

## 💎 Key Takeaways

=====

- ✓ In-depth understanding of SOLID principles
- ✓ Walk-throughs with examples
- ✓ Understand concepts like Dependency Injection, Runtime Polymorphism, ..
- ✓ Practice quizzes & assignment

## ? FAQ

=====

- ▶ Will the recording be available?  
To Scaler students only
- ⇒ Will these notes be available?  
Yes. Published in the discord/telegram groups (link pinned in chat)
- 🕒 Timings for this session?  
5pm – 8pm (3 hours) [15 min break midway]
- 🎧 Audio/Video issues  
Disable Ad Blockers & VPN. Check your internet. Rejoin the session.
- ? Will Design Patterns, topic x/y/z be covered?  
In upcoming masterclasses. Not in today's session.  
Enroll for upcoming Masterclasses @ [scaler.com/events]  
(<https://www.scaler.com/events>)
- 💻 What programming language will be used?  
The session will be language agnostic. I will write code in Java.  
However, the concepts discussed will be applicable across languages
- 💡 Prerequisites?  
Basics of Object Oriented Programming

## 👤 About the Instructor

=====

Pragy  
[[linkedin.com/in/AgarwalPragy/](https://www.linkedin.com/in/AgarwalPragy/)](<https://www.linkedin.com/in/AgarwalPragy/>)

Senior Software Engineer + Instructor @ Scaler

## Important Points

=====

- 💬 Communicate using the chat box
- 👤 Post questions in the "Questions" tab
- 💙 Upvote others' question to increase visibility
- 👍 Use the thumbs-up/down buttons for continuous feedback
- 🕒 Bonus content at the end

-----

```
>
> ? What % of your work time is spend writing new code?
>
> • 10-15%      • 15-40%      • 40-80%      • > 80%
>
```

< 15% of a devs time is spent writing fresh code

🕒 Where does the rest of the time go?

minimize time here

- meetings
- reading/testing/debugging/looking up docs
- business usecases/..

maximize time here

- break – playing TT, having snacks, chai sutta break

Once we have written our code, we NEVER have to go back and fix it.

## ✅ Goals

=====

We'd like to make our code

1. Extensible
2. Testable
3. Maintainable
4. Readable

#### Robert C. Martin 🧔 – Uncle Bob

=====

## 💎 SOLID Principles

=====

- Single Responsibility
- Open/Closed
- Liskov's Substitution
- Interface Segregation
- Dependency Inversion

Inversion of Control / Interface Segregation  
Dependency Inversion / Dependency Injection

## ☁ Context

=====

- Zoo Game – toy examples 🐱
- animals, cages, visitors, staff ..

-----

## 🐼 Design an Animal

=====

```
```java
```

```
class Animal {  
    // classes are used to represent concepts/abstractions/entities/ideas  
  
    // attributes [ properties]  
    String color;  
    String name;  
    String species;  
    Float height;  
}
```

```

Float weight;
Integer numberOfLegs;
Boolean canBreatheUnderwater;
Boolean isDangerous;

// behavior/functionality [methods]
void run(); // declaration – we would like to implement this

void eat();

void swim();

void swingFromTreeBranch();

void speak();
}

// objects are instances of classes

...

```

🐟 Different animals will run in different ways

```

```java
// Pseudo code
// using the java syntax highlighting

class Animal {

    String species;

    void run() {

        if(species == "horse") {
            print("gallop really fast");
        } else if (species.startsWith("l")) {
            // these are lions
            print("roar and run");
        } else if (numberOfLegs > 2) {
            print("run really fast – because I've more legs");
        } else if (hasWings) {
            print("why run when you can fly");
        } else if (species == "cheetah") {
            print("cheetahs cheat and run really fast");
        } else if (species == "fish") {
            print("I'm a fish dude..");
        } else if (species == "snek") {
            print("Slithery slithery – I can't run, only crawl");
        } else if (species == "peacock") {
            print("I'm peacock, so I will run with pride");
        }
        // 100 species ..

    }
}
/*

```

```

        else if(species == "...") {}
        else if(species == "...") {}
        else if(species == "...") {}
        else if(species == "...") {}
        else if(species == "...") {}
        else if(species == "...") {}
        else if(species == "...") {}
        ...
    */

```

```

        // horses gallop
        // cheetas run
        // snakes slither
        // birds & rabbits hop
    }
}

```

```

class AnimalTester {
    void testHorseRun() {
        Animal myLittlePony = new Animal("horse");
        myLittlePony.run();
        // assert that this prints "gallop really fast"
        // based on the actual value, I can raise an assertion error if it
        fails
    }

    void testSnekRun() {
        Animal snek = new Animal("cobra");
        snek.run();
    }
}
...

```

🐞 Problems with the above code?

? Readable

Yes. Right now I can definitely read this code!

But

1. too many if-else cases, so reading will be slow
2. cases can interfere with each other, so I will have to read and understand the entire code very carefully
3. chances of mis-understanding certain cases is very high

? Testable

Yes. I can write test cases

1. different species are tightly coupled
  - making changes to one can end up effecting the others' behaviour
2. this is a nightmare for the QA team
  - now you've to test all components together

? Extensible

Can I add a new species? Yep – just add a new else-if condition

This has an issue – but we'll see this later

## ? Maintainable

Multiple developers working on the codebase – they can all work together – nice happy family

When all of them have made their own changes to `Animal.run``, when they commit & push – merge conflicts – difficult for devs to collaborate

## 🔧 How to fix this?

### =====

## ★ Single Responsibility Principle

### =====

- Every function/class/module/unit-of-code should have a single, well-defined responsibility
- Any unit-of-code should have a single reason to change
- If there is some code which serves multiple responsibilities
  - split it into multiple pieces, each with a single responsibility

```
```java
```

```
// interfaces vs abstract classes  
// use interfaces most of the time  
// when you need state (attrs) – use abstract class
```

```
abstract class Animal {  
    String species;  
    String name;  
    // ...  
  
    abstract void run(); // animals can run, but right now, we don't know how  
}  
  
class Reptile extends Animal {  
    void run() {  
        print("No legs, no can't run");  
    }  
  
    void crawl() {}  
    void spitPoison() {}  
}  
  
class Mammal extends Animal {  
    void run() {  
        print("run using 4 legs")  
    }  
}
```

```

class Bird extends Animal {
    void run() {
        print("why run when you can fly")
    }
}
...

```

– Readable

1. There are way too many classes now!

- this is not really a problem
- at any given time, you will not have to read all the classes
- as a dev you will be working with 1 species
- yes, there are many classes, but each class is tiny and very easy to read
- still, if you don't like multiple classes, you can use metaprogramming to fix that
  - pre-processors
  - reflection
  - metaclasses
  - template programming
  - decorators

This code is highly readable!

– Testable

If I make a change to the `Bird.run()` method, will it effect the testcases for `Mammal.run()` – no!

Classes are now de-coupled!

– Maintainable

If there are multiple devs, each working on a diff species – will we have a lot of merge conflicts?

Significant reduction!

## 🧠 Design a Bird

=====

```

```java

```

```

// the same kinda issues will arise
// there many different types of birds, which have different behavior
// let's focus on ONLY extensibility

```

```

class Bird extends Animal {
    // String species; // inherited from the parent class Animal

    void fly() {
        ... // what to write here?
    }
}

```

...

🐦 Different birds fly in different ways

```java

```
[library] GnuZooLibrary {  
    // .com .dll .class .jar .o ... compiled libraries  
    // you may NOT have access to the source code  
    // even if you do have access to the source code, you might not be allowed  
    to change it
```

```
    // imagine that the library author has written "bad code"  
    abstract class Animal { ... }
```

```
    class Bird extends Animal {
```

```
        void fly() {  
            if(species == "Sparrow") {  
                print("fly low")  
            } else if (species == "Eagle") {  
                print("glide high")  
            } else if (species == "Peacock") {  
                print("only pe-hens (female) can fly, the male peacocks can't")  
            }  
        }  
    }
```

```
    }
```

```
}
```

```
[executable] MyAwesomeZooGame {
```

```
    import GnuZooLibrary.*;
```

```
    // what do you do here to add a new species?  
    // you CAN'T  
    /*  
        else if(species == "penguin") {}  
    */
```

```
    class Game {  
        static void main(String args[]) {  
            Bird b = new Bird("peacock");  
            b.fly();  
        }  
    }
```

```
}
```

...

🐛 Problems with the above code?



- Readable
- Testable
- Maintainable

- Extensible – FOCUS!

As the user of this library, can you add a new species easily?

No – the user of the library is now handicapped because the library was poorly designed

Zerodha – Kite API – you can write custom algorithmic trading bots

All large companies – Google / Slack / Discord / Microsoft .. they provide APIs

We need to ensure that other devs who are using our library can still extend our library without changing our source code!

🔧 How to fix this?

=====

## ★ Open-Close Principle

=====

- Your code should be closed for modification, yet still, open for extension!
  1. people who don't have permission to modify your code (users of your library) should still be able to extend the library's functionality
  2. people even in your team who have write access to your code should not have to modify your code to add new functionality

? Why should we prevent our own devs from modifying existing code?  
Because modification sucks!

? Why is modification bad?

- Dev – write the code on local system – test, commit, & push
- PR review – ask you to make changes – ... (iterations) .. merged
- QA team – write extensive tests, integration, end-to-end testing
- Deployment
  - + Staging servers – make sure that the rest of the code doesn't break
    - end-to-end testing
  - + A/B testing
    - \* only 2% of the users get to see this new change
      - monitor the performance, bugs, errors, feedback, metric
      - if we see any issues – rollback & redo
    - \* finally deploy it to all the 2B+ users

```java

```
[library] GnuZooLibrary {  
    // .com .dll .class .jar .o ... compiled libraries  
    // you may NOT have access to the source code  
    // even if you do have access to the source code, you might not be allowed  
    to change it
```

```
    // imagine that the library author has written "bad code"  
    abstract class Animal { ... }
```

```
    abstract class Bird extends Animal {  
        abstract void fly();  
    }
```

```
    class Sparrow extends Bird { void fly() { print("fly low") } }  
    class Eagle extends Bird { void fly() { print("glide high") } }  
    // ...  
}
```

```
[executable] MyAwesomeZooGame {
```

```
    import GnuZooLibrary.*;
```

```
    // what do you do here to add a new species?  
    class Penguin extends Bird { void fly() { print("I don't fly!") } }
```

```
    class Game {  
        static void main(String args[]) {  
            Sparrow s = new Sparrow();  
            s.fly();
```

```
            Penguin p = new Penguin();  
            p.fly(); // this will print "I don't fly!"  
        }  
    }
```

```
}
```

```
...
```

#### – Extension

Now even though we don't have modification access to the source code, we can still add new functionality!

Google pays you 60LPA as SDE-3 they pay you this money because you can anticipate these issues and pre-emptively design for them

? Didn't we have the exact same solution (remove if-else and use inheritance) for the Single Responsibility Principle as well?

? Does this mean that the Open/Closed Principle == Single Responsibility Principle?

No – the solution was same, but the intent and effect was different

🔗 All the SOLID principles are tightly linked to each other

---

🐔 Can all birds fly?

=====

```
```java
```

```
abstract class Animal { String species; }
```

```
abstract class Bird extends Animal { abstract void fly(); }
```

```
class Sparrow extends Bird { void fly() { print("fly low") } }
```

```
class Eagle extends Bird { void fly() { print("glide high") } }
```

```
class Kiwi extends Bird {  
    void fly() {  
        !?  
    }  
}
```

```
```
```

Ostrich, Penguin, Kiwi, Emu, Dodo .. are birds that can't fly!

>

> ? How do we solve this?

>

- > • Throw exception with a proper message
- > • Don't implement the `fly()` method
- > • Return `null`
- > • Redesign the system

>

🏃 Run away from the problem – Don't implement the `void fly()`

```
```java
```

```
abstract class Animal { String species; }
```

```
abstract class Bird extends Animal { abstract void fly(); }
```

```
// abstract class - incomplete class - canNOT be instantiated  
// we told the compiler, that Birds can fly, but we don't really know how  
// void fly() was marked abstract
```

```
class Kiwi extends Bird {  
    // simply don't implement void fly()  
}
```

```
...
```

🤖 Compiler won't allow it!

Any class that inherits from `abstract class Bird` must

- either implement the `void fly()`
- or must itself also be marked as abstract

```
```py
```

```
from abc import ABC, abstractmethod
```

```
class Animal(ABC):  
    species: str
```

```
class Bird(Animal, ABC):  
    @abstractmethod  
    def fly(self):  
        ...
```

```
class Kiwi(Bird):  
    ...  
    # we MUST implement void fly, or we will get an exception
```

```
...
```

C++ – pure virtual functions

-----

- Composition over Inheritance
- Interfaces vs Abstract classes
- What is reflection
- Difference b/w Design Patterns & SOLID Principles
- Please explain what "state" in a class means

## Low Level Design

-----

- design at the code level
- topics
  1. Object oriented programming
  2. SOLID Principles
  3. Design Patterns
    - refactoring.guru
    - Factory
    - Builder

- you will NEVER use this in Python
- while learning these topics, it is important to learn from someone who knows what they're doing
- Singleton
- Observer
- Pub/Sub
- Proxy
- Template
- Strategy
- Chain of Responsibility
- Adapter
- Repository
- which design patterns are most popular
- how do the design patterns differ for languages
- 4. Database Schema Design
- 5. Machine Coding rounds
  - take-home assignments, in which you've to make working code
- 6. REST API design
- 7. ER-diagram/class-diagram
- 8. Case studies
  - Design Snake-Ladder
  - Library Management System
  - Splitwise
- 9. how to integrate DSA into your design?

## Scaler

- 92% of our students get placed
    - 8%? most of them do NOT want to get placed
    - CTOs who are students in our batches
  - median package of 21.6 lakhs
    - google average vs median
    - this is 3x better than any IIT
  - Amazon has more Scaler grads than all IITS/NITS combined!
- 
- What job should I target?
  - What salary should I target?
  - At what company/at what position/what is the expected salary
  - I'm from QA background - I wanna switch to dev role - how should I go forward
  - I've 12 years of experience in service based company - how should I switch to prod based
  - I dropped 4 years of UPSC prep - what should I do?
  - I want to talk to someone who went from TCS to Google

⚠ Throw an exception

```
```java
```

```
class Kiwi extends Bird {
    void fly() {
```

```

        throw new FlightlessBirdException("Kiwis don't fly");
    }
}

...

```

🐛 This violates expectations!

```

```java

```

```

abstract class Animal { String species; }

abstract class Bird extends Animal { abstract void fly(); }

class Sparrow extends Bird { void fly() { print("fly low") } }
class Eagle extends Bird { void fly() { print("glide high") } }

class Main {
    Bird getBirdObjectFromUserChoice() {
        // ask user for the choice of species
        // return a bird object of that species
        // return new Sparrow();
        // Bird b = new Sparrow() // allowed - Runtime Polymorphism
    }

    static void main() {
        Bird b = new getBirdObjectFromUserChoice();
        b.fly();
    }
}

// EXTENSION - diff dev
class Kiwi extends Bird {
    void fly() {
        throw new FlightlessBirdException("Kiwis don't fly");
    }
}

...

```



Before extension

Code works perfectly fine – dev is happy, QA is happy, user is happy



After extension

1. did we change existing code? No
2. so should existing code break? Obviously – it was working fine before, and I did not change it, ergo – it should work fine now
3. But in reality the `void main()` now breaks with a runtime exception!

=====

★ Liskov's Substitution Principle

=====

- Math definition: any parent class `class P` object should be replaceable with any child class `class C extends P` object.
- Human meaning: don't violate expectations!

🎨 Re-design the system

```
```java
```

```
// we know already that not all birds can fly  
// the fly() method should simply NOT be there in the Bird class
```

```
abstract class Bird extends Animal {  
    abstract void eat(); // all birds eat  
    abstract void speak(); // all birds speak  
  
    // NOT all birds fly  
    // abstract void fly() - this should NOT be here  
}
```

```
interface ICanFly { // IFunctionality  
    void fly();  
}
```

```
class Sparrow extends Bird implements ICanFly {  
    void eat() { ... }  
    void speak() { ... }  
    void fly() { print("fly low") }  
}
```

```
class Eagle extends Bird implements ICanFly {  
    void eat() { ... }  
    void speak() { ... }  
    void fly() { print("glide high") }  
}
```

```
class Kiwi extends Bird {  
    // it does NOT implement the ICanFly interface  
    void eat() { ... }  
    void speak() { ... }  
    // so if we don't provide void fly() the compiler won't complain  
}
```

```
class Main {  
    ICanFly getFlyingObjectFromUserChoice() {  
        // ask user for the choice of species  
        // return a bird object of that species  
        // return new Sparrow();  
        // Bird b = new Sparrow() // allowed - Runtime Polymorphism  
    }  
}
```

```
static void main() {
```

```

        ICanFly ufo = new getFlyingObjectFromUserChoice();
        ufo.fly();
    }
}

...

```

---

6.54 – small break of 10 mins – 7.05  
 Mihil – will talk to you for a little while :)

---

## DDoS

- distributed denial of service
- 6 Million requests per minute!
  - 100,000 requests per second!
- soon announce a masterclass – how to handle a DDoS attack

## ➔ What else can fly?

=====

```
```java
```

```

abstract class Animal {}
abstract class Bird extends Animal {
    abstract void eat();
}

```

```

interface ICanFly() {
    void fly();

```

```
    void spreadWings();
```

```
    void kickFromGround();
```

```
    void flapWings();
```

```

    // 1. spread wings
    // 2. small kick from their tiny legs – launch
    // 3. flap wings
}

```

```

class Shaktiman implements ICanFly {
    void fly() {
        print("spin really fast and make a weird sound")
    }

```

```

    void flapWings() {
        print("Sorry Shaktiman!")
    }
}

```



Aeroplanes, insects, Patang(kite), Shaktiman, Bhai in Car on footpath, Mummy's chappal, Villians in South Indian movies ... are things that can fly

```
```java
class User {
    Integer getAgeInYears() { return Time.now - this.dateOfBirth }
}

class Organisation extends User {
    Integer getAgeInYears() // invalid here!
}
```
```

>

> ? Should these additional methods be part of the ICanFly interface?

>

> • Yes, obviously. All things methods are related to flying

> • Nope. [send your reason in the chat]

>

Yes. because all of these are related to flying?

## =====

### ★ Interface Segregation Principle

## =====

- Keep your interfaces minimal
- No code (the clients of your code) should be forced to implement a method that it does not need.

How will you fix `ICanFly`?

Split it into multiple interfaces `ICanFly` `IHasWings`

- 🔗 Isn't this just the Single Responsibility Principle applied to interfaces!
- 🔗 All SOLID principles are linked!

## Rules vs Guidelines

### =====

- Rules: don't murder
  - + Must follow.

- + Enforced by some authority
- + programming: compiler
- + compiler ensures that a class implements the abstract methods of the parent class
- Guidelines: don't look directly into the sun
  - + best practices
  - + good to follow, but not really enforced
  - + It is important to know exactly WHEN and WHY to obey or disobey them!

## SOLID Principles – guidelines

- Hackathon (3 hour) – you care about getting shit done ASAP! Cutting corners is okay – you're just developing a proof-of-concept
- Startup (CTO) – you care about the business outcomes more than the code quality – move fast, fail fast, time-to-market, ...

Now that we've the necessary characters, let's design some structures

## Design a Cage

=====

```
```java
```

```
/*
```

- high-level code
  - abstractions / concepts without the implementation details
  - interfaces / abstract classes
- low-level code
  - implementation details
  - implement all the different methods and you know exactly how things are working
  - classes that can be instantiated

```
*/
```

```
abstract class Animal {...} // high-level
abstract class Bird extends Animal {...} // high-level
class Sparrow extends Bird implement ICanFly {...} // low-level
class Eagle extends Bird implement ICanFly {...} // low-level
class Penguin extends Bird {...} // low-level
class Cheetah extends Animal {...} // low-level
```

```
interface IBowl {} // for feeding animals // high-level
class MeatBowl implements IBowl {} // low-level
class FruitBowl implements IBowl {} // low-level
class GrainBowl implements IBowl {} // low-level
```

```
interface IDoor {} // for containing animals // high-level
class IronDoor implements IDoor {} // low-level
class WoodenDoor implements IDoor {} // low-level
class AdamantiumDoor implements IDoor {} // wolverine // low-level
```

```
class Cage1 { // high-level
    // this is a controller class
    // it does not do things by itself
    // it delegates the tasks to its dependencies
```

```
    // this is a cage for containing small birds
```

```
    GrainBowl bowl = new GrainBowl(...);
    WoodenDoor door = new WoodenDoor(...);
```

```
    List<Bird> birds = new ArrayList<>(
        new Sparrow(...),
        new Eagle(...)
    );
```

```
    public Cage1() {}
```

```
    void feedAllOccupants() {
        for(Bird b: this.birds) {
            this.bowl.feed(b) // delegating the task to our Bowl
        }
    }
```

```
    void resistAttack(Attack attack) {
        this.door.resistAttack(attack) // delegate the task
    }
}
```

```
class Cage2 { // high-level
    // this is a cage for containing big cats
```

```
    MeatBowl bowl = new MeatBowl(...);
    IronDoor door = new IronDoor(...);
```

```
    List<Animal> kitties = new ArrayList<>(
        new Cheetah(...),
        new Leopard(...)
    );
```

```
    public Cage2() {}
```

```
    // for Wolverine I would have to write yet another class
```

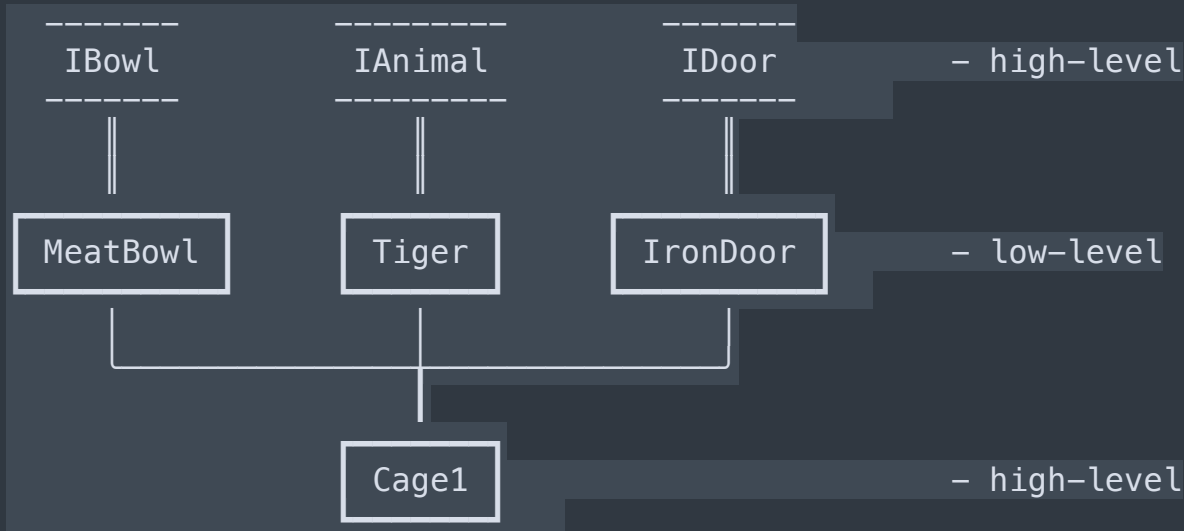
```
class Main {
    static void main() {
        Cage1 cageForBirds = new Cage1();
        Cage2 cageForBigPurrMachines = new Cage2();
    }
}
```

```
...
```

🐛 What is wrong with this code?

- lot of code repetition
- poor code reuse
- if I have hundreds of cages in the zoo, every cage will be slightly different
  - I need to have 100 classes in my code

...



...

- high level code `Cage1` depends on low-level code `MeatBowl`, `Tiger`, ...

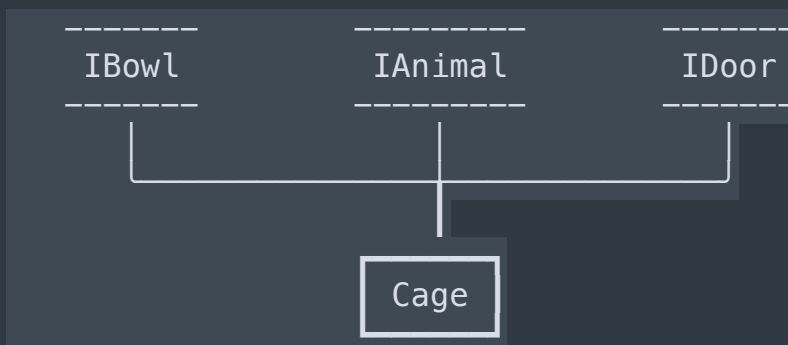
=====

## ★ Dependency Inversion Principle - what to do

=====

- High-level modules should ONLY depend on high-level abstractions
- they shouldn't depend on low level implementation details

...



...

But how?



- how do achieve inversion

- Instead of creating your dependencies yourself, you let your client (whoever is calling you) "inject" those dependencies

```
```java
```

```
abstract class Animal {...} // high-level
abstract class Bird extends Animal {...} // high-level
class Sparrow extends Bird implement ICanFly {...} // low-level
class Eagle extends Bird implement ICanFly {...} // low-level
class Penguin extends Bird {...} // low-level
class Cheetah extends Animal {...} // low-level
```

```
interface IBowl {} // for feeding animals // high-level
class MeatBowl implements IBowl {} // low-level
class FruitBowl implements IBowl {} // low-level
class GrainBowl implements IBowl {} // low-level
```

```
interface IDoor {} // for containing animals // high-level
class IronDoor implements IDoor {} // low-level
class WoodenDoor implements IDoor {} // low-level
class AdamantiumDoor implements IDoor {} // wolverine // low-level
```

```
class Cage {
    // is a "generic" Cage
```

```
IBowl bowl; // I don't know what kind of bowl it will be
IDoor door;
List<Animal> occupants;
```

```
// inject the dependencies via the constructor
//
//
public Cage(IBowl bowl, IDoor door, List<Animal> occupants) {
    this.bowl = bowl;
    this.door = door;
    this.occupants = Arrays.asList(occupants);
}
```

```
}
```

```
class Main {  
    static void main() {  
  
        Cage birdCage = new Cage(  
            new GrainBowl(...),  
            new WoodenDoor(...),  
            Arrays.asList(  
                new Sparrow(...),  
                new Eagle(...),  
            )  
        );  
  
        Cage kittyCage = new Cage(  
            new MeatBowl(...),  
            new IronDoor(...),  
            Arrays.asList(  
                new Cheetah(...),  
                new Leopard(...),  
            )  
        );  
  
        Cage xMenCage = new Cage(  
            new MeatBowl(...),  
            new AdamantiumDoor(...),  
            Arrays.asList(  
                new Wolverine(...),  
                new Beast(...),  
            )  
        );  
  
    }  
}
```

```
...
```

## Enterprise Code

=====

When you go to companies like Google, you will see very very very complex code

```
```java  
class StripePaymentGatewayFactory implements IPaymentGatewayFactory {  
    IFileLogger logger = SimpleFileLogger.getInstance();  
  
    IPaymentGateway getDefaultStripeGateway() {  
        ...  
    }  
}
```

- Junior devs that just joined Google and don't know LLD/SOLID principles
- they're miserable - because everything seems so tough
  - they're unable to contribute - everytime they submit a PR, their leads tell them to re-write
  - smallest of features take forever to implement

Junior Dev + LLD knowledge

- you won't even have to read the code
  - because the file-name/function-name tells you exactly what it does
- and whatever code you write - it is designed keeping future changes in mind

## Quick Recap

=====


SOLID principles

- Single Responsibility: each unit-of-code should have 1 reason to change
- Open/Close: code should be closed for modification, yet, open to extension
- Liskov's Substitution: any parent class object should be replaceable by a child class object
- Interface Segregation: keep your interfaces minimal
- Dependency Inversion: high-level code should depend only on high-level abstractions
  - Dependency Injection: way to achieve the inversion principle
    - let your clients create & inject the dependencies into you

=====

## Bonus Content

=====

- >
- > We all need people who will give us feedback.
- > That's how we improve.  Bill Gates
- >

=====

## ★ Interview Questions

=====

- > ? Which of the following is an example of breaking
- > Dependency Inversion Principle?
- >
- > A) A high-level module that depends on a low-level module
- > through an interface
- >

- > B) A high-level module that depends on a low-level module directly
- >
- > C) A low-level module that depends on a high-level module through an interface
- >
- > D) A low-level module that depends on a high-level module directly
- >

> ? What is the main goal of the Interface Segregation Principle?

- >
- > A) To ensure that a class only needs to implement methods that are actually required by its client
- >
- > B) To ensure that a class can be reused without any issues
- >
- > C) To ensure that a class can be extended without modifying its source code
- >
- > D) To ensure that a class can be tested without any issues

>

> ? Which of the following is an example of breaking Liskov Substitution Principle?

- >
- > A) A subclass that overrides a method of its superclass and changes its signature
- >
- > B) A subclass that adds new methods
- >
- > C) A subclass that can be used in place of its superclass without any issues
- >
- > D) A subclass that can be reused without any issues
- >

> ? How can we achieve the Interface Segregation Principle in our classes?

- >
- > A) By creating multiple interfaces for different groups of clients
- > B) By creating one large interface for all clients
- > C) By creating one small interface for all clients
- > D) By creating one interface for each class

> ? Which SOLID principle states that a subclass should be able to replace its superclass without altering the correctness of the program?



- > A) Single Responsibility Principle
- > B) Open-Close Principle
- > C) Liskov Substitution Principle
- > D) Interface Segregation Principle
- >

>

> ? How can we achieve the Open-Close Principle in our classes?

>

- > A) By using inheritance
- > B) By using composition
- > C) By using polymorphism
- > D) All of the above
- >

=====

★ How do we retain knowledge

=====

>

> ? Do you ever feel like you know something but are unable to recall it?

>

- > • Yes, happens all the time!
- >
- > • No. I'm a memory Jedi!
- >

-----  
🧩 Assignment

-----  
<https://github.com/kshitijmishra23/low-level-design-concepts/tree/master/src/oops/SOLID/>

# ===== That's all, folks! =====