DRY (Don't Repeat Yourself)

Introduction to DRY Principle

The **DRY (Don't Repeat Yourself)** principle is a fundamental concept in software engineering that encourages reducing repetition of code and logic in a codebase. The idea is to ensure that every piece of knowledge or logic in a system is represented in a single, unambiguous place. By doing so, you make your code more maintainable, easier to understand, and less error-prone.

Why DRY is Important

1. **Maintainability:** If you need to make a change, you only need to make it in one place. This reduces the risk of introducing bugs.

2. **Readability:** Your code is easier to read and understand because similar logic is not scattered across different parts of the codebase.

3. **Reusability:** When code is modular and not duplicated, it becomes easier to reuse components in different parts of the application.

Common Areas Where DRY Applies

- **Functions/Methods:** Extract repeated logic into functions.

- **Classes/Objects:** Use inheritance or composition to avoid duplicating logic across classes.

- **Database Schemas:** Avoid storing the same piece of data in multiple tables.

- **Configuration:** Centralize configuration settings rather than hardcoding them in multiple places.

Example: Without DRY

```
public class User {

    public void saveToDatabase() {

        // Code to save user data to the database

        System.out.println("Saving user to database");

    }

}
```

```
public class Order {
    public void saveToDatabase() {
        // Code to save order data to the database
        System.out.println("Saving order to database");
    }
}
```

In the example above, both the `User` and `Order` classes have their own `saveToDatabase` method, which likely contains very similar or even identical logic. If you need to change the way data is saved, you?ll have to do it in both places, increasing the risk of errors.

Example: With DRY

```
public class DatabaseManager {
    public void save(Object obj) {
        // Generalized code to save any object to the database
        System.out.println("Saving " + obj.getClass().getSimpleName() + " to database");
    }
}

public class User {
    private DatabaseManager dbManager = new DatabaseManager();

    public void save() {
        dbManager.save(this);
    }
}
```

```
public class Order {

    private DatabaseManager dbManager = new DatabaseManager();


    public void save() {

        dbManager.save(this);

    }

}
```

In this refactored version, the `DatabaseManager` class is responsible for saving objects to the database. Both `User` and `Order` classes delegate the save operation to this centralized class, thereby eliminating the repetition of the save logic.

Key Takeaways

- **Centralize Logic:** By moving repeated logic to a single place, you reduce redundancy and make your code easier to maintain.
- **Use Abstractions:** Abstract common logic into functions, classes, or modules to avoid duplication.
- **Regularly Refactor:** As your codebase grows, regularly refactor to ensure the DRY principle is maintained.

Would you like to explore a more complex example, or perhaps see how DRY applies in different contexts, such as front-end development or database design?