COURSEWARE

Professional Skills

Agile Fundamentals

Jira

Git

Databases Introduction

Java Beginner

Maven

Testing (Foundation)

Java Intermediate

- Optionals
- JDBC CRUD
- Exceptions
- SOLID Principles
- Single Responsibility
- Open/Closed
- Liskov Substituiton
- Interface Segregation
- Dependency Inversion
- Best Practice
- Design Patterns
- Creational Design Patterns
- Structural Design Patterns
- Behavioural Design Patterns
- Collection & Map
- HashSets
- HashMaps
- Enums
- Logging
- Generics
- Lambda Expressions
- Streams
- Complexity
- Input and Output
- Local Type Inference

HTML

CSS

Liskov Substituiton

Contents

- Overview
- Liskov In Action
  - The `Animal.java` class
  - Fixing `Animal.java`
- Tutorial
- Exercises
  - ExtensionBuilder

Overview

In object-oriented programming, the third of the **SOLID Principles** is **L** - which stands for **Liskov Substitution**.

The *Liskov Substitution Principle* states that functions which use pointers to *base classes* (parent classes) must be able to use objects of *derived classes* (child classes) *without knowing it*.

Liskov In Action

The `Animal.java` class

Let's unpack this by looking at a program containing four classes:

- `Animal.java`
- `Bird.java` (abstract, extends from Animal)
- `Penguin.java` (extends from Bird)
- `Owl.java` (extends from Bird)

▶ Animal
▶ Bird
▶ Penguin
▶ Owl

Currently, this does not adhere to the *Liskov Substitution Principle*, because the `Bird` base type is not directly substitutable by the `Penguin` derived type.

Therefore, in `Animal.java`, we must first know which derived type we are using before running any of the methods in `Bird.java`.

Fixing `Animal.java`

Let's reorder this code, so that the derived types are directly substitutable with their base types.

We'll do this by writing two new *abstract classes* - one for a bird which can fly (`FlyingBird.java`) and one for a bird which can't (`FlightlessBird.java`):

▶ FlyingBird
▶ FlightlessBird

We'll then move the `fly()` method out of `Bird.java`...

▶ Bird

...and into `Owl.java`, since not all objects of type `Bird` can fly:

▶ Owl

Since we want objects of type `Penguin` to not be able to `fly()`, we'll give `Penguin.java` its own method `flap()` instead:

▶ Penguin

Javascript
Spring Boot
Selenium
Sonarqube
Advanced Testing (Theory)
Cucumber
MongoDB
Express
NodeJS
React
Express-Testing
Networking
Security
Cloud Fundamentals
AWS Foundations
AWS Intermediate
Linux
DevOps
Jenkins Introduction
Jenkins Pipeline
Markdown
IDE Cheatsheet

By doing this, we can be sure that only `Owl.java` will ever inherit the `fly()` method.

It also ensures that `Owl.java` is the only class that can ever be passed to the `learnToFly()` method in `Animal.java`.

As such, we no longer need to perform any checks on the type of object passed into `learnToFly()`!

▶ Animal

This now adheres to the Liskov Substitution Principle, because `Animal.java` can use the `fly()` method in `Owl.java` without ever directly referring to an object of type `Owl`.

Tutorial

There is no Tutorial for this module.

Exercises

ExtensionBuilder

Consider the following four classes, which is meant to upgrade apartments with new bedrooms:

- `BedroomAdder.java`
- `Apartment.java` (abstract)
- `Penthouse.java` (extends from Apartment)
- `Studio.java` (extends from Apartment)

▶ BedroomAdder
▶ Apartment
▶ Penthouse
▶ Studio

These classes violate the *Liskov Substitution Principle* because `BedroomAdder.java` *seems like* it accepts any object of type `Apartment` which is fed into it, but *actually* checks the sub-class of the object to ensure that no object of type `Studio` is upgraded.

Refactor the program using the following three classes to ensure that it adheres to the Liskov Substitution Principle to complete this exercise:

- `BedroomAdder.java`
- `Penthouse.java`
- `Studio.java`

▶ See solution