## 👨‍💻 About the Instructor

Pragy
Senior Software Engineer + Instructor @ Scaler

https://linktr.ee/agarwal.pragy

## ❓ 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?
  7.30pm – 10.30pm (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

## 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 continous 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 dev's time is spent writing new code!

⏱️ where does the rest of the time go!?

- Code related stuff (but not coding)
  - looking up documentation
  - reading existing codebase
  - debugging
  - writing documentation
  - compiling
  - deploying
  - reviewing PRs
  - google/stackoverflow/chatGPT
- Managerial stuff
  - SCRUM
  - KTs
  - HR stuff / team events
- Fun part
  - chai/coffee
  - playing table tennis
  - loitering around / daydreaming
  - scroll instagram

I want to have high productivity during my "work" time - so that I don't have to spend a lot of in it

**✅ Goals**
=========

We'd like to make our code

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

What you can't measure, you can't improve!

#### 👴🏻

====================
**💎 SOLID Principles**
====================

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

Segregation of responsibility / single responsibility
Inversion of Control / Interface Segregation / Isolation / Independent
DRY / Dependency Injection / Dependency Inversion

**💭 Context**
==========

- "toyish" example - captures the nuances, without unnecessary complexity
- Build a Zoo Game 🦊
- characters
- structures

Programming language:
- not going to follow any particular programming language
- pseudo-code
- looks like modern java, but will not compile
- something that you can read & understand

- the concepts that we study will apply to any modern programming language that supports OOP
    - python
    - java / kotlin
    - javascript/typescript
    - ruby
    - php
    - c++
    - c#

---

## 🎨 Design a Character

Visitors - buy tickets, loiter around, see animals, eat, make noise, throw garbage..
Animals - eat food, eat the visitors, hide in their cage, get aggressive, ..
Staff - clean, feed the animals, do staffy stuff

all of these are "concepts" in our minds
we need a way to model these concepts into code
OOP - use a class!

```java
// THE CODE BELOW IS SHITTY CODE - ITS HORRIBLE
// DO NOT WRITE CODE LIKE THIS!
class ZooActor {
   // properties [attributes]
      // visitor
         String visitorName;
         String visitorAddress;
         Integer visitorAge;
         String visitorPhone;
         Integer visitorTicketID;
      // staff
         String staffName;
         String staffAddress;
         Integer staffAge;
         String staffPhone;
         Integer staffDepartment;
         Integer staffSalary;
      // animal
         String animalName;
         Integer animalWeight;
         Integer animalAge;
         String animalSpecies;
         String animalIsCarnivorous;

   // methods [behavior]
      // visitor
         void roam();
         void lookAtAnimals();
         void eat();
         void poop();
         Ticket bookTicket();
      // staff
         void feedAnimals();
         void cleanPremises();
         void eat();
         void poop();
         void checkIn();
      // animal
         void roam();
         void lookAtVisitors();
         void eat();
         void poop();
         void eatTheVisitors();
}


class ZooActorTester {
   void testAnimalEating() {
      ZooActor animal = new ZooActor(...);
      animal.eat();
      // ensure that the correct behavior is displayed
   }
}
```

- name collisions
   "name" exists for both staff, and visitor, and animal
   this is a non-issue, because I can simply rename my variables

Major Issues

🐞 Problems with the above code?

❓ Readable
      This code is readable right now
      But, as the number of actors increase (staff categories / VIP visitors / add animal species) this code will quickly become unreadable!

❓ Testable
      Yes, I can test this code.

      Because all the state is "shared" across all actor types, changing the behavior/code/logic for one can incorrectly effect the code/logic for other actors
      Tests will break all the time
      Very hard to write comprehensive tests & keep them up to date

❓ Extensible
   We'll come back to this

❓ Maintainable

```
      If the dev team is large, and multiple devs are working on different features
         - someone is working on visitors
         - someone is working on animals
      they will end up modifying the same class & same file
      when they try to submit the changes, they will incur - merge conflicts!
```

🤔 What is the main reason for all these issues?

```
      one single class is doing tooo many things!
      too many responsibilities!
```

⚒️ How to fix this?

==================================
⭐ **Single Responsibility Principle**
==================================

- Any function/class/module (unit-of-code) should have a single, well-defined responsibility
      - single, not multiple
      - well-defined, not vague
         - specify what this class will do, and what this class will NOT do

- (a.k.a) any piece of code should have only 1 reason to change

- if you see a piece of code that violates the SRP, then you should break it down into multiple pieces
   - we can use OOP tool "inhertance" to split this code

```java
class ZooActor {
   String name;
   Integer age;
   Integer weight;

   void eat();
   void poop();
}

class Animal extends ZooActor {
   Integer weight;
   String species;
   String isCarnivorous;

   void roam();
   void lookAtVisitors();
   void eatTheVisitors();
}
class Visitor extends ZooActor {
   String address;
   String phone;
   Integer ticketID;

   void roam();
   void lookAtAnimals();
   Ticket bookTicket();
}
class Staff extends ZooActor {
   String address;
```

```
    String phone;
    Integer department;
    Integer salary;

    void feedAnimals();
    void cleanPremises();
    void checkIn();
}
```

It's just not enough to make changes - we need to ensure that the changes are infact better!
Did we improve on our metrics?

❓ Readable
Aren't there wayy too many classes now?
- Yes, there are.
- But that does NOT make the code less readable
  - at any given time, any developer will only be working with 1 or a few classes/files
  - each of those classes/files is small, and easily readable

```
      The code is more readable now!
```

❓ Testable
Can a change made inside Visitor class effect (by mistake) the behavior of Animal class?
No!

```
      Code is more testable!
      Easier to maintain the testcases
      Easier to write testcases (because less complexity in each class)
```

❓ Extensible
We'll come back to this later

❓ Maintainable
If diff devs are working on different features (visitor / animal)
will they have merge conflicts?
significantly reduced!

Is the code perfect now?
No. (no such thing as perfect)
But it is better - it's a step in the right direction

---

Avoid the urge for spoon feeding
When you have a question
- note that question down somewhere (so that you don't forget it)
- try to answer it yourself
  - if your unable to
    - ask the mentor
  - if you're able to
    - ask the mentor for alternatives

---

🐦 **Design a Bird**

=================

```java
// Akash is building this
// Akash is already familiar with the single responsibility principle
// but he doesn't have enough time right now

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

   void fly() {
      // Akash needs to implement this function
   }

}
```

🕊️ different birds fly differently!

```java

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

   void fly() {
      if (species == "Sparrow")
         print("fly low, close to the ground")
      else if (species == "Pigeon")
         print("fly, and poop on people below while flying")
      else if (species == "Eagle")
         print("glide high above the clouds elegantly")
   }

}
```

🐞 Problems with the above code?

- Readable
- Testable
- Maintainable

- Extensible - FOCUS!
   Is this code extensible? Can I add a new type of bird to this code?
      Yes, easily!
      Just add another `else if` condition

      ```java
         else if (species == "Peacock")
            print("pe-hens(females) can fly, but males cannot")
      ```

❓ Do we always write all code ourselves from scratch?

No. We usually import code from external libraries

```java

[PublicZooLibary] // Akash wrote this library and published it on githuhb
{
```

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

        void fly() {
            if (species == "Sparrow")
                print("fly low, close to the ground")
            else if (species == "Pigeon")
                print("fly, and poop on people below while flying")
            else if (species == "Eagle")
                print("glide high above the clouds elegantly")
        }
    }
}

[MyCustomGame] { // Srihari is using the above library to build his own Zoo Game
    import PublicZooLibary.Bird;

    class SriZooGame {
        void main() {
            Bird tweety = new Bird(species="sparrow");

            tweety.fly();
        }
    }

    // Srihari now wants to add a new type of bird and make it fly
}
```

❓ Do we always have modification access?
    No!
    Most of the time we have the code (because open source) but we still cannot modify
    Sometimes, libraries aren't even shipped with code
        libraries are shipped as compiled extensions
            (.com .dll .so .jar ...)

❓ How can we add a new type of Bird?
    It becomes difficult to "extend" the code with new features if the original author never designed it for extension

🛠️ How to fix this?

========================
⭐ **Open/Close Principle**
========================

- Your code should be closed for modification, yet, it should remain open to extension!

> how is this even possible? cannot modify, but can still extend?

❔ Why is it bad to modify existing code?

Code lifecycle in a large company (Google)

~~~
   - dev writes code on their laptop. They test it (locally), they commit it,
submit a Pull Request (PR)
   - PR goes for review
      - seniors/other devs in the team review it, and suggest improvements
      - you go back and make the changes
      - (iterative process)
      - finally the PR gets approved & merged
   - Quality Assurance (QA) team
      - write unit tests
      - write integration tests
      - do manual testing
   - Deployment
       + Staging servers
            - monitor metrics
            - ensure that things are working fine
            - perform rigourous integration tests
       + Production servers
           * A/B test
               - deploy to only 5% of the userbase
               - monitor
                  - number of exceptions
                  - customer satisfaction & rating
                  - revenue
           * finally you can deploy to all users


```java
[PublicZooLibary] // Akash wrote this library and published it on githuhb
                  // Akash made sure to follow Open/Close principle
{

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

      abstract void fly();
   }

   class Sparrow extends Bird {
      void fly() { print("fly low, close to the ground") }
   }
   class Pigeon extends Bird {
      void fly() { print("fly, and poop on people below while flying") }
   }
   class Eagle extends Bird {
      void fly() { print("glide high above the clouds elegantly") }
   }
}

[MyCustomGame] { // Srihari is using the above library to build his own Zoo Game
   import PublicZooLibary.Bird;

   class SriZooGame {
      void main() {
         Bird tweety = new Bird(species="sparrow");

         tweety.fly();
      }
   }

   // Srihari now wants to add a new type of bird and make it fly
   class Peacock extends Bird {
      void fly() { print("pe-hens(females) can fly, but males cannot") }
    }
}
~~~

```

❓ Isn't this the same thing that we did for Single Responsibility as well?
      Yes, in both cases, we took a large class and broke it into smaller classes by using inheritance

❓ Does that mean that OCP == SRP?
      No. The solution was the same, but the intent and benefits were different!

🔗 All the SOLID principles are tightly linked together! If you adhere to one, you might get others for free!

❓ Didn't we modify the Bird class?
      No, I'm not saying that you should modify the original (bad) Bird class
      I'm saying that you should desing the Bird class in a way that it is "extensible without requiring modification" from day-1

-----------------------------------------------------------------------------

Salary of a (top-level) developer in tier-1 companies (Google/Amazon)
   - not US/UK offers
   - Indian offers - Banagalore / Hyderabad / Pune / Mumbai

Principle Engineer / Senior Architect (10+ yoe in a good company)
   - (base) upto 3 Cr Rs.

Why would a company pay this ridiculous amount to 1 developer?
   Because that developer has the ability to "predict" the future changes

Good engineers are able to anticipate future requirements and develop solutions today that do not need (major) changes for the future requirements.

-----------------------------------------------------------------------------

How can I become someone like that? Someone who can demand a salary of 3 Cr?
You need to upskill yourself

**Scaler Curriculum for Low Level Design**
======================================
   - basics & deep-dive into OOP
      - polymorphism / abstraction / encapsulation / generalization / inheritance / composition over inheritance / runtime polymorphism / abstract classes / multi-level & multiple inheritance / MRO / interfaces
   - SOLID principles
   - Design Patterns
      - Creational / behavioral / structural
      - master the top 10
      - be very careful! Most resources online are wrong!
         - builder pattern
            - how many of you know how to implement this in Python?
            - it's a design pattern that works around lnaguage limitation
               - java does not support optional params / named params / changing the order of params / default values
   - Class design - ER diagram, database schema
   - Requirements gathering
   - Case studies
      - Tic/Tac/Toe, Snake/Ladder, Chess
      - Library management
      - Parking Lot
```

    - Splitwise
    - ...

---

9.10 - 9.25 sharp (15 mins break)

---

## 🐓 Can all the birds fly?

```java
abstract class Bird extends Animal {
   // String species; inherited from the parent class
   Integer beakLength;

   abstract void fly();
}

class Sparrow extends Bird {
   void fly() { print("fly low, close to the ground") }
}
class Pigeon extends Bird {
   void fly() { print("fly, and poop on people below while flying") }
}
class Eagle extends Bird {
   void fly() { print("glide high above the clouds elegantly") }
}

class Kiwi extends Bird {
   // extending Bird, because Kiwi most certainly is a Bird!

   void fly() {
      // Kiwi cannot fly!

      // return null;
      throw new FlightlessBirdException("Kiwi's can't fly bro!");
   }
}
```

Are there some birds that cannot fly?
Yes!
   Ostritch, Penguin, Emu, Kiwi, Dodo, ...

>
> ❓ How do we solve this?
>
> - Throw exception with a proper message
> - Don't implement the `fly()` method
> - Return `null`
> - Redesign the system
>

🏃 Run aaway from the problem - Don't implement the `void fly` at all!

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

```java
class Kiwi extends Bird {
    // we have not implemented void fly!
}
```

🐞 Compiler Error!
Abstract class = incomeplete class
Abstract method = contract that says "either a child class should supply the imeplementation of this function, or the child child will itself be incomplete"

Either class Kiwi should provide an implementation of void fly, or class Kiwi should be abstract itself!

⚠ Throw a proper exception

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

class Sparrow extends Bird {
    void fly() { print("fly low, close to the ground") }
}
class Pigeon extends Bird {
    void fly() { print("fly, and poop on people below while flying") }
}

class ZooGame {

    Bird getBirdTypeFromUserSelection() {
        // shows a nice GUI to the user
        // it shows various bird species - abilities, weakness, food type, ..
        // user selects one species
        // function creates an object of that species
        // and returns it

        // possible returned types
        // Sparrow / Pigeon
    }

    void main() {
        Bird b = getBirdTypeFromUserSelection();
        b.fly();
    }
}
```

🐞 Violates Expectations

```java
// an intern joins the company, and the write the following code

class Kiwi extends Bird {
    void fly() {
        throw new FlightlessBirdException("Kiwi's can't fly!");
    }
}
```

```
class ZooGame {

    Bird getBirdTypeFromUserSelection() {
        // shows a nice GUI to the user
        // it shows various bird species - abilities, weakness, food type, ..
        // user selects one species
        // function creates an object of that species
        // and returns it

        // possible returned types
        // Sparrow / Pigeon / Kiwi
    }

// note:
// Bird getBirdTypeFromUserSelection() can now also return Kiwis
// we will get an exception in the main class

    void main() {
        Bird b = getBirdTypeFromUserSelection();
        b.fly(); // if this happens to be a Kiwi object, we will get an exception here
    }
}
```

✅ Before extension
   - code works
   - client happy
   - dev happy

❌ After extension
   - did the intern touch existing code?
      No
   - does the original dev of the code know about the addition that the intern made?
      Not necessarily
   - was the code working before the intern added their stuff?
      Yes
   - does the code work now?
      No
   - does the intern's code work?
      Yes
   - does the old code work?
      No.

==================================
**⭐ Liskov's Substitution Principle**
==================================

- (bording definition) Any object of a `class Parent` should be replacable with any object of a `class Child extends Parent` without any issues

- (intuition)
   - parents set the expectations
   - children are supposed to meet those expectations
   - if a child doesn't meet the parent's expectations - chappal

A child should not violate their parent's expectations

🎨 Redesign the system!

Use an interface (python: continue using abstract base class, c++: continue using pure virtual methods)

```java
abstract class Bird extends Animal {
   int beakLength;
   int weight;

   abstract void speak();

   // but since I know that NOT all birds can fly
   // I will not put a abstract void fly() here
}

interface ICanFly {
   void fly();
}

class Sparrow extends Bird implements ICanFly {
   void speak() {print("chirp chirp")}

   void fly() {print("flap flap")}
}

class Kiwi extends Bird {
   // no need to implement ICanFly

   void speak() { ... }

   // NOTE: no void fly here
   //       and no compiler error
}



class ZooGame {

   Bird getBirdTypeFromUserSelection() {
      // shows a nice GUI to the user
      // it shows various bird species - abilities, weakness, food type, ..
      // user selects one species
      // function creates an object of that species
      // and returns it

      // possible returned types
      // Sparrow / Pigeon / Kiwi
   }

// note:
// Bird getBirdTypeFromUserSelection() can now also return Kiwis
// we will get an exception in the main class

   void main() {
      Bird b = getBirdTypeFromUserSelection();
      if(b instanceof ICanFly) {
         ICanFly flyingB = (ICanFly) b;
         flyingB.fly();
      }
      b.speak();
   }
}
```

```py
from abc import ABC, abstractmethod

class Bird(ABC):
   @abstractmethod
   def speak(self):
      raise NotImplementedError()

   ...


class FlightMixin(ABC):
   @abstractmethod
   def fly(self):
      raise NotImplementedError()


class Sparrow(Bird, FlightMixin):
   def speak(self):
      print('chirp chirp')

   def fly(self):
      print('flap flap')

class Kiwi(Bird):
   def speak(self):
      print('chirp chirp')
```

Q: didn't we modify existing code for this change to happen?
   No - we expected the code to be written in a good manner from the very start

Q: aren't we violating the OCP?
   No - we;re just demanding that the dev anticipates future changes and codes today according to that

---

## ✈ What else can fly?

```java
abstract class Bird extends Animal {
   int beakLength;
   int weight;
}

interface ICanFly {
   void fly();

   // when a Bird is flying what steps does it take?
   void spreadWings();
```

```
    void flapWings();
    // spreadWings
    // flapVigourously
}

class Sparrow extends Bird implements ICanFly {
    void fly() {print("flap flap")}
}

class Kiwi extends Bird {
    // no need to implement ICanFly
    // NOTE: no void fly here
    //        and no compiler error
}

class Shaktiman implements ICanFly {
    void fly() {
        print("put finger up, rotate like crazy, fly!")
    }

    void spreadWings() {
        print("SORRY Shaktiman!")
    }
}
```

>
> ❓ 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]
>

==================================
## ⭐ Interface Segregation Principle
==================================

- Keep your interfaces thin / minimal

- Your clients should not be forced to implement methods that they do not need

❓ Isn't this similar to LSP? Isn't this just SRP applied to interfaces?

🔗 Yes
   But it also applies to other interfaces - like REST APIs
   Zeta
      - class constructer had 300 parameters

How will you fix `ICanFly`?

```java
interface ICanFly {
    void fly();
}
```

~~~~
interface IHasWings {
   // when a Bird is flying what steps does it take?
   void spreadWings();
   void flapWings();
}
```


-----------------------------------------------------------------------------


🗑️ Design a Cage
================

```java

interface IBowl {                      // High Level Abstraction
   void feed(Animal animal);
}
class FruitBowl implements IBowl {     // Low Level Implementation Detail
   void feed(Animal animal) {
      // what fruits are safe for this animal
      // cats cannot eat grapes - they'll die
      // ensure enough quanity
      // don't want to give 2 grapes to an ostritch
      // add the proper enzymes for digestion
      // put in a nice and presentable manner
      // ensure that the animal has not eaten twice
   }
}
class MeatBowl implements IBowl {     // Low Level Implementation Detail
   void feed(Animal animal) { ... }
}
class GrainBowl implements IBowl {    // Low Level Implementation Detail
   void feed(Animal animal) { ... }
}

interface IDoor {                     // High Level Abstraction
   void resistAttack(Attack attack);
}
class WoodenDoor implements IDoor {   // Low Level Implementation Detail
   void resistAttack(Attack attack) {
      if(this.strength <= attack.power) {
         // attack mitigated
         this.strength -= attack.power
      } else {
         this.strength = 0;
         // make the animals escape
      }
   }
}
class IronDoor implements IDoor {    // Low Level Implementation Detail
   void resistAttack(Attack attack) { ... }
}
class NetDoor implements IDoor {     // Low Level Implementation Detail
   void resistAttack(Attack attack) { ... }
}


class Cage1 {                        // High Level - Controller code
   // for tigers
   // it can house animals
   // animals can try to get out, so it needs to restraint them and survive attacks
   // the animals need to be fed
~~~~

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

    List<Tiger> kitties = new ArrayList<>();

    public Cage1() {
        kitties.add(new Tiger("simba"));
        kitties.add(new Tiger("musafa"));
        kitties.add(new Tiger("sharekhan"));
    }

    void feed() {
        // delegate to the bowl
        for(Tiger t: this.kitties)
            this.bowl.feed(t);
    }

    void resistAttack(Attack attack) {
        // delegate to the door
        this.door.resistAttack(attack);
    }
}

class Cage2 {
    // for pigeons

    GrainBowl bowl = new GrainBowl();
    NetDoor door = new NetDoor();

    List<Pigeon> poopies = new ArrayList<>();

    public Cage1() {
        poopies.add(new Pigeon("piggy 1"));
        poopies.add(new Pigeon("piggy 2"));
        poopies.add(new Pigeon("piggy 2"));
    }

    void feed() {
        // delegate to the bowl
        for(Pigeon p: this.poopies)
            this.bowl.feed(p);
    }

    void resistAttack(Attack attack) {
        // delegate to the door
        this.door.resistAttack(attack);
    }
}

// ...


class ZooGame {
    void main() {
        Cage1 kittyCage = new Cage1();
        Cage2 poopyCage = new Cage2();
        // now if I wish to add another cage, I need to first create a new class
        // client (main function) has no control
    }
}
```

🐞 Violates DRY (Don't repeat yourself)

Lot of code repetition

# High Level vs Low Level code
   - High Level code: tells you what to do, but not how to do it
      - High level abstractions: interfaces / abstract classes
      - Controllers: managerial code (that doesn't do the task itself, just forwards the task to the correct dependency)
   - Low Level code: it tells you exactly how something is being done (the detailed steps)

```
 interface    abstract class     interface
  -------      ---------         -------
   IBowl          Animal          IDoor         high level abstractions
  -------      ---------         -------
     ‖              ‖                ‖
┌──────────┐    ┌───────┐     ┌──────────┐
│ MeatBowl │    │ Tiger │     │ IronDoor │      low level
└──────────┘    └───────┘     └──────────┘      implementation details
     │              │                │
     └──────────────┼────────────────┘
                    │
               ┌───────┐
               │ Cage1 │                        high level controller
               └───────┘
```

High level controller `Cage1` depends on low level implementation details `MeatBowl`, `Tiger`, `IronDoor`

=================================
⭐ **Dependency Inversion Principle**        **- what to do**
=================================

- Code should NEVER depend on low level implementation details
- Code should ONLY depend on high level abstractions

```
   -------       ---------        -------
    IBowl          Animal          IDoor        high level abstractions
   -------       ---------        -------
      │              │                │
      └──────────────┼────────────────┘
                     │
                 ┌──────┐
                 │ Cage │                       controller
                 └──────┘
```

But how?

======================

## Dependency Injection - how to achieve the principle

- don't create your dependencies yourself
- instead, let your client "inject" them (via the constructor/callback/method call)

```java

interface IDoor { ... }                            // High Level Abstraction
class IronDoor implements IDoor { ... }            // Low Level Implementation Detail
class WoodenDoor implements IDoor { ... }          // Low Level Implementation Detail
class AdamantiumDoor implements IDoor { ... }      // Low Level ...

interface IBowl { ... }                            // High Level Abstraction
class MeatBowl implements IBowl { ... }            // Low Level Implementation Detail
class GrainBowl implements IBowl { ... }           // Low Level Implementation Detail
class FruitBowl implements IBowl { ... }           // Low Level Implementation Detail

class Cage {
   // generic class - can hold anything

   // BAD: MeatBowl bowl = new MeatBowl();
   // GOOD:
   IBowl bowl;  // not specified the details (type of bowl)
                // not "created" the dependency (just declared it)

   IDoor door;

   List<Animal> animals;


   // inject the dependencies via the constructor
   //          vvvvvvvvvv  vvvvvvvvvv  vvvvvvvvvvvvvvvvvvvv
   public Cage(IBowl bowl, IDoor door, List<Animal> animals) {
      this.bowl = bowl;
      this.door = door;
      this.animals.addAll(animals);
   }
}

class ZooGame {
   void main() {
      Cage kittyCage = new Cage(
         new MeatBowl(),
         new IronDoor(),
         Arrays.asList(new Tiger("simba"), new Tiger("musafa"), ...)
      );


      Cage poopyCage = new Cage(
         new GrainBowl(),
         new WoodenDoor(),
         ...
      );

      // now if I wish to add another cage
      // I can just make another object

      // client is all powerful
   }
}

```

## Enterprise Code

The code is "too" complex - "enterprise code" / "overengineered code"

If you go to top companies - like Google
- 100,000 devs
- 10,000 projects
- developers leave all the time
- requirements change
- projects get shelved and then reopened
- funding gets cut
- projects las t for 10+ years

You "need" to overengineer - so that things work.

Dev 1 - bad at LLD - they join Google
- everything will so confusing
- very long class names
- needless patterns everywhere

Dev 2 - good at LLD - they join Google
- don't have to read the code
- the class name tells them exactly what the code will be!

## 🎁 Bonus Content

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

## 🧩 Assignment

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

## ⭐ 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
>

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