💎 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/)

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 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 time writing new code!

⏱️ Where does the rest of the time go?

- reading code, KTs, debugging, testing, refactoring, requirements analysis, understanding other people's code
- meetings, chai-sutta breaks, TT

✅ Goals

We'd like to make our code

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

Robert C. Martin 👴 Uncle Bob

💎 SOLID Principles

- Single Respnsibility
- Open–Close
- Liskov's Substitution
- Interface Segregation
- Dependency Inversion

Inversion of Control / Interface Segregation
Dependency Inversion / Dependency Injection

💭 Context

- Zoo game 🦊
- Modeling various animals

🐶 Design an Animal

```java
class Animal {
   // OOP - class -> concepts / entities / blueprints

   // attributes [ properties ]
   String color;
   int numberOfLegs;
   String species;
   double weight;
   bool isHerbivore;
   bool hasWings;
   bool canBreatheUnderwater;


   // behavior [ methods ]

   void run();

   void fly();

   void makeSound();

   void eat();

   void attack();
}

```

 Different animals behave differently

```java
class Animal {
   // .. properties
   String species;

   // .. methods

   void run() {

      if(species == "Reptile") {
         print("I can't run, I can only crawl")
      } else if (numberOfLegs > 2) {
         print("I have more than 2 legs, so I can gallop")
      } else if (numberOfLegs == 2) {
         print("Hop adorably")
      }

   }
}

class AnimalTester {
   void testRun() {
      Animal snek = new Animal(...);

      assert snek.run() == "I can't run, I can only crawl"


      Animal horse = new Animal(...);

      assert snek.run() == "I have more than 2 legs, so I can gallop"
   }
}

```

 Problems with the above code?

❓ Readable
Yes, I can totally read & understand it!

However, as the number of species grows, or the logic becomes more complex, this if-else ladder will be difficult to read

❓ Testable
It seems that I can test it!
However, the species are tightly coupled. Changing behavior of 1 species might break many other species

❓ Extensible
explore this in detail later

❓ Maintainable
if multiple devs are maintaining different species, then there will be merge conflicts

🛠️ How to fix this?

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

- Every function/class/module/unit-of-code should have one, simple, well-defined responsibility

- Any unit-of-code should have exactly 1 reason to change

- if some piece of code is serving multiple responsibilities - break it down into invidual pieces

```java
abstract class Animal {
   String species;
   String color;

   abstract void run();
}

class Reptile extends Animal {
   void run() {
      print("I can't run, I can only crawl")
   }
}

class Mammal extends Animal {
   void run() {
      print("I can run fast")
   }
}

class Bird extends Animal {
   void run() {
      print("I can hop!")
   }
}
```

- Readable
There are too many classes now! Non-issue
Too many classes is not an issue, because you will be working with 1 or maybe a handful of classes at any given time
Each class is extremely simple to read & understand!

- Testable
Each subclass is independently testable. Test-cases are no longer tightly coupled

- Extensible

will come back to this later

- Maintainable
at least we have mitigates some merge conflicts

Design a Bird

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

   void fly() {
      ...?
   }
}
```

Different birds fly differently

```java
[library] Zoo {
   // .com .dll .class .jar .o

   abstract class Animal {
      String species;
       ...
   }

   class Bird extends Animal {
      void fly() {
         if(species == "Sparrow")
            print("fly low")
         else if (species == "Eagle")
            print("glide high")
         else if(species == "Dodo")
            print("dude, I'm extinct - no can't fly!")
      }
   }

}

[executable] Client {
   import Zoo.Animal;
   import Zoo.Bird;

   // suppose I wish to add a new species of Bird - Peacock
   // I need to add an else-if case
   // I don't have write access to the library code
   // I can't do it!


   class Main {
      void main() {
         // .. use the classes
      }
   }
}
```

Problems with the above code?

- Readable
- Testable
- Maintainable

- Extensible - FOCUS!
If we've imported the code from library, we can't modify the code, because we don't have write access to it
For this reason, we're unable to extend the code

Kite API

🛠️ How to fix this?

==========
⭐ **Open-Close Principle**
==========

- Your code should be CLOSED for modification, yet still, OPEN for extension!

❓ Why is modification bad?

- Dev - write code, test locally, commit - Pull request
- Other devs review the MR, ask you to make changes .. repeats .. merged
- QA team - write new tests, integration, end-to-end tests
- Deployment
    + staging servers - monitoring, tests, metrics
    + A/B testing
        * deploy to only 5% of the userbase
            - monitor it - performance, bugs, user satisfaction, ...
        * deployed to the entire userbase

```java
[library] Zoo {
   // .com .dll .class .jar .o

   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") }
   }
   ...

}

[executable] Client {
   import Zoo.Animal;
   import Zoo.Bird;
   import Zoo.Sparrow;
   import Zoo.Eagle;

   // I wish to add a new Bird species - Peacock

   class Peacock extends Bird {
      void fly() { print("only pe-hens can fly, not the males") }
   }

   class Main {
```

```
      void main() {
         // .. use the classes
      }
   }
}

```

- Modification.

- Extension
now it is extensible!
I should be able extend code which I can't modify!

- Readable
- Testable
- Extensible
- Maintainable

? Isn't this the same thing we did for the Single Responsibility Principle as well?

Certainly yes!

? Does that mean that SRP == Open-Close Principle?
No. The way of achieving it was same, however, the intention was different

🔗 All SOLID principles are interlinked.

🐓 Can all birds fly?

No!

```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() {
      ...?
   }
}

```

? How do we solve this?

- Throw exception with a proper message
- Don't implement the `fly()` method

- Return `null`
- Redesign the system

🏃 Let's not implement the `fly()` method

```java
class Kiwi extends Bird {
   // no fly method
}
```

🐞 Compiler will complain! Compiler enforces you to either implement the void fly or to make the Kiwi class abstract too

⚠️ Throw an exception

```java
class Kiwi extends Bird {
   void fly() {
      throw new NonFlyingBirdException("Kiwis don't fly bro!")
   }
}
```

🐞 This violates expectations

```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 Client {
   void main() {
      Bird b = getBirdFromUserInput(); // returns any Bird subclass
      b.fly();
   }
}

// life is perfect! everything is tested

// added new functionality

class Kiwi extends Bird {
   void fly() {
      throw new NonFlyingBirdException("Kiwis don't fly bro!")
   }
}
```

✅ Before extension
code was thoroughly tested and working fine

❌ After extension
without changing any existing code, the existing code magically breaks!

Class resumes at 6.45

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

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

- any extension to existing code should not break the existing code

🎨 How to re-design the system?

```java
abstract class Bird extends Animal {
   bool hasBeak;

   void speak() {}

   // we won't have fly() here - because not all birds can fly
}

interface ICanFly {
   void fly();
}


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

class Kiwi extends Bird {
   // kiwi does not fly
   // so kiwi will NOT implement ICanFly
}


class Client {
   void main() {
      ICanFly b = getBirdFromUserInput(); // returns any Flying Bird subclass
      // Runtime Polymorphism
      b.fly();
   }
}


```


```py
from abc import ABC, abstractmethod

class Bird(ABC): # abstract
   def speak(self):
      ...

class ICanFly(ABC): #abstract
   def fly(self):
      ...

class Sparrow(Bird, ICanFly): #multiple inheritence
   ...

class Kiwi(Bird):
   ...
```

```

---

## ✈ What else can fly?

- Birds can fly
- but are there non-Bird things that can also fly?

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

interface ICanFly {
   void fly();

   void smallJump();
   void spreadWings();
}

class Shaktiman implements ICanFly {
   void fly() { print("spin fast") }
   void spreadWings() {
      // SORRY SHAKTIMAN!
   }
}
```

Aeroplane, Kite, Shaktiman, Mummy's Chappals, Dreams, Missiles, ...

> ? 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]

No. Because not all the things that can fly have wings..

⭐ Interface Segregation Principle

- keep your interfaces minimal

- No code (your code, or code written by your clients) should be forced to implement a method it does not need

How will you fix `ICanFly`?

Split the interface into multiple interfaces

🔗 Looks like Liskov's principle
🔗 This is just SRP applied to interfaces!

Composition over inheritance
```

Rules vs Guidelines

- Rules
    + If you violate a rule, someone will punish you.
    + Rules are enforced!
- Guidelines
    + Not enforced
    + Good-to-have things that make your life easier
    + They can be broken if needed

SOLID Principle - guidelines
Sometimes, it is okay to break these guidelines
Hackathon - 3 hours - you have to develop a working MVP
Startup - iterate fast - find product-market-fit

But you're a large company like Google, which deploys to billion users, it becomes crucial to write high quality code

Please don't over-engineer your code - it is extremely important to know when to apply things

---

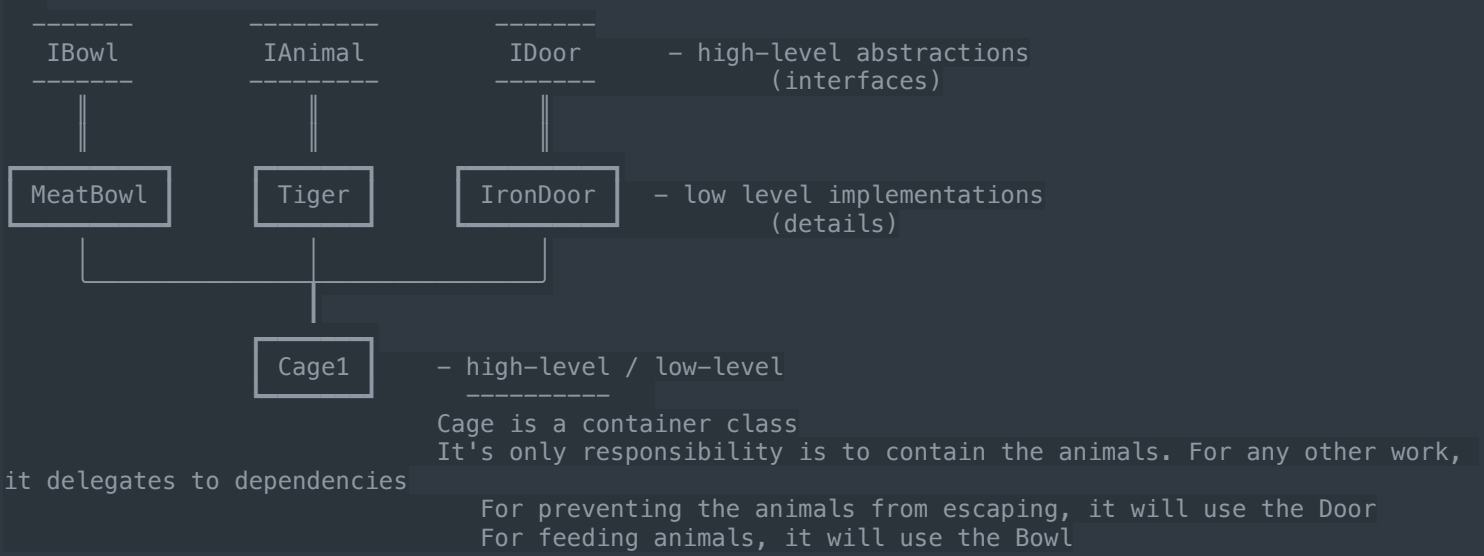
Now that we've the necessary characters, let's design some structures

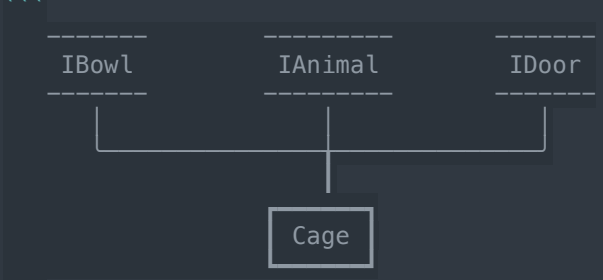
🗑️ Design a Cage

```java
interface IDoor {}
class WoodenDoor implements IDoor {}
class IronDoor implements IDoor {}
class AdamantiumDoor implements IDoor {} // for wolverine

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

class Cage1 {
   // birds

   GrainBowl bowl;
   WoodenDoor door;

   List<Bird> birds;

   public Cage1() {
      bowl = new GrainBowl()
      door = new WoodenDoor()
      birds.add(new Peacock(...))
      birds.add(new Eagle(...))
      birds.add(new Sparrow(...))
   }

}


class Cage2 {
   // big cats

   MeatBowl bowl;
   IronDoor door;

   List<Animal> cats;

   public Cage1() {
      bowl = new MeatBowl()
      door = new IronDoor()
      cats.add(new Tiger(...))
      cats.add(new Lion(...))
      cats.add(new Meerkat(...))
```

```
    }

}


// Cage3
// Cage4
// Cage100

class Zoo {
    void main() {
        Cage1 birdCage = new Cage1();
        Cage2 catCage = new Cage2();
        // ...
    }
}


```

🐞 Lot of code repetition




```

==========
⭐ **Dependency Inversion Principle**
==========

- High level modules should NOT depend on low level details
- Instead, high level modules should only depend on high-level abstractions (interfaces/abstract classes)




But how?
```

Dependency Injection

- Instead of creating our dependencies ourselved, we will inject them (via constructor, via method, ...)

```java
interface IDoor {}
class WoodenDoor implements IDoor {}
class IronDoor implements IDoor {}
class AdamantiumDoor implements IDoor {} // for wolverine

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

class Cage {
   IDoor door; // dependencies
   IBowl bowl;
   List<Animal> animals;

   /*                  inject them vvvvvv                */
   public Cage(IDoor door, IBowl bowl, List<Animal> animals) {
      this.door = door;
      this.bowl = bowl;
      this.animals.addAll(animals);
   }
}

class Zoo {
   void main() {

      Cage birdCage = new Cage(new WoodenDoor(),
                               new GrainBowl(),
                               Arrays.asList(new Peackock(), ...));

      Cage catCage = new Cage(new IronDoor(),
                              new MeatBowl(),
                              Arrays.asList(new Tiger(), ...));

   }
}
```

Enterprise Code

- code in large companies like Google/Amazon
- very very long variable & class names
- a lot of design pattern - every piece of code will be following some pattern
- extensive logging and exception handling

```java
SimpleFileLogger simpleFileLogger = SimpleFileLogger.getInstance();
```

- project may involve 100s of developers
   + devs come and go
- these companies care a lot about extensibility & maintainability
- anticipate requirements pre-emptively and design for that from the starting

- if you're not familiar with LLD (Low Level Design), or SOLID, or Design Patterns
   + you get selected at Google
   + and you look at codebase

    + jump off the building
- once you're familiar with these
    + you won't even have to read the code to understand it!

⏱️ Quick Recap

SOLID Principles

• Single Responsibility - every piece of code should have exactly 1, well-defined responsibility
    - if not, then break it down into smaller pieces

• Open-Close - your code should be open for extension, yet, closed for modification
    - even if you don't have write access, you should still be able to extend the code

• Liskov Substitution - any functionality that works with parent class must also work with child classes
    - don't violate existing expectations

• Interface Segregation - keep your interfaces minimal
    - so that your client don't have to implement something they don't need

• Dependency Inversion - high level modules should depend on high-level abstractions and not low level details
    + Depdency injection
        * don't create dependencies, instead inject them

🎁 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

B

> ❓ 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

A

? 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

A

? 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

A

? 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

C

? 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

D

==========

⭐ **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! ============================

SOLID Principle - guidelines on how to structure your code - some constraints that you code should satisfy

Design Patterns - battle-tested solutions to common problems

Dependency Inversion - process
Dependency Injection - way to achieve that process - pass dependencies via function parameters

Inversion of Control - there is a framework that is sitting on top and is in control

Python, Javascript, Java - most amount of jobs & opportunities
Python - my fav - super fun and easy to lear