*Law of Demeter also known as principle of least knowledge is a coding principle, which says that a module should not know about the inner details of the objects it manipulates. If a code depends upon internal details of a particular object, there is good chance that it will break as soon as internal of that object changes. Since*[*Encapsulation*](http://javarevisited.blogspot.com/2012/03/what-is-encapsulation-in-java-and-oops.html)*is all about hiding internal details of object and exposing only*[*operations*](http://javarevisited.blogspot.com/2014/05/law-of-demeter-example-in-java.html)*, it also assert Law of  Demeter. One mistake many Java programmer makes it exposing internal detail of object using getter methods and this is where principle of least knowledge alerts you. I first come to know about this principle, while reading one of the*[*must read programming book*](http://javarevisited.blogspot.com/2013/05/which-programming-book-would-you-buy-if.html)*, Robert C. Martin's Clean code. Apart from many good thing the book teaches you,****"principle of least knowledge"****is one principle, which I still remember. Like many bad things, you will tempt to violate Law of Demeter, because of beautiful chaining of methods written in fluent style. On surface it looks pretty good, but as soon as you think about principle of least knowledge, you start seeing the real picture. In this article, we will see formal definition of Law of Demeter and explore code snippet which violates this principle.*

***Law of Demeter***

*According to Law of Demeter, a method M of object O should only call following types of methods:*

1. *Methods of Object O itself*
2. *Methods of Object passed as an argument*
3. *Method of object, which is held in instance variable*
4. *Any Object which is created locally in method M*

*More importantly method should not invoke methods on objects that are returned by any subsequent method calls specified above and as Clean Code says "talk to friends, not to strangers". Apart from knowing object oriented programming basic concepts e.g.*[*Abstraction*](http://javarevisited.blogspot.sg/2010/10/abstraction-in-java.html)*,*[*Polymorphism*](http://javarevisited.blogspot.sg/2011/08/what-is-polymorphism-in-java-example.html)*,*[*Inheritance*](http://java67.blogspot.com/2012/08/what-is-inheritance-in-java-oops-programming-example.html)*and*[*SOLID design principle*](http://javarevisited.blogspot.sg/2012/03/10-object-oriented-design-principles.html)*, it's also worth knowing useful principle like this, which has found its way via experience. In following example, we will see how a method can violate above rules to violate Law of Delimiter.*

public class LawOfDelimterDemo {

    /\*\*

     \* This method shows two violations of "Law of Delimiter" or "Principle of least knowledge".

     \*/

    public void process(Order o) {

        // as per rule 1, this method invocation is fine, because o is a argument of process() method

        Message msg = o.getMessage();

        // this method call is a violation, as we are using msg, which we got from Order.

        // We should ask order to normalize message, e.g. "o.normalizeMessage();"

        msg.normalize();

        // this is also a violation, instead using temporary variable it uses method chain.

        o.getMessage().normalize();

        // this is OK, a constructor call, not a method call.

        Instrument symbol = new Instrument();

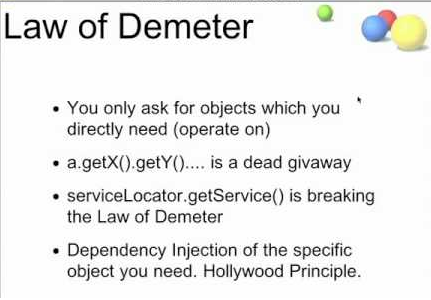
        // as per rule 4, this method call is OK, because instance of Instrument is created locally.

        symbol.populate();

    }

}

*You can see that when we get internal of Order class and  call a method on that object, we violate Law of delimiter, because now this method knows about Message class. On the other hand calling method on Order object is fine because its passed to the method as parameter.  This image nicely explains what you need to do to follow Law of Demeter.*

[](http://2.bp.blogspot.com/-eyIv7MV4CDw/U4XkaYxuyuI/AAAAAAAABj4/RbNXifaTPaE/s1600/Law+of+Demeter+in+Java+with+Example.jpg)

*Let's see another example of code, which violates the Law of Demeter and how does it affect code quality.*

public class XMLUtils {

public Country getFirstBookCategoryFromXML(XMLMessage xml) {

return xml.getXML().getBooks().getBookArrary(0).getBookHeader().getBookCategory();

}

}

*This code is now dependent upon lot of classes e.g.*    XMLMessage  
    XML  
    Book  
    BookHeader  
    BookCategory  
  
*Which means this function knows* *about XMLMessage, XML, Book, BookHeader and BookCategory. It knows that XML has list of Book, which in-turn has BookHeader and which internally has BookCategory, that's a lot of information. If any of the intermediate class or accessor method in this chained method call changes, then this code will break. This code is highly coupled and brittle. It's much better to put the*[*responsibilityhttp://images.intellitxt.com/ast/adTypes/icon1.png*](http://javarevisited.blogspot.com/2014/05/law-of-demeter-example-in-java.html)*of finding internal data into the object, which owns it. If we look closely, we should only call getXML() method because its method from XMLMessage class, which is passed to method as argument.  Instead of putting all this code inXMLUtils, should be putting on BookUtils or something similar, which can still follow Law of Demeter and can return the required information.  
  
That's all on this example of Law of Demeter or "Principle of least knowledge". It's better not to have a chain of methods, originating from unknown object, which may change. I also suggest to read Chapter 6 of Clean Code, a must read book for any software developers.  
  
Want to read more on design principles and patterns?*[*Checkhttp://images.intellitxt.com/ast/adTypes/icon1.png*](http://javarevisited.blogspot.com/2014/05/law-of-demeter-example-in-java.html)*out these amazing articles*

* *Real life example of Open Close design principle in Java. (*[*read here*](http://javarevisited.blogspot.sg/2011/11/great-example-of-open-closed-design.html)*)*
* *How to use Decorator pattern in Java? (*[*read here*](http://javarevisited.blogspot.com/2011/11/decorator-design-pattern-java-example.html)*)*
* *When to use Builder pattern in Java? (*[*read here*](http://javarevisited.blogspot.sg/2012/06/builder-design-pattern-in-java-example.html)*)*
* *How double checked locking of Singleton works in Java? (*[*see here*](http://javarevisited.blogspot.sg/2014/05/double-checked-locking-on-singleton-in-java.html)*)*
* *Observer Pattern in Java with Example (*[*see here*](http://javarevisited.blogspot.sg/2011/12/observer-design-pattern-java-example.html)*)*
* *Difference between Factory and Abstract Factory pattern in Java? (*[*see here*](http://javarevisited.blogspot.sg/2013/01/difference-between-factory-and-abstract-factory-design-pattern-java.html)*)*
* *When to use Factory method pattern? (*[*see here*](http://javarevisited.blogspot.com/2011/12/factory-design-pattern-java-example.html)*)*
* *Difference between*[*Strategyhttp://images.intellitxt.com/ast/adTypes/icon1.png*](http://javarevisited.blogspot.com/2014/05/law-of-demeter-example-in-java.html)*and State design Pattern? (*[*check here*](http://javarevisited.blogspot.sg/2014/04/difference-between-state-and-strategy-design-pattern-java.html)*)*
* *Why Enum as Singleton is better pattern? (*[*check here*](http://javarevisited.blogspot.gr/2012/07/why-enum-singleton-are-better-in-java.html)*)*
* *Difference between Singleton and Static class in Java? (*[*check here*](http://javarevisited.blogspot.sg/2013/03/difference-between-singleton-pattern-vs-static-class-java.html)*)*
* *How to create thread-safe Singleton in Java? (*[*link*](http://javarevisited.blogspot.sg/2012/12/how-to-create-thread-safe-singleton-in-java-example.html)*)*
* *20 Software design and pattern Interview question for Java programmer? (*[*link*](http://javarevisited.blogspot.sg/2012/06/20-design-pattern-and-software-design.html)*)*
* *Dependency Injection and Inversion of Control in Java (*[*link*](http://javarevisited.blogspot.com/2012/12/inversion-of-control-dependency-injection-design-pattern-spring-example-tutorial.html)*)*
* *How to make a class Immutable in Java? (*[*link*](http://javarevisited.blogspot.sg/2013/03/how-to-create-immutable-class-object-java-example-tutorial.html)*)*

Read more: <http://javarevisited.blogspot.com/2014/05/law-of-demeter-example-in-java.html#ixzz41fhQ431e>