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?
8pm - 11pm (3 hours) [15 min break midway]

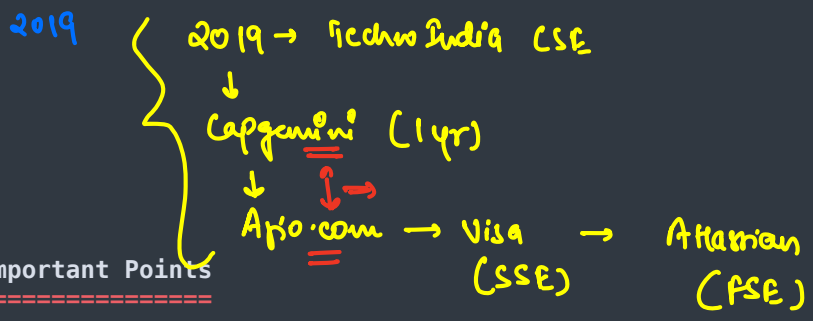
🎧 Audio/Video issues
Disable Ad Blockers & VPN. Check your internet. Rejoin the session.

? 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



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 fresh code!

~12%

Where does the rest of the time go?

- reading other people's code
- stackoverflow / researching / reading docs
- Knowledge Transfers (KT)
- Breaks - playing TT / snooker / chai & sutta
- Meetings

• Reading code, understanding requirement, task breakdown, estimation

Whatever stuff I get done - it is done FOREVER

Goals

We'd like to make our code

1. Readable
2. Extensible ⇒ easy to add new features
3. Maintainable ⇒ easy to maintain the system [keep the system running] KILO (keep the lights on)
4. Testable

log4J ⇒ version change

version = ++1.0.2

Robert C. Martin - Uncle Bob

SOLID Principles

- Single Responsibility Principle (SRP)
- Open/Closed Principle (OCP)
- Liskov's Substitution Principle (LSP)
- Interface Segregation Principle
- Dependency Inversion

Interface Segregation / Inversion of Control
Dependency Inversion / Dependency Injection

We will write pseudo-code (code that is not in any particular language)
Java

object Orientented Programming
C++, C#, any .Net language, Java, Python, Ruby, Javascript, Kotlin, Php..

Context

- Zoo Game 🦊
- [Modeling various animals]

Classes for all animals → eg. can change or inc.

(same shape & structure) ⇒ diff. flavours

Cake tin → blueprint for cake

Design an Animal

```java
// concepts  ->  class   (blueprint/idea/concept)

class Animal {

    // attributes [ properties]
    String color;
    String gender;
    String species;
    Integer age;
    Double weight;
    Boolean hasWings;
    Boolean canBreatheUnderwater;


    // behaviour [methods]

    void eat();

    void run();

    void swim();

    void attack();

}
```

⇒ Cat | Dog | Tiger | Elephant

```py
class Animal:
    color: str
    gender: str
    age: int
    weight: float

    def eat(self):
        ...

    def run(self):
        ...
```

🐠 Different Animals will behave in different manners

```java
class Animal {

    // attributes [ properties]
    String species;


    // behaviour [methods]

    void run() {
        // what should I do here?

        String horseSays = "neeeeiiiiiggggghhhhh.... I'm horse. I run fast";
```

```
      if(species == "Bird") {
         ...
      }

      if(species == "Cobra") {
         print("Hiss Hiss - I don't run. I ain't got no legs")
      } else if (getCategoryOf(species) == "Mammal") {
         // print("let's run")
         print(horseSays)
      } else if (numberOfLegs <= 2) {
         print("I can run but not that fast")
      } else if (numberOfLegs == 4) {
         print("Gallop really fast")
      } else {
         print("what do I do here?")
      }

   }
}


class AnimalTester {
   bool testReptileRun() {
      Animal snek = new Animal();
      snek.run();

      // make assertions that the above function call must print "Hiss Hiss - I don't run. I ain't
got no legs"
   }
}

```


 Problems with the above code?

If-else ladder is bad - instinct
WHY?

? Readable
Yes, it seems readable. I can totally read & understand it.
If I have lots of species, I need to look carefully at every single if-else condition to understand
what is really happening

? Testable
Yes, I can totally write testcases.

Changing the behavior of one species can affect the behavior of other species.
Testcases / code are tightly coupled

? Extensible
Seems extensible - we'll come back to this later

? Maintainable
10 devs - each dev is working with a different species
All of them are modifying the same function at the same time - Merge Conflicts!

Junior dev's perspective / Initial assessment - code seemed fine
Closer look / Senior dev's perspective - code is a giant mess

 How to fix this?

S O L I D

Single Responsibility Principle

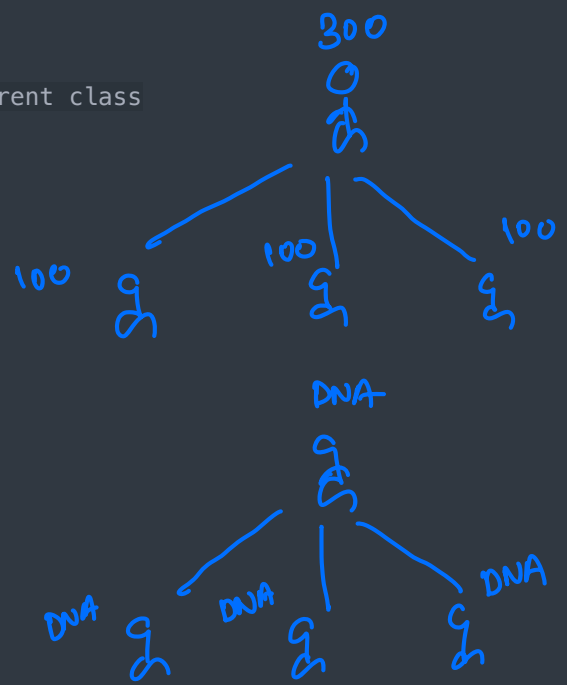
- Every function / class / module / unit-of-code should have a single, well-defined respnsibility

- Another way to say it - any unit-of-code should have exactly 1 reason to change

- If we find that some code is serving multiple purposes - break it down into smaller, individual pieces - each with it's own well defined responsibility

class
attribute
normal concrete methods
abstract methods
↓
dont have an implement

```java

// incomplete concepts - Abstractions

// Java - Abstract class / Interface
// Python - from abc import ABC   @abstractmethod
// C# - Abstract class / Interface
// C++ - pure virtual methods
// Typescript - Interfaces
// Swift - Protocols

abstract class Animal {
    String species;
    String color;
    // ...

    abstract void run(); // I don't really know how to implement this
}

class Reptile extends Animal {
    void run() {
        print("I'm a reptil - I ain't got no legs - I can only crawl")
    }
}

class Mammal extends Animal {
    // Integer numberOfLegs;   // inherited from the Animal parent class

    void run() {
        if(numberOfLegs <= 2) {
            print("Run slow")
        } else {
            print("gallop fast")
        }
    }
}

class Bird extends Animal {
    void run() {
        print("Why run when you can fly!?")
    }
}

class Insect extends Animal {
    void run() {
        print("hippity hoppity")
    }
}


```



- Readable
There are so many classes now. 100 species -> 100 classes
    - not really an issue
        + you can use metaprogramming to reduce the code
            * templates / macros / decorators / preprocessors / reflection / generics
        + as a developer you will NEVER have to read all the files at the same time
            * you will be working on 1 functionality
                - at max you might have to read 3-4 files

- every single file in itself is very short and extremely easy to read!

- Testable

if we make a change to `Mammal.run()` does that break any of the testcases of `Bird.run()`?
No! - More testable.
Code is now de-coupled

- Extensible
Can we still add new species?
All we have to do is create a new class

- Maintainable
If 10 devs are working on 10 species - do we have merge conflicts?
No - better maintainability

Design a Bird

=> apart from other animals, now add Birds

```java
abstract class Animal {}

class Bird extends Animal {

    void fly() {
    }

}
```

🕊️ Different birds will fly in different ways

```java
[library] SimpleZooLibrary {
    // .dll .com .exe .so .o .class .jar
    // even if you have the source code of the library, you might not have write permissions to that source code

    abstract class Animal {}   => Penguin

    class Bird extends Animal {

        void fly(){
            if(species == "Sparrow") {
                print("fly low")
            } else if (species == "Eagle") {
                print("glide high")
            }
            /* else if (species == "Peacock") */
            // can I do this^^^?
        }
    }
}
```

SRP + OCP principle violation

Penguin

class Penguin extends Bird ?
no
fly()

```java
[executable] MyAwesomeZooGame { // client

    import SimpleZooLibrary.Animal;
    import SimpleZooLibrary.Bird;

    // I wish to add a new type of bird - Peacock
```

```
    public void main() {
        Bird b = new Bird();

        // interact with this bird
        b.fly()
    }

}
```

🐞 Problems with the above code?

- Readable
- Testable
- Maintainable

- Extensible - FOCUS!
As the client of the library, can we add a new bird species?
We can't - because we don't have write access to the library code

add more code (extend) => Yes

update an existing code (modify) => NO

🛠️ How to fix this?

==========
Open-Close Principle [OCP]
==========

- Your code should be open for extension, however, it should be closed for modification
--- even people who don't have access to your code should be able to extend your code! ---

❓ Why is modification bad?

typical dev cycle for a new feature

- dev - spend hours & hours to write code. Test it locally. Write comments. Ensure all commits are good. Finally submit a Pull Request (PR)
- Team - review the PR, ask you to make changes/improvements - iterations ... merged
- QA team - write new tests, integration tests
- Deployment
    + Staging servers - jmonitoring / tests/ metrics
    + Canary deployments / AB deployments
        * deployed to 5% of the user base
            - are there new exceptions
            - are the people complaining
            - have the ratings gone down
        * finally deploy the code

1.5 months

As the library writer, how can I design my classes, so that my end users (who are devs themselves) are able to extend my code without modifying my code?

→ LLD => Gangs of Four → introduced design patterns

```java
[library] SimpleZooLibrary {

    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 { // client

    import SimpleZooLibrary.Animal;
    import SimpleZooLibrary.Bird;

    // I wish to add a new type of bird - Peacock

    // I am able to add new functionality without touching the existing code!
    class Peacock extends Bird {
        void fly() { print("Pe-hens can fly, the male peacocks can't") }
    }

    public void main() {
        Bird b - new Bird();

        // interact with this bird
        b.fly()
    }

}
```

- Modification.

- Extension
The client can extend the code and add new functionality without having to modify the existing library code

- Readable
- Testable
- Extensible
- Maintainable

The fix was - remove if-else ladder and convert into inheritence

? Isn't the same fix that we used for the Single Responsibility Principle too?
Yes!

? Is the SRP == O/C ?
No. The solution was the same, but the intent was different
SRP =/= OCP

🔗 All the SOLID principles are tightly linked to each other

When you write good code / try to adhere to one of the SOLID principles - you might end up getting other ones for free

---

Can all birds fly?

```java
abstract class Animal {}
```

```java
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() {
      !?
   }

}
```

Penguin, Ostrich, Emu, Kiwi, Dodo .. are birds which cannot 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 fly method!

```java
abstract class Animal {}

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


class Kiwi extends Bird {
   // no void fly() here
}
```

🐞 Compiler Error!

`Bird` is an incomplete class (because it is marked abstract)
inside the `Bird` class, the method `fly` is the reason why it is incomplete

`Kiwi` is a complete class (because you haven't marked it as abstract), but at the same time, you have not provided the implementation for void fly

compliler - either implement fly, or mark kiwi as abstract

⚠ Throw a proper exception

```java
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") }}
```

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

🐞 This will violate expectations

```java
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") }}


class MyAwesomeZooGame {

   Bird getBirdFromUserSelection() {
      // show all the species of the available birds to user
      // let user select one type
      // create an object of that type
      if(userSelection == "Sparrow") {
         Sparrow s = new Sparrow("Tweety")
         return s
      } else if(userSelection == "Parrot") {
         Parrot p = new Parrot("Mitthu")
         return p
      } ... other cases
      // reflect and find all subclasses of the bird class
   }

   void main() {
      Bird b = getBirdFromUserSelection();

      b.fly();
   }
}



// ----------------


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

```

✅ Before extension
The above code works perfectly! Everyone is happy.
Dev, QA, User

❌ After extension
Even though we did NOT touch the existing code, the existing code breaks

==========
Liskov's Substitution Principle
==========

- Any functionality in the parent class, must also work for all child classes

- theoritical: any Parent class object must be replacable for any child class object

- any extension to a class should not break the existing class

🎨 How should we re-design this?

We understand that NOT all birds can fly
So let's make a distinction. Let us NOT have the `fly()` method inside the Bird class

```java
abstract class Animal {}

abstract class Bird extends Animal {
   abstract void eat();
   abstract void poop();
   // do NOT put the abstract void fly() here
}

interface ICanFly { // ISomeBehavior
   void fly();
}

class Sparrow extends Bird implements ICanFly {
   void eat() {}
   void poop() {}

   void fly() {}
}
class Eagle extends Bird implements ICanFly {
   void eat() {}
   void poop() {}

   void fly() {}
}


class Kiwi extends Bird { // note that Kiwi does NOT implement ICanFly
   void eat() {}
   void poop() {}

   // no need to implement void fly()
}


class MyAwesomeZooGame {

   ICanFly getFlyingBirdFromUserSelection() {
      // show all the species of the available birds to user
      // let user select one type
      // create an object of that type
      if(userSelection == "Sparrow") {
         Sparrow s = new Sparrow("Tweety")
         return s
      } else if(userSelection == "Parrot") {
         Parrot p = new Parrot("Mitthu")
         return p
      } ... other cases
      // reflect and find all implementations of the ICanFly interface
   }

   void main() {
      ICanFly b = getBirdFromUserSelection();

      b.fly();
   }
}

```

What should you anticipate?

- changes in requirements
- database migrations / adding new columns in tables / adding new indexes / optimizing certain queries / sharding the database
- specializing a particular class
  + user
  + free / paid / premium user
- strategies
  + different features
- feature flags

Pre-pare for all of these changes - by writing good code from the start

Low Level Design - how to write good code

- Object Oriented Programming
- SOLID Principles
- Design Pattern
  + Singleton
  + Builder
    * language specific - yes for java, but no for python
  + Factory
  + ...
- Database Schema Design
  + Indexes
  + Normalize
  + Optimize queries
- ER-diagrams / Class diagram
- REST API design
- A ton of case studies
  + Snake-Ladder
  + Chess
  + Parking Lot
- Machine Coding rounds / Take home assessments

-- --

What language do you have to know to be a developer?
- doesn't matter
- typists / thinkers
  + problem solving - Algorithms & Data Structures
  + design - HLD / LLD / Database
  + communication - HM rounds
- know at least 1 programming language
  + which one? doesn't matter
  + modern language
    * Python, C++, Java, C#, F#, Javascript (Typescript), Scala, Swift, Rust, Kotlin, Golang, Haskell, Php
      - if you're already experienced with any of these, stick with it
    * Python / Java (Kotlin, Scala) / Javascript (Typescript) - can NOT go wrong with these
      - if you're starting fresh - choose any of the above

Should a backend dev learn SOLID principles
- ABSOLUTELY! Even for frontend roles, you will have Low-Level-Design (LLD) rounds
  + Spring Boot / Django / Laravel / Rails
  + React / Svelte / Angular
  + all these frameworks use SOLID, Design Patterns, LLD in a lot of depth
- For entry level roles, maybe you can skip these
  + SDE 2+ at a good company (Adobe/Amazon/Google) - absolutely MUST

Effects of AI

Short Term (5 years)
- The barrier to entry for coding will reduce
  + increase your competition

    * 100,000 people who have made small project / apps by using easy frameworks - but they don't have in-depth understanding of how & why things works
- hiring bar will increase
  + salaries also go up

Recession

- the number of jobs has decreased (temporarily)
- the competition is higher
- the salaries are also sky-high

What do you have to do - make sure that you have in-depth understanding of things
Superficial knowledge will no longer work

Long Term (10+ years)
- I've absolutely no idea!
- If we have strong AGI, then it becomes impossible to predict the future

---

Single Responsibility
Open Closed
Liskov's Substitution

✈ What else can fly?
==========

```java
abstract class Animal {}
abstract class Bird extends Animal {}

interface ICanFly {
   void fly();

   // setup for birds flying
   void spreadWings();

   void smallJump();
}

class Sparrow extends Bird implements ICanFly {
   void fly() { ... }
}
class Eagle extends Bird implements ICanFly {
   void fly() { ... }
}

class Kiwi extends Bird {
}

class Shaktiman implements ICanFly {
   void fly() { /* rotate super fast */ }

   void spreadWings() {
      // 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]

Apart from birds, what else can fly?
- Kites (patang)
- Aeroplanes
- Drones
- Abhishek's mummy's chappal
- Shaktiman
- Baloons
- Papa ki Pari

==========

Interface Segregation Principle

==========

- Keep your interfaces minimal → ideally your interface should contain only 1 method

- No code (the clients/users of your code) should not be forced to implement methods that they don't need

(Java) ideally all your interfaces should be FI

How will you fix `ICanFly`?

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

interface IFliesLikeBird{
   // setup for birds flying
   void spreadWings();

   void smallJump();
}

```

"ISP is SRP for interfaces"

anonymous class → FI (single method) → lambda exp. + stream

Split the large interface into multiple smaller, more specific interfaces

🔗 Isn't this just the Single-Responsibility Principle applied to interfaces?

Yes. And that's okay.

Rules vs Guidelines

==========

- Rules
    + mandatory - must be followed
    + if you break them - something bad will happen
        * go to jail
        * die
        * pay a penalty
- Guidelines
    + good to follow - not enforced
    + It's okay to sometimes not follow the guidelines
    + Very important to know WHEN & WHY to violate the guidelines

SOLID - guidelines

Hackathon - 2 hours to build a running app end-to-end

In a lot of startups - you might see code that doesn't follow these principles

---

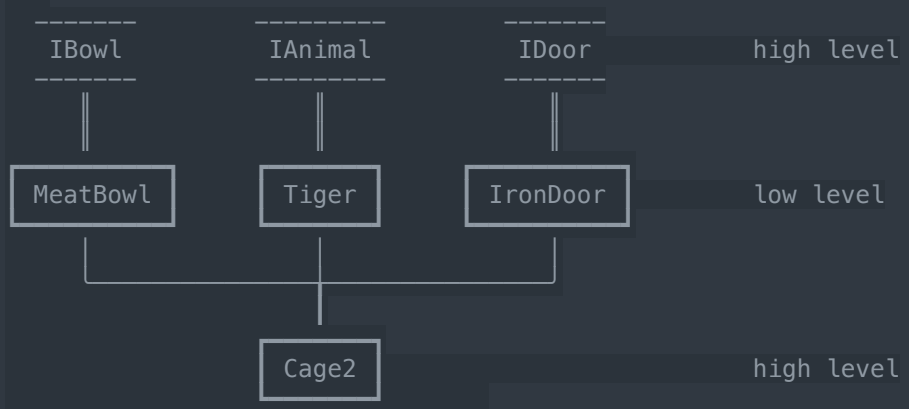
We've designed a bunch of animals - so now let's shift focus and look at the infrastrcuture of the Zoo

Design a Cage

```java
// High-level code - abstractions (superficial structure)   (abstract class / interface)
// Low-level code - implementation details (exact code)

interface IBowl { void fill(); void clean(); void startMeal(); }   // High level abstraction

class MeatBowl implements IBowl {  void fill() { /* fill with meat / add enzymes / grind it / split it by size */ }}    // Low Level code - details
class FruitBowl implements IBowl {} // Low level
class GrainBowl implements IBowl {} // Low level

interface IDoor {  void lock();  void unlock();  void resistAttack(); } // High level
class WoodenDoor implements IDoor {} // Low level
class IronDoor implements IDoor {} // Low level
class AdamantiumDoor implements IDoor {} // Low level


// Controller/Manager/Delegator class   - High level abstraction

class Cage1 { // for birds

   FruitBowl bowl = new FruitBowl("apples", "grapes");
   WoodenDoor door = new WoodenDoor();

   List<Bird> residents;

   public Cage1() {
      // do some initialization
   }

   public void startLunch() {
      for(Bird b: residents) {
         bowl.feed(b);               // delegate the task to the bowl
      }
   }

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

class Cage2 { // big cats
   MeatBowl bowl = new MeatBowl("chimkens", "shrimps", "humans");
   IronDoor door = new IronDoor();

   List<Cat> residents;

   public Cage2() {
      // do some initialization
   }

   public void startLunch() {
      for(Cat c: residents) {
         bowl.feed(c);
      }
   }
```

```
    public void resistAttack(Attack attack) {
        door.resistAttack(attack);
    }
}

class MyAwesomeZooGame {
    void main() {
        Cage1 birdCage = new Cage1();
        Cage2 kittyCage = new Cage2();
    }
}
```

🐞 What is wrong with this code?
- duplication
- no code reuse
- if we have 100 cages in the zoo, we will have to create 100 classes
    + our client (the zoo game) must be aware of how these classes work
    + which class to use for which type of cage



High-level class Cage2 depends on Low level details MeatBowl, Tiger and IronDoor

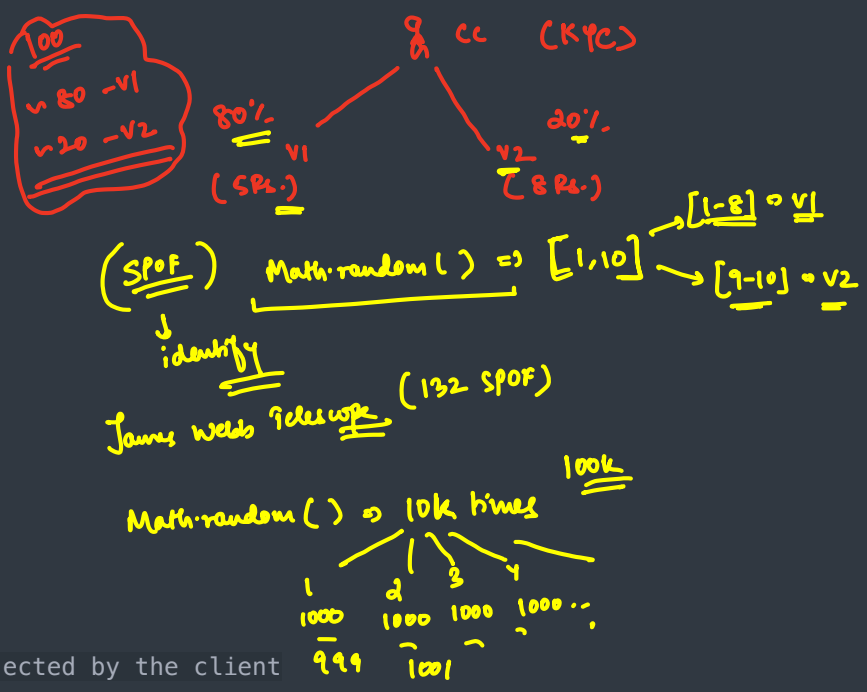
==========
Dependency Inversion Principle
==========

- High-level code should NOT depend on low-level code.
- High level code should only depend on high level abstractions

But how?

==========
Dependency Injection **- why to achieve the principle**
==========

- Instead of creating your own dependencies, you let your client provide (inject) the dependencies into you




```java

interface IBowl {}
class MeatBowl implements IBowl {}
class FruitBowl implements IBowl {}
class GrainBowl implements IBowl {}

interface IDoor {}
class WoodenDoor implements IDoor {}
class IronDoor implements IDoor {}
class AdamantiumDoor implements IDoor {}


class Cage {
   IBowl bowl;
   IDoor door;

   List<Animal> residents;

   public Cage(IBowl bowl, IDoor door) {
      // use the dependencies provided/injected by the client
      this.bowl = bowl;
      this.door = door;
   }

   public void startLunch() {
      for(Animal a: residents) {
         bowl.feed(a);                // delegate the task to the bowl
      }
   }

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



class MyAwesomeZooGame {
   void main() {
      Cage birdCage = new Cage(new FruitBowl(), new WoodenDoor());
      Cage kittyCage = new Cage(new MeatBowl(), new IronDoor());
   }
}
```


Spring Boot / Django / React   - heavily use the dependency injection

Enterprise Code
==========

- Google/Amazon
- you might see "over-engineered" code
- if you don't know LLD
   + you will not be able to understand any code
   + everything looks so complex
- if you know LLD
   + you won't even have to read the code
   + if you know the patterns/principles
   + just by looking at the filename, you will know EXACTLY what the code does!


```java

class RazorPayPaymentGatewayRecieptBuilder implements IPaymentGatewayReceiptBuilder {
```

```
    ...
}

SimpleFileLogger logger = SimpleFileLoggerFactory.getInstance();

```

Quick Recap

SOLID Principles

- Single Responsibility
- Open/Close
- Liskov's Substitution
- Interface Segregation
- Dependency Inversion
  + Dependency Injection

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! ============================
```