory is first step, but what is most important is to *develop ability to find out when to apply these design principle*. Find out, whether we are violating any design principle and compromising flexibility of code, but again as nothing is perfect in this world, don't always try to solve problem with **design patterns and design principle** they are mostly for large enterprise project which has longer maintenance cycle.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

<http://javarevisited.blogspot.sg/2015/06/difference-between-inheritance-and-Composition-in-Java-OOP.html>

Difference between Inheritance and Composition in Java OOPS

Though both Inheritance and Composition provides code reusablility, main difference between Composition and Inheritance in Java is that Composition allows reuse of code without extending it but for Inheritance you must extend the class for any reuse of code or functionality. Another difference which comes from this fact is that by using Composition you can reuse code for even final class which is not extensible but Inheritance cannot reuse code in such cases. Also by using Composition you can reuse code from many classes as they are declared as just a member variable, but with Inheritance you can reuse code form just one class because in Java you can only extend one class, because multiple Inheritance is not supported in Java. You can do this in C++ though because there one class can extend more than one class. BTW, You should always *prefer Composition over Inheritance in Java*, its not just me but even Joshua Bloch has suggested in his book [Effective Java](http://www.amazon.com/dp/0321356683/?tag=javamysqlanta-20), which is a great resource to learn how to do things in right way in Java. I have listed my argument in favor of Composition over Inheritance in my [earlier post](http://javarevisited.blogspot.sg/2013/06/why-favor-composition-over-inheritance-java-oops-design.html), which you can check now or later.

**Inheritance vs Composition**

Now let's understand difference between Inheritance and Composition in little bit more detail. I will go point by point and try to explain each point in as much detail as possible without boring you :)

**1) Static vs Dynamic**

First difference between Inheritance and Composition comes from flexibility point of view. When you use Inheritance, you have to define which class you are extending in code, it cannot be changed at runtime, but with Composition you just define a Type which you want to use, which can hold its different implementation. In this sense, Composition is much more flexible than Inheritance.

**2) Limited code reuse with Inheritance**

As I told, with Inheritance you can only extend one class, which means you code can only reuse just one class, not more than one. If you want to leverage functionalities from multiple class, you must use Composition. For example, if your code needs authentication functionality, you can use an Authenticater, for authorization you can use an Authorizer etc, but with Inheritance you just stuck with only class, Why? because [Java doesn't support multiple Inheritance](http://javarevisited.blogspot.sg/2011/07/why-multiple-inheritances-are-not.html). This difference between Inheritance vs Composition actually highlight a severe limitation of later.

**3) Unit Testing**

This is in my opinion most important difference between Inheritance and Composition in OOP and probably is the deciding factor in whether to use Composition or Inheritance. When you design classes using Composition they are easier to test because you can supply mock implementation of the classes you are using but when you design your class using Inheritance, you must need parent class in order to test child class. Their is no way you can provide mock implementation of parent class.

**4) Final classes**

Third difference between them also highlight another limitation of Inheritance. Composition allows code reuse even from final classes, which is not possible using Inheritance because [you cannot extend final class in Java](http://javarevisited.blogspot.sg/2011/12/final-variable-method-class-java.html), which is necessary for Inheritance to reuse code.

**5) Encapsulation**

Last difference between Composition and Inheritance in Java in this list comes from Encapsulation and robustness point of view. Though both Inheritance and Composition allows code reuse, Inheritance breaks encapsulation because in case of Inheritance, sub class is dependent upon super class behavior. If parent classes changes its behavior than child class is also get affected. If classes are not properly documented and child class has not used the super class in a way it should be used, any change in super class can break functionality in sub class. In order to understand with a great example, I strongly suggest you to read [Effective Java Item 16 and 17](http://www.amazon.com/dp/0321356683/?tag=javamysqlanta-20).

That's all about **difference between Inheritance and Composition in Java and OOP**. You can see that even though Inheritance and Composition has same goal to assist in reusing tried and tested goal their choice brings different challenges. Composition provides better way to reuse code and same time protect the class you are reusing from any of its client, but Inheritance doesn't offer that guarantee. Sometime though Inheritance is necessary, mainly when you are creating class from same family.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

<http://javarevisited.blogspot.com.by/2011/11/great-example-of-open-closed-design.html>

Great Example of Open Closed Design Principle in Java

**Great Example of Open Closed Design Principle**

I am big fan of design pattern article and love to read articles on design patterns and recently wrote about [decorator design pattern in Java](http://javarevisited.blogspot.com/2011/11/decorator-design-pattern-java-example.html),  [Observer pattern](http://javarevisited.blogspot.sg/2011/12/observer-design-pattern-java-example.html),  [static factory pattern](http://javarevisited.blogspot.com/2011/12/factory-design-pattern-java-example.html) and about Singleton pattern. Today I come across this good article on open closed design pattern, what I like most is there example and clear way of explanation, first example is true value and it will help you understand open closed principle very quickly and second example is also not bad. I have also shared many questions on design patterns and Object oriented design principles like 10 Java [singleton interview questions](http://javarevisited.blogspot.sg/2011/03/10-interview-questions-on-singleton.html) , which discusses some of the most common questions on Singleton pattern like how to write Thread-Safe Singleton class , should we use [Enum as Singleton](http://javarevisited.blogspot.gr/2012/07/why-enum-singleton-are-better-in-java.html) or not etc and 20 [design pattern interview questions](http://javarevisited.blogspot.sg/2012/06/20-design-pattern-and-software-design.html), if you are preparing for senior level Java interview, where questions from design is expected, its worth looking for quick review.

One word about **Open Closed principle** is a design principle which says that a class should be open for extension but closed for modification. Open Closed Principles is one of the principle from SOLID design principle where it represent "O". Read more about OOPS and SOLID design pattern in my post [10 OOPS and SOLID design principles Java programmer should know](http://javarevisited.blogspot.sg/2012/03/10-object-oriented-design-principles.html).

In Simple language Open closed design principles says that new functionality should be added by introducing new classes, methods or fields instead of modifying already tried and tested code. One of the way to achieve this is Inheritance where class is extended to introduce new functionality on top of inherited basic features.

Benefit or Open Closed Design Principle:

1) Application will be more robust because we are not changing already tested class.

2) Flexible because we can easily accommodate new requirements.

3) Easy to test and less error prone.

**How to make code extensible**

Basic principle of making your [code](http://javarevisited.blogspot.sg/2011/09/code-review-checklist-best-practice.html) extensible and following open closed principle is providing object to [class](http://javarevisited.blogspot.sg/2011/10/class-in-java-programming-general.html) at run time and making use of polymorphism to invoke extended functionality.

If functionality is hard Coded than it wouldn’t be extensible but if you write interface and provide implementation of that interface at run time you make it extensible.

For detailed example see this link open closed design pattern on sanaulla blog post http://blog.sanaulla.info/2011/11/19/solid-open-closed-principle/

**Interview Questions you may like**

[Top 30 UNIX Command Interview Questions Answers](http://javarevisited.blogspot.com/2011/05/unix-command-interview-questions.html)

[10 Interview questions on Singleton Pattern in Java](http://javarevisited.blogspot.com/2011/03/10-interview-questions-on-singleton.html)

[Top 30 Programming Interview Questions](http://javarevisited.blogspot.com/2011/06/top-programming-interview-questions.html)

[Top 10 Serialization Interview Questions in java](http://javarevisited.blogspot.com/2011/04/top-10-java-serialization-interview.html)

[Top 20 Core Java Interview Questions Answers asked in Investment banks](http://javarevisited.blogspot.com/2011/04/top-20-core-java-interview-questions.html)

[Top 15 Multi-Threading Interview Questions asked in IB](http://javarevisited.blogspot.com/2011/07/java-multi-threading-interview.html)

Posted by Javin Paul at [6:02 AM](http://javarevisited.blogspot.com.by/2011/11/great-example-of-open-closed-design.html) [http://img1.blogblog.com/img/icon18_email.gif](https://www.blogger.com/email-post.g?blogID=8712770457197348465&postID=8173257287594489205)

[Email This](https://www.blogger.com/share-post.g?blogID=8712770457197348465&postID=8173257287594489205&target=email) [BlogThis!](https://www.blogger.com/share-post.g?blogID=8712770457197348465&postID=8173257287594489205&target=blog) [Share to Twitter](https://www.blogger.com/share-post.g?blogID=8712770457197348465&postID=8173257287594489205&target=twitter) [Share to Facebook](https://www.blogger.com/share-post.g?blogID=8712770457197348465&postID=8173257287594489205&target=facebook)

Labels: [core java](http://javarevisited.blogspot.com.by/search/label/core%20java?max-results=3)

Location: [United States](https://maps.google.com/maps?q=United+States@38.27268853598097,-103.359375&z=10)

#### 3 comments :

OOOD said...

I think best example of Open Closed Principle is alphabets and words, you have just got 26 alphabets but you can make any number of words. 26 Alphabets are close for modification but open to made any words, what do you say?

[March 22, 2013 at 12:55 AM](http://javarevisited.blogspot.com/2011/11/great-example-of-open-closed-design.html?showComment=1363938952181#c807430854769261786)

Anonymous said...

I think key to understand Open Closed principle or OCP is creating better abstraction. If you have better abstraction, you will more likely to confirm open closed design principle. At the same time, you should be careful of creating too many abstraction, which will result in lots of vapor classes, Instead you can delay creation of Abstraction related to Open Closed principle, till requirement comes up, but once a requirement to change a module comes up, you should put together correct set of abstraction, so that module can handle similar changes on that module.

Side note: If your class tends to follow Single Responsibility principle (SRP), then they are more likely to follow OCP as well, because it delegates responsibility to a different class, which handles a particular abstraction.

[April 15, 2013 at 1:47 AM](http://javarevisited.blogspot.com/2011/11/great-example-of-open-closed-design.html?showComment=1366015661212#c3162497294183964364)

Raj said...

Strategy Design Pattern is another example of Open Closed design Principle. By using Strategy to perform certain task, we keep Service Class, which uses this strategy closed, but same time, System is open for extension, by introducing new Strategy, by implementing Strategy interface. At runtime, you can compose Service Class with any new Strategy, based upon your need.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

<http://javarevisited.blogspot.sg/2012/03/what-is-encapsulation-in-java-and-oops.html>

What is Encapsulation in Java and OOPS with Example

**Encapsulation in Java** or object oriented programming language is a concept which enforce protecting variables, functions from outside of class, in order to better manage that piece of code and having least impact or no impact on other parts of program duec to change in protected code. *Encapsulation in Java* is visible at different places and Java language itself provide many construct to encapsulate members. You can completely encapsulate a member be it a variable or method in Java by using private keyword and you can even achieve a lesser degree of encapsulation in Java by using other access modifier like protected or public. true value of encapsulation is realized in an environment which is prone to change a lot and we know that in software requirements changes every day at that time if you have your code well encapsulated you can better manage risk with change in requirement. Along with [abstaction in java](http://javarevisited.blogspot.com/2010/10/abstraction-in-java.html) and [polymorphism in Java](http://javarevisited.blogspot.com/2011/08/what-is-polymorphism-in-java-example.html), Encapsulation is a must know concept. In this java tutorial  we will see **How to use encapsulation in Java**, advantage and disadvantage of Encapsulation and various design patterns and real life problems which makes use of Encapsulation object oriented concept. If you are looking for a quick guide on both OOPS and SOLID design principle in Java than you may find [**10 Object Oriented Design principles Java programmer should know**](http://javarevisited.blogspot.com/2012/03/10-object-oriented-design-principles.html)  interesting.

## What is Encapsulation in Java

**Encapsulation** is nothing but protecting anything which is prone to change. rational behind encapsulation is that if any functionality which is well encapsulated in code i.e maintained in just one place and not scattered around code is easy to change. this can be better explained with a simple example of encapsulation in Java. we all know that constructor is used to create object in Java and constructor can accept argument. Suppose we have a class Loan has a constructor and than in various classes you have created instance of loan by using this [constructor](http://javarevisited.blogspot.com/2012/01/what-is-constructor-overloading-in-java.html). now requirements change and you need to include age of borrower as well while taking loan. Since this code is not well encapsulated i.e. not confined in one place you need to change everywhere you are calling this constructor i.e. for one change you need to modify several file instead of just one file which is more error prone and tedious, though it can be done with refactoring feature of advanced IDE wouldn't it be better if you only need to make change at one place ? Yes that is possible if we encapsulate Loan creation logic in one method say createLoan() and client code call this method and this method internally crate Loan object. in this case you only need to modify this method instead of all client code.

**Example of Encapsulation in Java**

**class** Loan{

**private** **int** duration;  //private variables examples of encapsulation

**private** [**String**](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html) loan;

**private** **String** borrower;

**private** **String** salary;

    //public constructor can break encapsulation instead use factory method

**private** Loan(**int** duration, **String** loan, **String** borrower, **String** salary){

**this**.duration = duration;

**this**.loan = loan;

**this**.borrower = borrower;

**this**.salary = salary;

    }

    //no argument consustructor omitted here

   // create loan can encapsulate loan creation logic

**public** Loan createLoan(**String** loanType){

     //processing based on loan type and than returning loan object

**return** loan;

    }

}

In this same example of *Encapsulation in Java* you see all member variables are made private so they are well encapsulated you can only change or access this variable directly inside this class. if you want to allow outside world to access these variables is better creating a getter and setter e.g. getLoan() that allows you to do any kind of validation, security check before return loan so it gives you complete control of whatever you want to do and single channel of access for client which is controlled and managed.

## Advantage of Encapsulation in Java and OOPS

Here are few advantages of using **Encapsulation** while writing code in Java or any Object oriented programming language:

1. Encapsulated Code is more flexible and easy to change with new requirements.

2. Encapsulation in Java makes unit testing easy.

3. Encapsulation in Java allows you to control who can access what.

4. Encapsulation also helps to write immutable class in Java which are a good choice in multi-threading

environment.

5. Encapsulation reduce coupling of modules and increase cohesion inside a module because all piece of one thing

are encapsulated in one place.

6. Encapsulation allows you to change one part of code without affecting other part of code.

**What should you encapsulate in code**

Anything which can be change and more likely to change in near future is candidate of Encapsulation. This also helps to write more specific and cohesive code. Example of this is object creation code, code which can be improved in future like sorting and searching logic.

## Design Pattern based on Encapsulation in Java

Many design pattern in Java uses encapsulation concept, one of them is [Factory pattern](http://javarevisited.blogspot.com/2011/12/factory-design-pattern-java-example.html) which is used to create objects. Factory pattern is better choice than new operator for creating object of those classes whose creation logic can vary and also for creating different implementation of same interface. BorderFactory class of JDK is a good example of encapsulation in Java which creates different types of Border and encapsulate creation logic of Border. [Singleton pattern in Java](http://javarevisited.blogspot.com/2011/03/10-interview-questions-on-singleton.html) also encapsulate how you create instance by providing getInstance() method. since object

is created inside one class and not from any other place in code you can easily change how you create object without

affect other part of code.

### Important points aboue encapsulation in Java.

1. "Whatever changes encapsulate it" is a famous design principle.

2. Encapsulation helps in loose coupling and high cohesion of code.

3. Encapsulation in Java is achieved using access modifier private, protected and public.

4. Factory pattern , Singleton pattern in Java makes good use of Encapsulation.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

<http://javarevisited.blogspot.com/2010/10/abstraction-in-java.html>

What is Abstraction in Java? Abstract Class or Interface

**What is abstraction?**

Abstraction in Java or Object oriented programming is a way to segregate implementation from interface and one of the five fundamentals along with Encapsulation, Inheritance, Polymorphism, Class and Object.  Abstraction in Java is achieved by  using interface and abstract class in Java. An interface or abstract class is something which is not concrete , something which is incomplete. In order to use interface or abstract class we need to extend and implement abstract method with concrete behavior. One example of Abstraction is creating interface to denote common behavior without specifying any details about how that behavior works e.g. You create an [interface](http://javarevisited.blogspot.sg/2012/04/10-points-on-interface-in-java-with.html) called Server which has start() and stop() method. This is called abstraction of Server because every server should have way to start and stop and details may differ. As I said earlier Abstraction in Java is implemented using abstract class and interface as discussed in next section. In fact what is abstraction in Java, difference between Abstraction and Encapsulation  is also a very popular core Java interview because strong OOPS skill is one of the primary requirement for Java developers.

## What is abstract class in Java

An **abstract class** is something which is incomplete and you can not create instance of abstract class. If you want to use it you need to make it complete or concrete by extending it. A [class](http://javarevisited.blogspot.com/2011/10/class-in-java-programming-general.html) is called concrete if it does not contain any abstract method and implements all abstract method inherited from abstract class or interface it has implemented or extended.By the way Java has concept of abstract classes, abstract method but a variable can not be abstract in Java. Popular example of abstract class in Java is ActionListener which has abstract method called actionPerformed(ActionEvent ae). This method is called when an ActionEvent is fired like when you click on JButton. Its common in java to attach ActionListener with JButton by implementing abstract method actionPerformed(ActionEvent ae) using Anonymous class, as shown in below Example :

JButton ok = **new** JButton("OK");

ok.addActionListener(**new** ActionListener(){

**public** **void** **actionPerformed**(ActionEvent ae){

//code to handle event

}

});

***An abstract method in Java doesn't have body , its just a declaration.*** In order to use abstract method you need to [override](http://javarevisited.blogspot.sg/2011/12/method-overloading-vs-method-overriding.html) that method in sub class***.***

so ***when do you use abstraction*** ? ( most important in my view )

when you know something needs to be there but not sure how exactly it should look like. e.g. when I am creating a class called Vehicle, I know there should be methods like start() and stop() but don't know how that start and stop method should work, because every vehicle can have different start and stop mechanism e..g some can be started by kicking or some can be by pressing buttons . Same concept apply to interface in Java as well, which we will discuss in some other post.

So implementation of those start() and stop() methods should be left to there concrete implementation e.g. Scooter , MotorBike , Car etc.

## Abstraction Using Interface in Java

InJava Interface is an another way of providing **abstraction**, **Interfaces** are by default abstract and only contains public, static, final constant or abstract methods. Its very common interview question is that where should we use ***abstract class*** and where should we use ***Java Interfaces*** in my view this is important to understand to [design better Java application](http://javarevisited.blogspot.sg/2011/09/how-to-write-production-quality-code.html), you can go for java interface if you only know the name of methods your class should have e.g. for Server it should have start() and stop() method but we don't know how exactly these start and stop method will work. if you know some of the behavior while designing class and that would remain common across all sub classes add that into ***abstract class***. Interface like Runnable interface is good example of abstraction in Java which is used to abstract task executed by multiple thread. Callable is another good abstract of a task which can return value.

### Abstraction : Things to Remember

1) Use abstraction if you know something needs to be in class but implementation of that varies. Abstraction is actually result of thought process and it really need good experience of both domain and Object oriented analysis and design to come up with good abstraction for your project.

2) In Java you can not create instance of abstract class using new operator, its compiler error. Though abstract class can have constructor.

3) abstract is a keyword in Java, which can be used with both class and method.  Abstract class can contain both abstract and concrete method. Abstract method doesn't have body, just declaration.

4) A class automatically becomes abstract class when any of its method declared as abstract.

5) abstract method doesn't have method body.

6) In Java, variable can not be made abstract , its only class or methods which would be abstract.

7) If a class extends an abstract class or interface it has to provide implementation to all its abstract method to be a concrete class. alternatively this class can also be abstract.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

<http://javarevisited.blogspot.com.by/2011/08/what-is-polymorphism-in-java-example.html>

What is polymorphism in Java? Method overloading or overriding?

**What is Polymorphism in Java**

Polymorphism is an important [Object oriented concept](http://javarevisited.blogspot.sg/2012/03/10-object-oriented-design-principles.html) and widely used in Java and other programming language.  **Polymorphism in java** is supported along with other concept like [Abstraction](http://javarevisited.blogspot.sg/2010/10/abstraction-in-java.html), [Encapsulation](http://javarevisited.blogspot.sg/2012/03/what-is-encapsulation-in-java-and-oops.html) and Inheritance. Few words on historical side; Polymorphism word comes from ancient Greek where poly means many so polymorphic are something which can take many form. In this Java Polymorphism tutorial we will see what is Polymorphism in Java , How Polymorphism is implemented in Java e.g [method overloading and overriding](http://javarevisited.blogspot.sg/2011/08/what-is-polymorphism-in-java-example.html), why should we use Polymorphism and how can we take advantage of polymorphism while [writing code in Java](http://javarevisited.blogspot.sg/2011/09/how-to-write-production-quality-code.html). Along the way we will also see a real world example of using **Polymorphism in Java**.

By the way I learned about [Abstraction](http://javarevisited.blogspot.sg/2010/10/abstraction-in-java.html), [Encapsulation](http://javarevisited.blogspot.sg/2012/03/what-is-encapsulation-in-java-and-oops.html) and [Polymorphism](http://javarevisited.blogspot.sg/2011/08/what-is-polymorphism-in-java-example.html) during my college time but never able to recognize its real potential until I started doing programming and involved in bigger projects. On theory Polymorphism is a simple concept where one variable can denote multiple object but in real life it just a gem and a code written using polymorphism concept is much flexible to change and quite easy to maintain than the one which is written without polymorphism. In Java programming whole [concept of interface](http://javarevisited.blogspot.sg/2012/04/10-points-on-interface-in-java-with.html) is based on Polymorphism and its a [famous design principle](http://javarevisited.blogspot.sg/2012/03/10-object-oriented-design-principles.html) that to code for interface than implementation to take advantage of Polymorphism and introducing new implementation in future.

### What is polymorphism in Java

**Polymorphism is an Oops concept** which advice use of **common interface** *instead of concrete implementation* while [writing code](http://javarevisited.blogspot.sg/2011/09/code-review-checklist-best-practice.html). When we program for interface our code is capable of handling any new requirement or enhancement arise in near future due to new implementation of our common interface. If we don't use common interface and rely on concrete implementation, we always need to change and duplicate most of our code to support new implementation. Its not only Java but other object oriented language like C++ also supports polymorphism and it comes as fundamental along with other OOPS concepts like [Encapsulation](http://javarevisited.blogspot.sg/2012/03/what-is-encapsulation-in-java-and-oops.html) , Abstraction and Inheritance.

### How Polymorphism supported in Java

Java has excellent support of polymorphism in terms of Inheritance, [method overloading](http://javarevisited.blogspot.sg/2011/12/method-overloading-vs-method-overriding.html) and [method overriding](http://javarevisited.blogspot.sg/2011/12/method-overloading-vs-method-overriding.html). Method overriding allows Java to [invoke method](http://javarevisited.blogspot.sg/2012/04/how-to-invoke-method-by-name-in-java.html) based on a particular object at run-time instead of declared type while coding. To get hold of concept let's see an **example of polymorphism in Java:**

**public class** TradingSystem{

**public** String getDescription(){

**return** "electronic trading system";

   }

}

**public class** DirectMarketAccessSystem **extends** TradingSystem{

**public** String getDescription(){

**return** "direct market access system";

   }

}

**public class** CommodityTradingSystem **extends** TradingSystem{

**public** String getDescription(){

**return** "Futures trading system";

   }

}

Here we have a super class called TradingSystem and there two implementation DirectMarketAccessSystem and CommodityTradingSystem and here we will write code which is flexible enough to work with any future implementation of TradingSystem we can achieve this by using Polymorphism in Java which we will see in further example.

### Where to use Polymorphism in code

Probably this is the most important part of this Java Polymorphism tutorial and It’s good to know *where you can use Polymorphism in Java* while [writing code](http://javarevisited.blogspot.sg/2011/08/code-comments-java-best-practices.html). Its common practice to always replace concrete implementation with interface it’s not that easy and  comes with practice but here are some common places where I check for polymorphism:

**1) Method argument:**

Always use super type in method argument that will give you leverage to pass any implementation while invoking method. For example:

**public** **void** showDescription(TradingSystem tradingSystem){

   tradingSystem.description();

}

If you have used concrete implementation e.g. CommodityTradingSystem or DMATradingSystem then that code will require frequent changes whenever you add new Trading system.

**2) Variable names:**

Always use Super type while you are storing reference returned from any [Factory method in Java](http://javarevisited.blogspot.sg/2011/12/factory-design-pattern-java-example.html), This gives you flexibility to accommodate any new implementation from Factory. Here is an example of polymorphism while writing Java code which you can use retrieving reference from Factory:

String systemName = Configuration.getSystemName();

TradingSystem system = TradingSystemFactory.getSystem(systemName);

**3) Return type of method**

Return type of any method is another place where you should be using interface to take advantage of Polymorphism in Java. In fact this is a requirement of [Factory design pattern in Java](http://javarevisited.blogspot.sg/2011/12/factory-design-pattern-java-example.html) to use interface as return type for factory method.

**public** TradingSystem getSystem(String name){

   //code to return appropriate implementation

}

### Method overloading and method overriding in Java

[Method overloading and method overriding](http://javarevisited.blogspot.sg/2011/08/what-is-polymorphism-in-java-example.html) uses concept of Polymorphism in Java where method name remains same in two classes but actual method called by [JVM](http://javarevisited.blogspot.sg/2011/11/hotspot-jvm-options-java-examples.html) depends upon object at run time and done by [dynamic binding in Java](http://javarevisited.blogspot.sg/2012/03/what-is-static-and-dynamic-binding-in.html)*. Java supports both overloading and overriding of methods*. In case of overloading method signature changes while in case of overriding method signature remains same and binding and invocation of method is decided on runtime based on actual object. This facility allows Java programmer to write very flexibly and maintainable code using [interfaces](http://javarevisited.blogspot.sg/2012/04/10-points-on-interface-in-java-with.html) without worrying about concrete implementation. One disadvantage of using **Polymorphism in code** is that while reading code you don't know the actual type which annoys while you are looking to find bugs or trying to debug program. But if you do [Java debugging in IDE](http://javarevisited.blogspot.sg/2011/07/java-debugging-tutorial-example-tips.html) you will definitely be able to see the actual object and the method call and variable associated with it.

### Parameteric Polymorphism in Java

Java started to support **parametric polymorphism** with introduction of [Generic in JDK1.5](http://javarevisited.blogspot.sg/2011/09/generics-java-example-tutorial.html). Collection classes in JDK 1.5 are written using Generic Type which allows Collections to hold any type of object in run time without any change in code and this has been achieved by passing actual Type as parameter. For example see the below code of a parametric cache written using Generic which shows use of **parametric polymorphism in Java**. Read [how to create Generic class and methods in Java](http://javarevisited.blogspot.sg/2012/08/how-to-write-parametrized-class-method-Generic-example.html) for more details.

**interface** cache{

**public** **void** put(K key, V value);

**public** V get(K key);

}

That’s all on polymorphism in java for now, please suggest and share some of other coding practices which involve use o f polymorphic behavior of java for benefit of all. Thank you.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

SOLID

|  |  |  |
| --- | --- | --- |
| **Буква** | **Означает** | **Описание** |
| **S** | [Single responsibility principle](https://en.wikipedia.org/wiki/Single_responsibility_principle) | [Принцип единственной обязанности](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BD%D1%86%D0%B8%D0%BF_%D0%B5%D0%B4%D0%B8%D0%BD%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D0%BE%D0%B9_%D0%BE%D0%B1%D1%8F%D0%B7%D0%B0%D0%BD%D0%BD%D0%BE%D1%81%D1%82%D0%B8)  На каждый [класс](https://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B0%D1%81%D1%81_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29) должна быть возложена одна-единственная обязанность.  a class should have only a single responsibility (i.e. only one potential change in the software's specification should be able to affect the specification of the class) |
| **O** | [Open/closed principle](https://en.wikipedia.org/wiki/Open/closed_principle) | [Принцип открытости/закрытости](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BD%D1%86%D0%B8%D0%BF_%D0%BE%D1%82%D0%BA%D1%80%D1%8B%D1%82%D0%BE%D1%81%D1%82%D0%B8/%D0%B7%D0%B0%D0%BA%D1%80%D1%8B%D1%82%D0%BE%D1%81%D1%82%D0%B8)  Программные сущности должны быть открыты для расширения, но закрыты для изменения.  “software entities … should be open for extension, but closed for modification.” |
| **L** | [Liskov substitution principle](https://en.wikipedia.org/wiki/Liskov_substitution_principle) | [Принцип подстановки Барбары Лисков](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BD%D1%86%D0%B8%D0%BF_%D0%BF%D0%BE%D0%B4%D1%81%D1%82%D0%B0%D0%BD%D0%BE%D0%B2%D0%BA%D0%B8_%D0%91%D0%B0%D1%80%D0%B1%D0%B0%D1%80%D1%8B_%D0%9B%D0%B8%D1%81%D0%BA%D0%BE%D0%B2)  *Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом.*  “objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.” See also design by contract. |
| **I** | [Interface segregation principle](https://en.wikipedia.org/wiki/Interface_segregation_principle) | [Принцип разделения интерфейса](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BD%D1%86%D0%B8%D0%BF_%D1%80%D0%B0%D0%B7%D0%B4%D0%B5%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F_%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81%D0%B0)  Много специализированных интерфейсов лучше, чем один универсальный.  “many client-specific interfaces are better than one general-purpose interface.”[8] |
| **D** | [Dependency inversion principle](https://en.wikipedia.org/wiki/Dependency_inversion_principle) | [Принцип инверсии зависимостей](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BD%D1%86%D0%B8%D0%BF_%D0%B8%D0%BD%D0%B2%D0%B5%D1%80%D1%81%D0%B8%D0%B8_%D0%B7%D0%B0%D0%B2%D0%B8%D1%81%D0%B8%D0%BC%D0%BE%D1%81%D1%82%D0%B5%D0%B9)  Зависимости внутри системы строятся на основе абстракций. Модули верхнего уровня не зависят от модулей нижнего уровня. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.  one should “Depend upon Abstractions. Do not depend upon concretions.”[8] |

**SRP** - Martin defines a responsibility as a *reason to change*, and concludes that a class or module should have one, and only one, reason to change. As an example, consider a module that compiles and prints a report. Imagine such a module can be changed for two reasons. First, the content of the report could change. Second, the format of the report could change. These two things change for very different causes; one substantive, and one cosmetic. The single responsibility principle says that these two aspects of the problem are really two separate responsibilities, and should therefore be in separate classes or modules. It would be a bad design to couple two things that change for different reasons at different times.

The reason it is important to keep a class focused on a single concern is that it makes the class more robust. Continuing with the foregoing example, if there is a change to the report compilation process, there is greater danger that the printing code will break if it is part of the same class.

The responsibility is defined as *a charge assigned to a unique actor to signify its accountabilities concerning a unique business task*.[[4]](https://en.wikipedia.org/wiki/Single_responsibility_principle" \l "cite_note-4)

**OCP - open/closed principle** states "*software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification*";[[1]](https://en.wikipedia.org/wiki/Open/closed_principle" \l "cite_note-1) that is, such an entity can allow its behaviour to be extended without modifying its [source code](https://en.wikipedia.org/wiki/Source_code). This is especially valuable in a production environment, where changes to source code may necessitate [code reviews](https://en.wikipedia.org/wiki/Code_reviews" \o "Code reviews), [unit tests](https://en.wikipedia.org/wiki/Unit_tests" \o "Unit tests), and other such procedures to qualify it for use in a product: code obeying the principle doesn't change when it is extended, and therefore needs no such effort.

The name *open/closed principle* has been used in two ways. Both ways use [inheritance](https://en.wikipedia.org/wiki/Inheritance_%28computer_science%29" \o "Inheritance (computer science)) to resolve the apparent dilemma, but the goals, techniques, and results are different.

## Meyer's open/closed principle

[Bertrand Meyer](https://en.wikipedia.org/wiki/Bertrand_Meyer) is generally credited for having originated the term *open/closed principle*,[[2]](https://en.wikipedia.org/wiki/Open/closed_principle" \l "cite_note-2) which appeared in his 1988 book [*Object Oriented Software Construction*](https://en.wikipedia.org/wiki/Object-Oriented_Software_Construction). The idea was that once completed, the implementation of a class could only be modified to correct errors; new or changed features would require that a different class be created. That class could reuse coding from the original class through [inheritance](https://en.wikipedia.org/wiki/Inheritance_%28computer_science%29" \o "Inheritance (computer science)). The derived subclass might or might not have the same [interface](https://en.wikipedia.org/wiki/Protocol_%28object-oriented_programming%29" \o "Protocol (object-oriented programming)) as the original class.

Meyer's definition advocates [implementation inheritance](https://en.wikipedia.org/wiki/Implementation_inheritance" \o "Implementation inheritance). Implementation can be reused through inheritance but interface specifications need not be. The existing implementation is closed to modifications, and new implementations need not implement the existing interface.

## Polymorphic open/closed principle

During the 1990s, the open/closed principle became popularly redefined to refer to the use of [abstracted](https://en.wikipedia.org/wiki/Abstraction_%28computer_science%29" \o "Abstraction (computer science)) interfaces, where the implementations can be changed and multiple implementations could be created and [polymorphically](https://en.wikipedia.org/wiki/Polymorphism_%28computer_science%29" \o "Polymorphism (computer science)) substituted for each other.

In contrast to Meyer's usage, this definition advocates inheritance from [abstract base classes](https://en.wikipedia.org/wiki/Abstract_base_class" \o "Abstract base class). Interface specifications can be reused through inheritance but implementation need not be. The existing interface is closed to modifications and new implementations must, at a minimum, implement that interface.

[Robert C. Martin](https://en.wikipedia.org/wiki/Robert_C._Martin" \o "Robert C. Martin)'s 1996 article "The Open-Closed Principle"[[3]](https://en.wikipedia.org/wiki/Open/closed_principle" \l "cite_note-3) was one of the seminal writings to take this approach. In 2001 [Craig Larman](https://en.wikipedia.org/wiki/Craig_Larman" \o "Craig Larman) related the open/closed principle to the pattern by [Alistair Cockburn](https://en.wikipedia.org/wiki/Alistair_Cockburn" \o "Alistair Cockburn) called *Protected Variations*, and to the [David Parnas](https://en.wikipedia.org/wiki/David_Parnas" \o "David Parnas) discussion of *[information hiding](https://en.wikipedia.org/wiki/Information_hiding" \o "Information hiding)*.[[4]](https://en.wikipedia.org/wiki/Open/closed_principle" \l "cite_note-4)

**LSP - Substitutability** is a principle in [object-oriented programming](https://en.wikipedia.org/wiki/Object-oriented_programming" \o "Object-oriented programming). It states that, in a [computer program](https://en.wikipedia.org/wiki/Computer_program" \o "Computer program), if S is a [subtype](https://en.wikipedia.org/wiki/Subtype" \o "Subtype) of T, then objects of [type](https://en.wikipedia.org/wiki/Datatype" \o "Datatype) T may be replaced with objects of type S (i.e., objects of type S may *substitute* objects of type T) without altering any of the desirable properties of that program (correctness, task performed, etc.). More formally, the **Liskov substitution principle** (**LSP**) is a particular definition of a [subtyping](https://en.wikipedia.org/wiki/Subtyping" \o "Subtyping) relation, called **(strong) behavioral subtyping**, that was initially introduced by [Barbara Liskov](https://en.wikipedia.org/wiki/Barbara_Liskov" \o "Barbara Liskov) in a 1987 conference [keynote](https://en.wikipedia.org/wiki/Keynote" \o "Keynote) address entitled *Data abstraction and hierarchy*. It is a [semantic](https://en.wikipedia.org/wiki/Formal_semantics_of_programming_languages" \o "Formal semantics of programming languages) rather than merely syntactic relation because it intends to guarantee semantic interoperability of [types](https://en.wikipedia.org/wiki/Data_type" \o "Data type) in a hierarchy, [object types](https://en.wikipedia.org/wiki/Object_%28computer_science%29" \o "Object (computer science)) in particular. [Barbara Liskov](https://en.wikipedia.org/wiki/Barbara_Liskov" \o "Barbara Liskov) and [Jeannette Wing](https://en.wikipedia.org/wiki/Jeannette_Wing" \o "Jeannette Wing) formulated the principle succinctly in a 1994 paper as follows:

*Let Φ(x) be a property provable about objects x of type T. Then Φ(y) should be true for objects y of type S where S is a subtype of T.*

In the same paper, Liskov and Wing detailed their notion of behavioral subtyping in an extension of [Hoare logic](https://en.wikipedia.org/wiki/Hoare_logic" \o "Hoare logic), which bears a certain resemblance with [Bertrand Meyer](https://en.wikipedia.org/wiki/Bertrand_Meyer" \o "Bertrand Meyer)'s [Design by Contract](https://en.wikipedia.org/wiki/Design_by_Contract" \o "Design by Contract) in that it considers the interaction of subtyping with [preconditions](https://en.wikipedia.org/wiki/Precondition" \o "Precondition), [postconditions](https://en.wikipedia.org/wiki/Postcondition" \o "Postcondition) and [invariants](https://en.wikipedia.org/wiki/Invariant_%28computer_science%29" \o "Invariant (computer science)).

## Principle

Liskov's notion of a behavioral subtype defines a notion of substitutability for [mutable](https://en.wikipedia.org/wiki/Mutable" \o "Mutable) objects; that is, if *S* is a subtype of *T*, then objects of type *T* in a program may be replaced with objects of type *S* without altering any of the desirable properties of that program (e.g., [correctness](https://en.wikipedia.org/wiki/Correctness_%28computer_science%29" \o "Correctness (computer science))).

Behavioral subtyping is a stronger notion than typical [subtyping of functions](https://en.wikipedia.org/wiki/Subtyping_of_functions" \o "Subtyping of functions) defined in [type theory](https://en.wikipedia.org/wiki/Type_theory" \o "Type theory), which relies only on the [contravariance](https://en.wikipedia.org/wiki/Covariance_and_contravariance_%28computer_science%29" \o "Covariance and contravariance (computer science)) of argument types and [covariance](https://en.wikipedia.org/wiki/Covariance_and_contravariance_%28computer_science%29" \o "Covariance and contravariance (computer science)) of the return type. Behavioral subtyping is trivially [undecidable](https://en.wikipedia.org/wiki/Undecidable_problem" \o "Undecidable problem) in general: if *q* is the property "method for *x* [always terminates](https://en.wikipedia.org/wiki/Halting_problem" \o "Halting problem)", then it is impossible for a program (e.g., a compiler) to verify that it holds true for some subtype *S* of *T*, even if *q* does hold for *T*. Nonetheless, the principle is useful in reasoning about the design of class hierarchies.

Liskov's principle imposes some standard requirements on [signatures](https://en.wikipedia.org/wiki/Type_signature" \o "Type signature) which have been adopted in newer object-oriented programming languages (usually at the level of classes rather than types; see [nominal vs. structural subtyping](https://en.wikipedia.org/wiki/Subtype" \l "Subtyping_schemes" \o "Subtype) for the distinction):

* [Contravariance](https://en.wikipedia.org/wiki/Covariance_and_contravariance_%28computer_science%29" \o "Covariance and contravariance (computer science)) of method arguments in the subtype.
* [Covariance](https://en.wikipedia.org/wiki/Covariance_and_contravariance_%28computer_science%29" \o "Covariance and contravariance (computer science)) of return types in the subtype.
* No new exceptions should be thrown by methods of the subtype, except where those exceptions are themselves subtypes of exceptions thrown by the methods of the supertype.

In addition to the signature requirements, the subtype must meet a number of behavioral conditions. These are detailed in a terminology resembling that of [design by contract](https://en.wikipedia.org/wiki/Design_by_contract" \o "Design by contract) methodology, leading to some restrictions on how contracts can interact with [inheritance](https://en.wikipedia.org/wiki/Inheritance_%28computer_science%29" \o "Inheritance (computer science)):

* [Preconditions](https://en.wikipedia.org/wiki/Precondition" \o "Precondition) cannot be strengthened in a subtype.
* [Postconditions](https://en.wikipedia.org/wiki/Postcondition" \o "Postcondition) cannot be weakened in a subtype.
* [Invariants](https://en.wikipedia.org/wiki/Invariant_%28computer_science%29" \o "Invariant (computer science)) of the supertype must be preserved in a subtype.
* History constraint (the "history rule"). Objects are regarded as being modifiable only through their methods ([encapsulation](https://en.wikipedia.org/wiki/Encapsulation_%28computer_science%29" \o "Encapsulation (computer science))). Since subtypes may introduce methods that are not present in the supertype, the introduction of these methods may allow state changes in the subtype that are not permissible in the supertype. The history constraint prohibits this. It was the novel element introduced by Liskov and Wing. A violation of this constraint can be exemplified by defining a *mutable point* as a subtype of an *immutable point*. This is a violation of the history constraint, because in the history of the *immutable point*, the state is always the same after creation, so it cannot include the history of a *mutable point* in general. Fields added to the subtype may however be safely modified because they are not observable through the supertype methods. Thus, one can derive *a circle with fixed center but mutable radius* from *immutable point* without violating LSP.

## A typical violation

Further information: [Circle-ellipse problem](https://en.wikipedia.org/wiki/Circle-ellipse_problem" \o "Circle-ellipse problem)

A typical example that violates LSP is a Square class that derives from a Rectangle class, assuming getter and setter methods exist for both width and height. The Square class always assumes that the width is equal with the height. If a Square object is used in a context where a Rectangle is expected, unexpected behavior may occur because the dimensions of a Square cannot (or rather should not) be modified independently. This problem cannot be easily fixed: if we can modify the setter methods in the Square class so that they preserve the Square invariant (i.e., keep the dimensions equal), then these methods will weaken (violate) the [postconditions](https://en.wikipedia.org/wiki/Postcondition" \o "Postcondition) for the Rectangle setters, which state that dimensions can be modified independently. Violations of LSP, like this one, may or may not be a problem in practice, depending on the postconditions or invariants that are actually expected by the code that uses classes violating LSP. Mutability is a key issue here. If Square and Rectangle had only getter methods (i.e., they were [immutable objects](https://en.wikipedia.org/wiki/Immutable_object" \o "Immutable object)), then no violation of LSP could occur.

**ISP -** The **interface-segregation principle** (ISP) states that no client should be forced to depend on methods it does not use.[[1]](https://en.wikipedia.org/wiki/Interface_segregation_principle" \l "cite_note-ASD-1) ISP splits interfaces which are very large into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them. Such shrunken interfaces are also called *role interface*s.[[2]](https://en.wikipedia.org/wiki/Interface_segregation_principle" \l "cite_note-RoleInterface-2) ISP is intended to keep a system decoupled and thus easier to refactor, change, and redeploy. ISP is one of the five [SOLID](https://en.wikipedia.org/wiki/Solid_%28object-oriented_design%29" \o "Solid (object-oriented design)) principles of Object-Oriented Design, similar to the High Cohesion Principle of [GRASP](https://en.wikipedia.org/wiki/GRASP_%28object-oriented_design%29" \o "GRASP (object-oriented design)).[[3]](https://en.wikipedia.org/wiki/Interface_segregation_principle" \l "cite_note-CB-3)

Within [object-oriented design](https://en.wikipedia.org/wiki/Object-oriented_design" \o "Object-oriented design), [interfaces](https://en.wikipedia.org/wiki/Interface_%28computer_science%29" \o "Interface (computer science)) provide layers of abstraction that facilitate conceptual explanation of the code and create a barrier preventing [dependencies](https://en.wikipedia.org/wiki/Dependency_inversion_principle" \o "Dependency inversion principle).

According to many software experts who have signed the Manifesto for [Software Craftsmanship](https://en.wikipedia.org/wiki/Software_Craftsmanship" \o "Software Craftsmanship), writing well-crafted and self-explanatory software is almost as important as writing working software.[[4]](https://en.wikipedia.org/wiki/Interface_segregation_principle" \l "cite_note-SC-4) Using interfaces to further describe the intent of the software is often a good idea.

A system may become so coupled at multiple levels that it is no longer possible to make a change in one place without necessitating many additional changes.[[1]](https://en.wikipedia.org/wiki/Interface_segregation_principle" \l "cite_note-ASD-1) Using an interface or an [abstract class](https://en.wikipedia.org/wiki/Abstract_class" \o "Abstract class) can prevent this side effect

A more commonly known example is the ATM Transaction example given in Agile Software Development: Principles, Patterns, and Practices [[1]](https://en.wikipedia.org/wiki/Interface_segregation_principle#cite_note-ASD-1) and in an article also written by [Robert C. Martin](https://en.wikipedia.org/wiki/Robert_C._Martin) specifically about the ISP.[[5]](https://en.wikipedia.org/wiki/Interface_segregation_principle#cite_note-ISPArticle-5) This example is about an interface for the User Interface for an ATM, that handles all requests such as a deposit request, or a withdrawal request, and how this interface needs to be segregated into individual and more specific interfaces.

**DIP** - **dependency inversion principle** refers to a specific form of [decoupling](https://en.wikipedia.org/wiki/Coupling_%28computer_programming%29) software [modules](https://en.wikipedia.org/wiki/Modular_programming). When following this principle, the conventional [dependency](https://en.wikipedia.org/wiki/Dependency_%28computer_science%29) relationships established from high-level, policy-setting modules to low-level, dependency modules are inverted (i.e. reversed), thus rendering high-level modules independent of the low-level module implementation details. The principle states:[[1]](https://en.wikipedia.org/wiki/Dependency_inversion_principle" \l "cite_note-1)

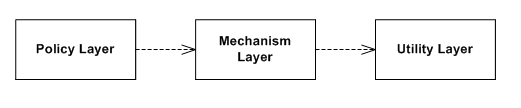
*A. High-level modules should not depend on low-level modules. Both should depend on [abstractions](https://en.wikipedia.org/wiki/Abstraction_%28computer_science%29" \o "Abstraction (computer science)).*

*B. Abstractions should not depend on details. Details should depend on abstractions.*

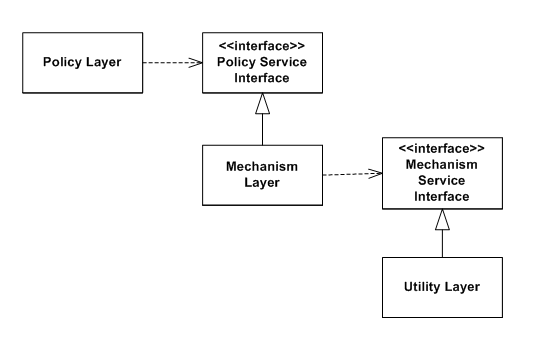
The principle *inverts* the way some people may think about object-oriented design, dictating that *both* high- and low-level objects must depend on the same abstraction

## Traditional layers pattern

In conventional application architecture, lower-level components are designed to be consumed by higher-level components which enable increasingly complex systems to be built. In this composition, higher-level components depend directly upon lower-level components to achieve some task. This dependency upon lower-level components limits the reuse opportunities of the higher-level components.[[3]](https://en.wikipedia.org/wiki/Dependency_inversion_principle" \l "cite_note-3)

[](https://en.wikipedia.org/wiki/File:Traditional_Layers_Pattern.png)The goal of the dependency inversion principle is to avoid this highly coupled distribution with the mediation of an abstract layer, and to increase the re-usability of higher/policy layers.

## Ownership inversion

[](https://en.wikipedia.org/wiki/File:DIPLayersPattern.png)With the addition of an abstract layer, both high- and lower-level layers avoid the traditional dependencies from top to bottom. Nevertheless the ″inversion″ concept does not mean that lower-level layers depend on higher-level layers. Both layers should depend on abstractions that draw the behavior needed by higher-level layers.

In a direct application of dependency inversion, the abstracts are owned by the upper/policy layers. This architecture groups the higher/policy components and the abstracts that define lower services together in the same package. The lower-level layers are created by inheritance/implementation of these abstract classes or interfaces.[[4]](https://en.wikipedia.org/wiki/Dependency_inversion_principle" \l "cite_note-4)

The inversion of the dependencies and ownership encourages the re-usability of the higher/policy layers. Upper layers could use other implementations of the lower services. When the lower-level layer components are closed or when the application requires the reuse of existing services, it is common that an [Adapter](https://en.wikipedia.org/wiki/Adapter_pattern" \o "Adapter pattern) mediates between the services and the abstractions.

## Abstraction dependency

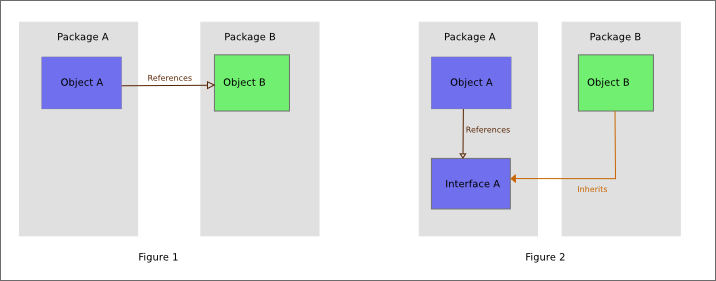
The presence of abstractions to accomplish DIP have other design implications in an [Object Oriented](https://en.wikipedia.org/wiki/Object_Oriented) program:

* All member variables in a class must be interfaces or abstracts.
* All concrete class packages must connect only through interface/abstract classes packages.
* No class should derive from a concrete class.
* No method should override an implemented method.[[5]](https://en.wikipedia.org/wiki/Dependency_inversion_principle" \l "cite_note-5)
* All variable instantiation requires the implementation of a [Creational pattern](https://en.wikipedia.org/wiki/Creational_pattern" \o "Creational pattern) as the [Factory Method](https://en.wikipedia.org/wiki/Factory_method_pattern" \o "Factory method pattern) or the [Factory](https://en.wikipedia.org/wiki/Factory_%28object-oriented_programming%29" \o "Factory (object-oriented programming)) pattern, or the more complex use of a [Dependency Injection](https://en.wikipedia.org/wiki/Dependency_Injection" \o "Dependency Injection) framework.

## DIP implementations

Two common implementations of DIP use similar logical architecture, with different implications.

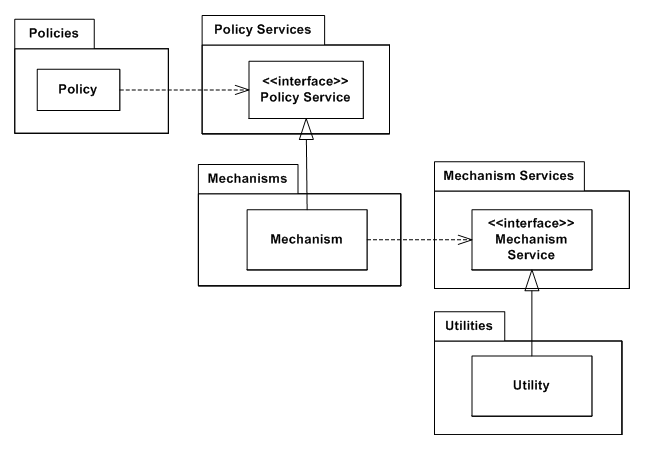
A direct implementation packages the policy classes with service abstracts classes in one library. In this implementation high-level components and low-level components are distributed into separate packages/libraries, where the [interfaces](https://en.wikipedia.org/wiki/Interface_%28computer_science%29" \o "Interface (computer science)) defining the behavior/services required by the high-level component are owned by, and exist within the high-level component's library. The implementation of the high-level component's interface by the low level component requires that the low-level component package depend upon the high-level component for compilation, thus inverting the conventional dependency relationship.

[](https://en.wikipedia.org/wiki/File:Dependency_inversion.png)

Figures 1 and 2 illustrate code with the same functionality, however in Figure 2, an interface has been used to invert the dependency. The direction of dependency can be chosen to maximize policy code reuse, and eliminate cyclic dependencies.

In this version of DIP, the lower layer component's dependency on the interfaces/abstracts in the higher-level layers makes re-utilization of the lower layer components difficult. This implementation instead ″inverts″ the traditional dependency from top-to-bottom to the opposite from bottom-to-top.

A more flexible solution extracts the abstract components into an independent set of packages/libraries:

[](https://en.wikipedia.org/wiki/File:DIPLayersPattern_v2.png)The separation of all layers into their own package encourages re-utilization of any layer, providing robustness and mobility.[[6]](https://en.wikipedia.org/wiki/Dependency_inversion_principle#cite_note-6)

## Related patterns

Applying the dependency inversion principle can also be seen as an example of the [Adapter pattern](https://en.wikipedia.org/wiki/Adapter_pattern), i.e. the high-level class defines its own adapter interface which is the abstraction that the other high-level classes depend on. The adaptee implementation also depends on the adapter interface abstraction (of course, since it implements its interface) while it can be implemented by using code from within its own low-level module. The high-level has no dependency on the low-level module since it only uses the low-level indirectly through the adapter interface by invoking polymorphic methods to the interface which are implemented by the adaptee and its low-level module.

Various patterns such as [Plugin](https://en.wikipedia.org/wiki/Plug-in_%28computing%29), [Service Locator](https://en.wikipedia.org/wiki/Service_locator_pattern), or [Dependency Injection](https://en.wikipedia.org/wiki/Dependency_Injection) are employed to facilitate the run-time provisioning of the chosen low-level component implementation to the high-level component.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

In “The Dependency Inversion Principle” (or **DIP**), the author states the three defining factors of “bad code”:

1. It is hard to change because every change affects too many other parts of the system (Rigidity)
2. When you make a change, unexpected parts of the system break (Fragility)
3. It is hard to reuse in another application because it cannot be disentangled from the current application (Immobility) [[3](http://www.springbyexample.org/examples/core-concepts.html" \l "ftn.d0e237)]

According to Martin, interdependency causes these coding problems (we'll call them **RFI** for Rigidity, Fragility, and Immobility). To fix RFI issues in your OO code, DIP has two basic rules:

1. High level modules should not depend upon low level modules, both should depend upon abstractions.

In other words, high level modules – which contain your business logic and all of the important meat of your application – should not depend on lower level components. The reason for this is if these lower level components were to change, the changes might affect the higher level components as well. This is the defining concept behind dependency inversion, that the prevailing wisdom of having higher-level modules dependent on lower-level modules is in fact a bad idea.

2. Abstractions should not depend upon details, details should depend upon abstractions.

This is another way to say that before you begin coding to the abstraction – the interface or abstract class -- you should find the common behaviors in the code and work backwards. Your interface /abstraction should cater to the intersection between the needs of your business logic and the common behaviors of the lower level modules. You should also leave the details of how these behaviors are implemented to the implementation classes.