Law of Demeter (LoD)

Introduction to Law of Demeter Principle

The **Law of Demeter**, also known as the **Principle of Least Knowledge**, is a design guideline that encourages minimizing the knowledge a class has about the structure or details of other classes. The idea is that a given object should only "talk" to its immediate friends and not deeply nested internal objects or objects returned by other objects.

Why Law of Demeter is Important

1. **Reduces Coupling:** By limiting the interactions between objects, you reduce the dependencies between classes, leading to more modular and maintainable code.

2. **Increases Encapsulation:** Each class knows only about its immediate collaborators, which helps in encapsulating changes within a class without affecting others.

3. **Improves Readability:** Code that adheres to LoD is generally easier to read and understand because the interactions are simpler and more straightforward.

How to Apply Law of Demeter

- **Only Call Methods on:**
  - The object itself (`this`).
  - Objects passed as parameters to the method.
  - Objects created within the method.
  - Directly held instance variables (but not their internal objects).

Example: Without Law of Demeter

```
public class Address {

    private String street;


    public String getStreet() {

        return street;
```

```
    }
}


public class Customer {
    private Address address;

    public Address getAddress() {
        return address;
    }
}

public class Order {
    private Customer customer;

    public String getCustomerStreet() {
        return customer.getAddress().getStreet(); // Violates LoD
    }
}
```

In this example, the `Order` class violates the Law of Demeter because it directly accesses the `Address` object through the `Customer` object. This creates a dependency chain that can make the code harder to maintain and understand.

Example: With Law of Demeter

```
public class Address {
    private String street;
```

```
    public String getStreet() {
        return street;
    }
}

public class Customer {
    private Address address;

    public String getCustomerStreet() {
        return address.getStreet(); // Customer provides street directly
    }
}

public class Order {
    private Customer customer;

    public String getCustomerStreet() {
        return customer.getCustomerStreet(); // Adheres to LoD
    }
}
```

In this refactored version, the `Order` class interacts only with the `Customer` object and asks it for the street information, adhering to the Law of Demeter. This keeps the `Order` class ignorant of the internal structure of the `Customer` class.

Key Takeaways

- **Limit Object Interactions:** Restrict your objects to interacting only with closely related objects to reduce dependencies and improve modularity.

- **Delegate Responsibility:** Push the responsibility of fetching or processing data to the object that owns it, rather than accessing deep object structures.
- **Promote Encapsulation:** By adhering to LoD, you ensure that changes to the internal structure of one class don't ripple through the entire codebase.

When LoD Might Not Apply

While LoD is generally a good principle to follow, there are cases where strict adherence might lead to unnecessary code bloat:

- **Utility Classes:** In some cases, especially with utility classes or methods, strict adherence to LoD can make the code overly verbose.
- **Performance Considerations:** In rare cases, following LoD might lead to additional method calls that could impact performance, though this is usually negligible.

Would you like to explore further examples or discuss scenarios where applying the Law of Demeter might be challenging?