# OOP Principles, SRP, LSP, Object Composition

CO5225 – Andrew Muncey

# Contents

- Core Principles
  - Inheritance
  - Abstraction
  - Encapsulation
  - Polymorphism
- Single Responsibility Principle
- Whole-part relationships and Object Composition
- Liskov Substitution Principle

## Inheritance

- General routines to handle general properties of an object
- More specific routines to cover specific characteristics of different kinds of the object.
- Specific objects inherit from the more general object
- E.g. Kettle **inherits from** HomeAppliance
  - A Kettle **IS** a home appliance
  - Will have all properties of HomeAppliance & more

# Inheritance in Kotlin

- Not enabled by default
- Must mark the class with the ‘open’ keyword

## Abstraction (Concept)

- “the quality of dealing with ideas rather than events”
- How something works
- You’re allowed to look at an object at a high level (i.e. not at the details)
- Take a simpler view of a complex concept

## Abstraction in Kotlin

- Abstract classes allow us to defer implementation of functionality to a subclass
- Interfaces allow us to show how something should ‘be’ without defining the ‘how’

## Encapsulation (Technique)

- You are not allowed to look at an object at anything other than a high level
- Hides the internal workings of a class – we only see the publically accessible methods.
    - The class is treated as a black box

## Implementing Encapsulation

- Public variables (properties) for exposed data
- Private variables for private data
- Constructors are one way of setting up instance variables

## Visibility modifiers in Kotlin (in classes)

- Private: only visible inside the class
- Protected: visible in class, and subclasses
- Internal: visible in the same module
    - Module is a set of Kotlin files compiled together
- Public: visible by anything that can see the class
    - This is default if no visibility modifier is specified

## Polymorphism

- Literally “Many forms”
- Builds on inheritance
- Allows us to use different types of objects in (for example) a method without knowing what it is until run time
  - Provided these different types share a superclass

## Polymorphism demo

- Cooking with Kotlin demo

## Single Responsibility Principle

- “A class should have only one reason to change”
  - If a class has multiple responsibilities, then there will be more than one reason for it to change (One responsibility may still mean multiple methods)
- e.g. A washing machine class would only change if the process of washing clothes changed
- Consider a DVD-TV combo
  - Optical media changes - need to play Blu-ray instead of DVD
  - TV format changes - High Def

## Whole-Part relationships

- The idea that a class (a whole) can be comprised of other objects (the part)
- This is the idea that objects can contain other objects
  - Simply having a String property for a class constitutes an object (the string) being part of a whole (the class)
- Consider the Employee class, an employee may have an address
  - Could include address fields in Employee class – messy
  - Instead create an Address class – this can be used anywhere an address is needed
- Likely that some objects will contain collections of other objects
  - e.g. Employee who has held a series of posts at an organisation might have a collection of Contract Objects
- Allows us to keep code more modular and maintainable

## Two types of Whole-Part relationship

**Aggregation (Has a)**

- Properties exists independently of class (implies Usage)
- Parts may be shared
- E.g. Passengers in Car object
  - Destroy the car ≠ destroy the passengers

**Composition (Must have a)**

- Properties has no meaning or purpose without class (implies Ownership)
- Parts are not shared
- E.g. Gearbox in Car object
  - Lifetime dependency
  - E.g. Destroy the car = destroy the gearbox

# Liskov Substitution Principle

- *“If S is a subtype of T, then objects of type T may be replaced by objects of type S”*
- *Or “Subtypes must be substitutable for their base types”*
- *Behaviour of the method must not change.*
- *One example of this in action in Android is when we call the addView: method of a Layout - the method requires a View, but we can, for example, pass a Button.*
    - *Button inherits from TextView which inherits from View, and therefore can be substituted in the addView: method*

## LSP

- While Polymorphism allows us to use subclasses, LSP requires subclasses to be substitutable for the parent type.
- A common way LSP is violated would be when an if/else statement is used to change behaviour based on (sub)type

- Demo – LSP Violation

## Reminder

- We have used interfaces and abstract classes in the practical sessions. Ensure you are able to explain their purposes, uses and advantages

## Reading

- Martin, R.C. (2012). *Agile Software Development.* Upper Saddle River, NJ: Pearson. Pages 95-98 & 111-126
- Liskov, B. (1988). Keynote address-data abstraction and hierarchy. ACM Sigplan Notices, 23(5), 17-34.