Day 125,127,
SRP & OCP
SRP Violation
1 https://docs.oracle.com/javase/8/docs/api/java/util/package-summary.html

Solid principle read
4 https://github.com/kanmaytacker/fundamentals/blob/master/oop/notes/04-solid-01.md
5 https://github.com/kanmaytacker/fundamentals/blob/master/oop/notes/05-solid-02.md

Further Reads
https://www.boldare.com/blog/solid-cupid-grasp-principles-object-oriented-design/#what-is-solid-and-why-is-it-more-than-just-an-acronym?-solid-vs.-cupid---is-the-new-always-better

Bird problem
https://github.com/kanmaytacker/fundamentals/tree/master/oop/code/oop/src/main/java/com/scaler/lld/bird

https://github.com/kanmaytacker/fundamentals/tree/bird-v3/oop/code/oop/src/main/java/com/scaler/lld/bird

https://github.com/kanmaytacker/fundamentals/tree/bird-v5/oop/code/oop/src/main/java/com/scaler/lld/bird

Homework
https://github.com/kanmaytacker/design-questions/blob/master/notes/01-design-a-pen-worksheet.md

https://github.com/kanmaytacker/design-questions/blob/master/notes/01-design-a-pen.md

Good Software
- Maintainable
1 Debug
2 Test
3 Understandable
- Extensible
- scalable

- Design Principles
Set of rules
Guideline to create good software

SOLID

S - single responsibility principle : Every class/method/package should have one responsibility .
O- open/closed principle : software entities(class,method,module) should be opened for extension but closed for modification.
L- Liskov substitution
I- Interface segregation
D- Dependency Inversor

Single Responsibility Principle

There should never be more than one reason for a class/code unit to change. Every class should have only one responsibility .
When designing our classes, we should aim to put related features together, so whenever they tend to change they change for the same reason. And we should try to separate features if they will change for reasons .

```
public void fly() {
    if (type.equals("eagle")) {
        flyLikeEagle();
    } else if (type.equals("penguin")) {
        flyLikePenguin();
    } else if (type.equals("parrot")) {
        flyLikeParrot();
    }
}
```

- `Readability` - The code is not readable. It is difficult to understand what the code is doing.

- `Testing` - It is difficult to test the code. We would have to test each type of bird separately.
- `Reusability` - The code is not reusable. If we want to re-use the code of specific type of bird, we would have to change the above code.
- `Parallel development` - The code is not parallel development friendly. If multiple developers are working on the same code, they could face merge conflicts.
- `Multiple reasons to change` - The code has multiple reasons to change. If we want to change the way a type of bird flies, we would have to change the code in the `fly` method.

How/Where to spot violations of SRP?

- A method with multiple `if-else` statements. An example would be the `fly` method of the `Bird` class. This is not a silver bullet, but it is a good indicator. There can be other reasons for multiple `if-else` statements such as business logic e.g. calculating the tax, checking access rights, etc.
- `Monster methods` or `God classes` - Methods that are too long and doing much more than the name suggests. This is a good indicator of a violation of SRP.

```
public saveToDatabase() {
    // Connect to database
    // Create a query
    // Execute the query
    // Create a user defined object
    // Close the connection
}
```

The above method is doing much more than the name suggests. It is connecting to the database, creating a query, executing the query, creating a user defined object, and closing the connection. This method violates the SRP. It should be split into multiple methods such as `connectToDatabase`, `createQuery`, `executeQuery`, `createUserDefinedObject`, and `closeConnection`.

open/closed Principle

Software entities (classes,functions,modules etc) should open for extension and closed for modification .

```
public void fly() {
    if (type.equals("eagle")) {
        flyLikeEagle();
    } else if (type.equals("penguin")) {
        flyLikePenguin();
    } else if (type.equals("parrot")) {
        flyLikeParrot();
    }
}
```

In the above code, we are checking the type of the bird and then calling the appropriate method. If we want to add a new type of bird, we would have to change the code in the `fly` method. This is a violation of the Open/Closed Principle.

The Open/Closed Principle states that a class should be open for extension but closed for modification. This means that we should be able to add new functionality to the class without changing the existing code. To add a new feature, we should ideally create a new class or method and have very little or no changes in the existing code. In doing so, we stop ourselves from modifying existing code and causing potential new bugs in an otherwise happy application. We should be able to add new functionality without touching the existing code for the class. This is because whenever we modify the existing code, we are taking the risk of creating potential bugs. So we should avoid touching the tested and reliable (mostly) production code if possible.

How to solve this.

Abstract class

Abstract class is nothing but a class that declares itself by abstract keyword. Can define abstract methods which are forced to implement by subclasses. Abstract classes have static variables, instance variables, instance methods, static methods and abstract methods.

Interface

Interface has only abstract method and final constant.

When to use abstract class / interface.

If entity has common attributes and common behavior then go for abstract class
If common behavior only then for interface.

Fixing OCP violation in the `Bird` class

Now that we have learnt about abstract classes and interfaces, let us fix the SRP and OCP violation in the `Bird` class. In order to fix the SRP violations, we would consider having a parent class `Bird` and child classes `Eagle`, `Penguin`, and `Parrot`. Since, different birds have the same attributes and behaviours, we would want to use classes. An instance of the `Bird` class does not make sense, hence we would use an abstract class. We can't use an interface since we would want to have instance variables. We would also want to have a fixed contract for the subclasses to implement the common functionalities. Hence, we would use an abstract class. Now, our `Bird` class would look like this.

```
classDiagram
    Bird <|-- Eagle
    Bird <|-- Penguin
    Bird <|-- Parrot
    class Bird{
        +weight: int
        +colour: string
        +type: string
        +size: string
        +beakType: string
        +fly()
    }
    class Eagle{
        +fly()
    }
    class Penguin{
        +fly()
    }
    class Parrot{
        +fly()
```

L - Liskov substitution principle
1 The Liskov Substitution Principle states that objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.
2 To identify violations, we can check if we can replace a class with its subclasses having to handle special cases and expect the same behavior.
3 Prefer using interfaces over abstract classes to implement behavior since abstract classes tend to tie behavior to the class hierarchy.

I - Interface segregation principle
Separating means separate things and interface segregation principle about separating the interface.
Clients should never be forced to interface methods which are not related to particular work.
Always use a lean interface rather than a general purpose interface but related methods should be in one interface.

D - Dependency Inversion
High level module should not depend on low level module

Give a read to document of SOLID