Law of Demeter

Have you ever called a method on an object… then chained another… and another… until the line looked like a trail of dots?

Or made a small internal change to one class… and suddenly had to update code across five other layers?

If yes, you've probably run into a violation of one of the most overlooked design principles in software engineering: **The Law of Demeter (LoD)**.

Let's understand it with a real-world example—and see why this principle matters more than you might think.

The Problem

Imagine you're building a simple e-commerce system.

You have:

- A `Customer` who owns a `ShoppingCart`
- The cart contains a list of `CartItems`
- Each `CartItem` refers to a `Product`
- And every `Product` has a `Price`

Now let's say you want to display the price of the **first product** in a customer's shopping cart.

A common (but flawed) way to write this would be:

Java

```
Money price =
customer.getShoppingCart().getItems().get(0).getProduct().getPri
ce();
```

- **Python**

```
price =
customer.get_shopping_cart().get_items()[0].get_product().get_pr
ice()
```

And a complete version might look like this:

Java

```
void 1displayFirstItemPrice(Customer customer) {
2    Money price =
customer.getShoppingCart().getItems().get(0).getProduct().getPri
ce();
3    System.out.println("Price of the first item: " +
price.getAmount());
4}
```

```
Python
def display_first_item_price(customer):
    price =
customer.get_shopping_cart().get_items()[0].get_product().get_pr
ice()
    print("Price of the first item:", price.get_amount())
```

This approach works, but it **smells bad**.

It's what we call a **"train wreck"** or **"dot-chaining"**: one object reaching through several others to get what it wants.

You start with a `Customer`, go into their `ShoppingCart`, peek into its internal list of `CartItems`, grab the first one, extract the `Product`, and finally—get the `Price`.

That's a long dependency chain. And it's fragile.

What's Wrong With This?

1. High Coupling

The `OrderService` method is now tightly coupled to the **entire internal structure** of the customer and their cart.

- If `ShoppingCart` changes how it stores items (e.g., using a `Map` instead of a `List`)
- If `CartItem` renames its `getProduct()` method
- Or if `Product` evolves to store pricing in a new way…

Boom. Your `OrderService` code breaks. Even though it had nothing to do with those decisions.

2. Encapsulation Violation

You're reaching deep into object internals—**violating encapsulation at multiple levels**.

- `Customer` exposes its `ShoppingCart`
- `ShoppingCart` exposes its internal list
- You assume the structure of that list
- And even dig through `CartItem` and `Product` just to get a price

This kind of reach-through breaks the principle of object-oriented design: **objects should tell, not ask**.

3. Maintenance Nightmare

Imagine this change: You switch from using a `Money` wrapper to a `BigDecimal` for price representation in `Product`.

Now, every part of your codebase that dot-chased its way to `product.getPrice()` must be updated.

Your implementation detail leaked—and now you're paying for it.

4. Testability Issues

Testing `displayFirstItemPrice()` becomes a *mocking marathon*.

To test it in isolation, you'd need to mock:

- A `Customer`
- That returns a `ShoppingCart`
- That returns a `List`
- That returns a `CartItem`
- That returns a `Product`
- That returns a `Price`

One function. Six mocks. Exhausting.

Enter: The Law of Demeter (LoD)

"Only talk to your immediate friends." — Law of Demeter

The Law of Demeter (also known as the Principle of Least Knowledge) says an object should only call methods on:

- Itself
- Its own fields
- Its method parameters
- Objects it creates

That’s it.

In plain terms: **don't reach through one object to get to another.**

Refactoring with LoD in Mind

Let’s rewrite this in a cleaner, more respectful way. We’ll **push the responsibility down** to the classes that know the most.

Step 1: Add a method to `ShoppingCart`

Java

```
class ShoppingCart {
    // ... existing code

    public Money getFirstItemPrice() {
        if (items.isEmpty()) return Money.ZERO;
        return items.get(0).getProduct().getPrice();
    }
}
```

Python

```
class ShoppingCart:
    # ... existing code

    def get_first_item_price(self):
        if not self.items:
            return Money.ZERO
        return self.items[0].get_product().get_price()
```

Step 2: Add a method to Customer

Java

```
class Customer {
    // ... existing code

    public Money getFirstCartItemPrice() {
        return shoppingCart.getFirstItemPrice();
    }
}
```

Python

```
class Customer:
    # ... existing code

    def get_first_cart_item_price(self):
        return self.shopping_cart.get_first_item_price()
```

Step 3: Update the OrderService

Java

```
void displayFirstItemPrice(Customer customer) {
    Money price = customer.getFirstCartItemPrice();
    System.out.println("Price of the first item: " + price.getAmount());
}
```

Python

```
def display_first_item_price(customer: Customer):
    price = customer.get_first_cart_item_price()
    print("Price of the first item:", price.get_amount())
```

Much better, right?

Now OrderService doesn’t care about:

How the cart stores items
What a CartItem contains
How the Product holds price
It just asks the Customer for what it needs.

Benefits of Law of Demeter
Low Coupling: Code changes in one place don’t ripple across your codebase.
Better Encapsulation: Each class handles its own logic—no more peeking into internals.
Easier Refactoring: You can evolve internal implementations without affecting consumers.
Improved Testability: Fewer mocks needed. More focused tests.
Cleaner APIs: Public methods are expressive, intentional, and meaningful.
Common Questions About LoD
Isn't this just more code? I have to write all these wrapper methods!

Yes, following LoD can result in additional small, delegating methods. But this “extra” code serves a critical purpose: it reduces coupling.

Think of it this way—would you rather write 3 lines now to isolate behavior, or refactor 300 later when your system breaks?

These small wrappers protect the rest of your codebase from internal changes. They enforce the principle of “Tell, Don’t Ask”—you tell an object what you want it to do, instead of reaching inside and doing it yourself.

Does the Law of Demeter mean I can’t use getters at all?
Not at all. LoD doesn’t forbid getters.

Simple property access like customer.getName() is perfectly fine—name is a direct part of Customer.

The issue arises with chained getters across object boundaries:

customer.getCart().getItems().get(0).getProduct().getPrice();
This creates tight coupling between the caller and the internal structure of several unrelated classes. Instead, you’re encouraged to delegate the operation to the object that owns the knowledge.

What about data structures? Can I call size() on a list I get from an object?
This is a nuanced area.

If getUsers() returns a standard collection like List<User>, then getUsers().size() is generally acceptable. Lists are transparent and well-understood abstractions, and operations like size() don’t break encapsulation.

However, this would be a violation:

getUsers().get(0).getAddress().getStreet();
The more layers of domain objects you traverse, the more you're violating LoD.The key is whether you're interacting with a simple data structure or delving into another object’s responsibility chain.

When is it okay to “violate” the Law of Demeter?
Like most principles, LoD is a guideline—not a hard rule. Some common exceptions include:

DTOs / Value Objects: It’s acceptable to traverse simple data carriers where behavior isn't expected.
Stable, Low-Level Libraries: Using well-known APIs (like Map.get() or List.size()) is typically safe.

Fluent APIs / Builders: Method chaining in fluent interfaces is usually an intentional design, not a violation.
The key is intentional design. If you understand the coupling trade-off and still find it justifiable—go for it. Just don't do it by accident.

How does LoD relate to other SOLID principles like SRP or Encapsulation?
LoD works hand-in-hand with other key design principles:

Encapsulation: LoD reinforces encapsulation by discouraging external code from depending on internal structure.
Single Responsibility Principle (SRP): LoD encourages putting logic where it belongs. If OrderService needs pricing logic, LoD nudges you to push it into ShoppingCart or Customer—where the relevant data already lives.
Low Coupling: This is the central theme. By limiting how far your code reaches, you reduce fragility and make your system easier to maintain, extend, and test.
In short: LoD is like a guardrail. It helps you avoid the slippery slope of exposing internals and tying your code together too tightly.