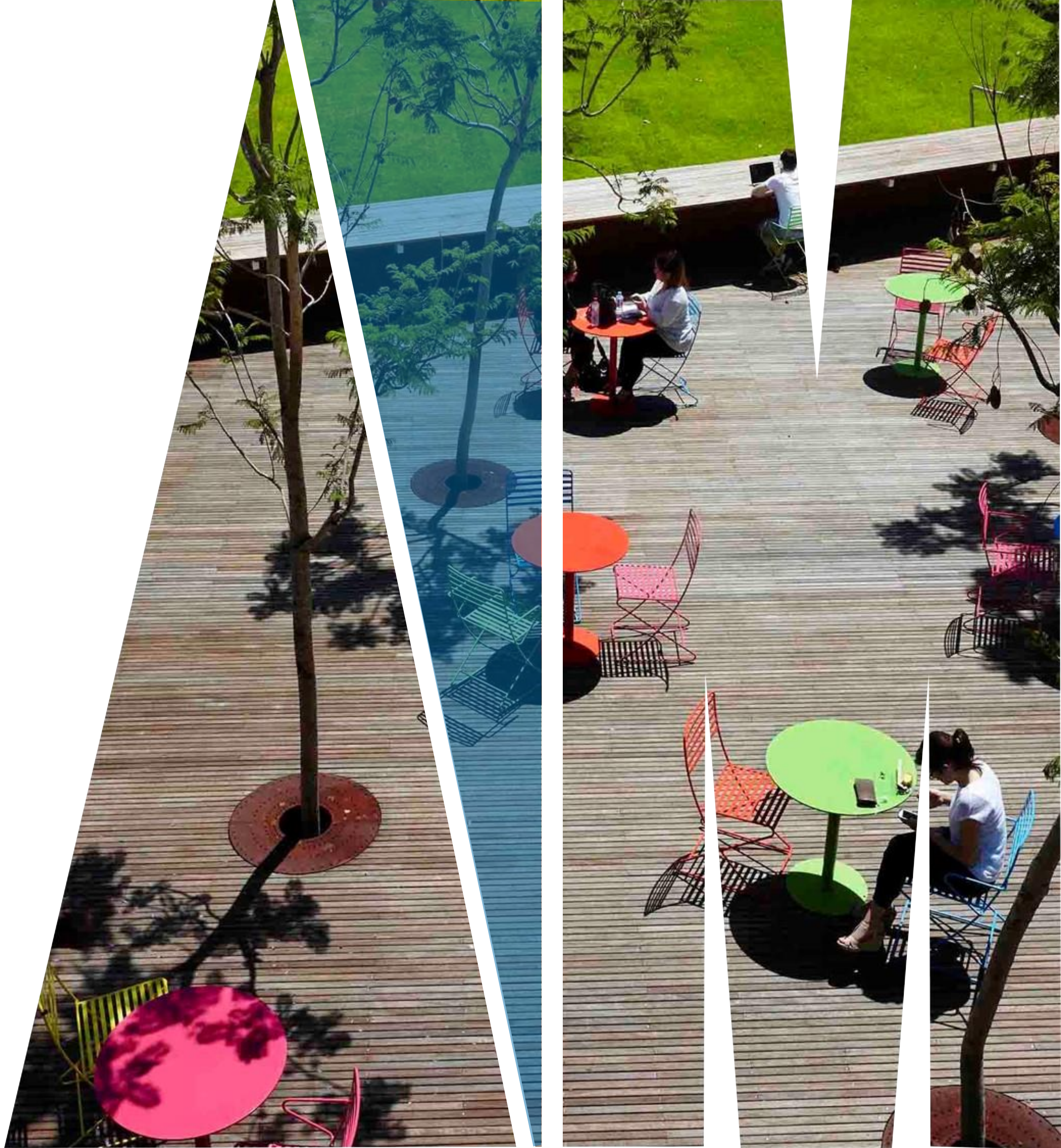
MONASH
University

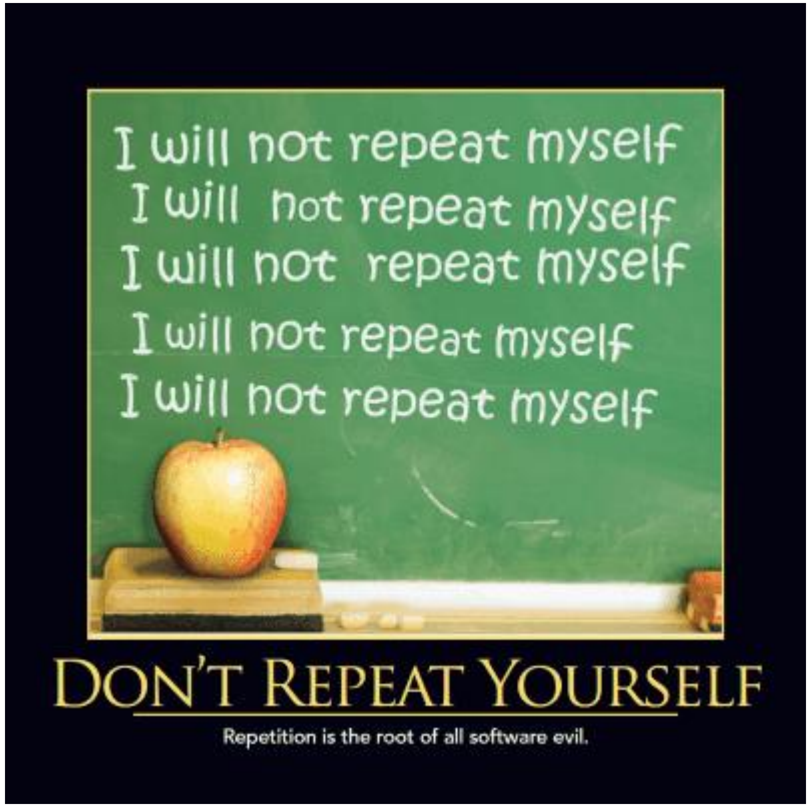
FIT2099 Object-Oriented Design and Implementation

Review of
OO Design Principles



PRINCIPLE A
DON’T REPEAT YOURSELF (DRY)

Don't repeat yourself" (DRY) is a principle of software development aimed at reducing repetition of software patterns, replacing repeated code with **abstractions** to avoid redundancy.



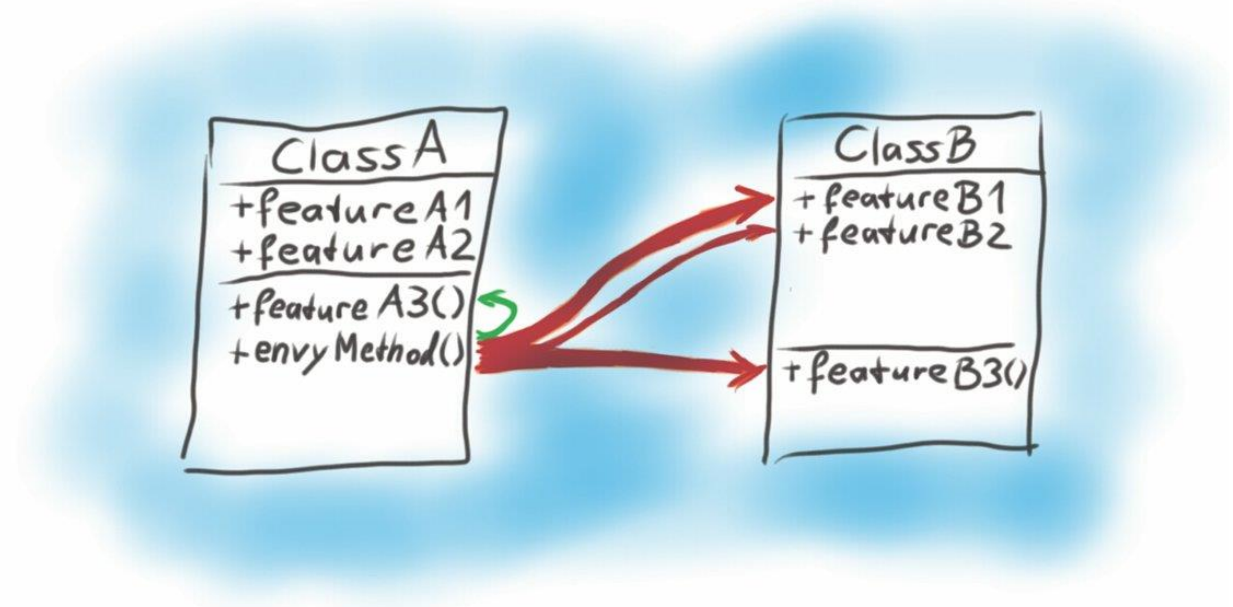
MONASH University

PRINCIPLE B

CLASSES SHOULD BE RESPONSIBLE FOR THEIR OWN PROPERTIES

As a basic rule, if things change at the same time, you should keep them in the same place.

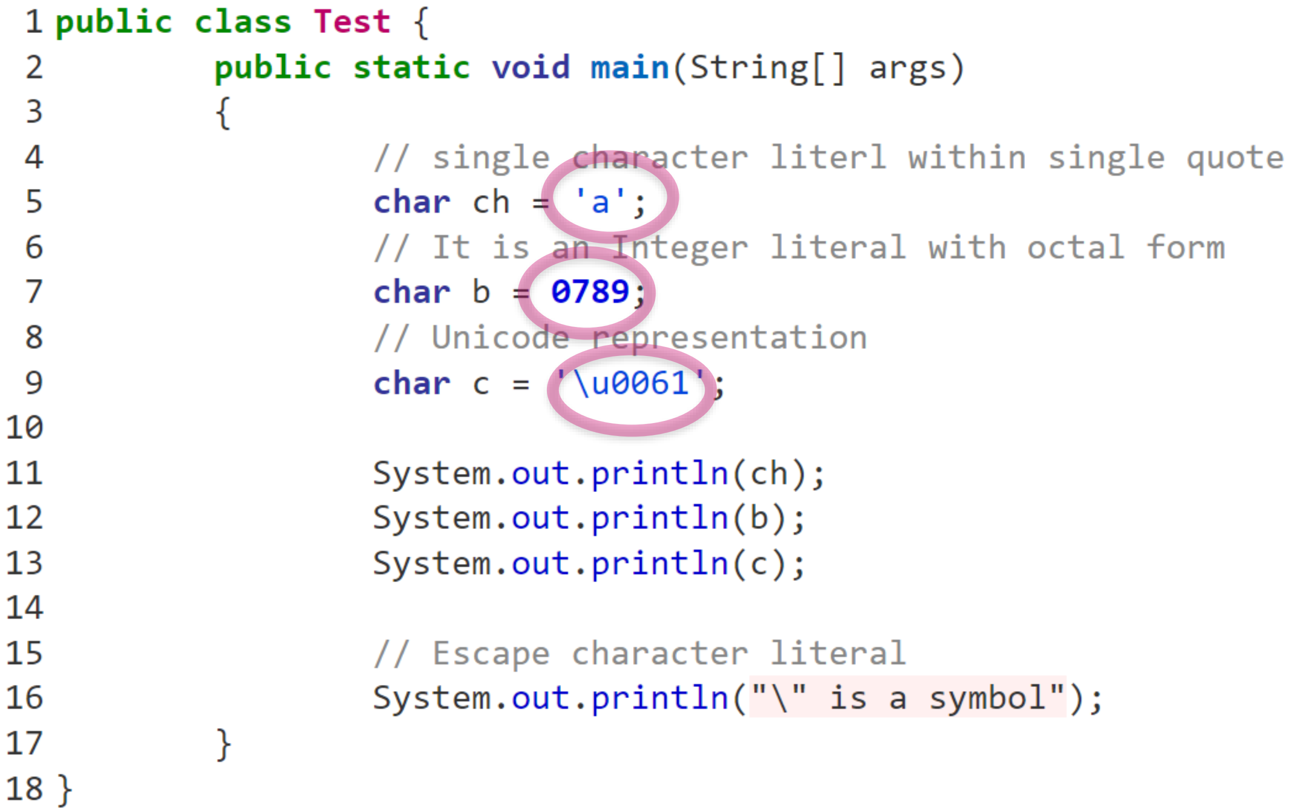
Note: this is related to a design smell called “feature envy” and a principle called “single-responsibility (SRP). We will more deeply cover these later.



PRINCIPLE C
AVOID EXCESIVE USE OF LITERALS

Every piece of software contains **literals** (usually numbers, strings or booleans).

These are **fixed values in source code**, commonly related to application configuration, parts of the business logic, natural or language constants, etc..



```
public class Test {
        public static void main(String[] args)
        {
                // single character literl within single quote
                char ch = 'a';
                // It is an Integer literal with octal form
                char b = 0789;
                // Unicode representation
                char c = '\u0061';

                System.out.println(ch);
                System.out.println(b);
                System.out.println(c);

                // Escape character literal
                System.out.println("\" is a symbol");
        }
}
```

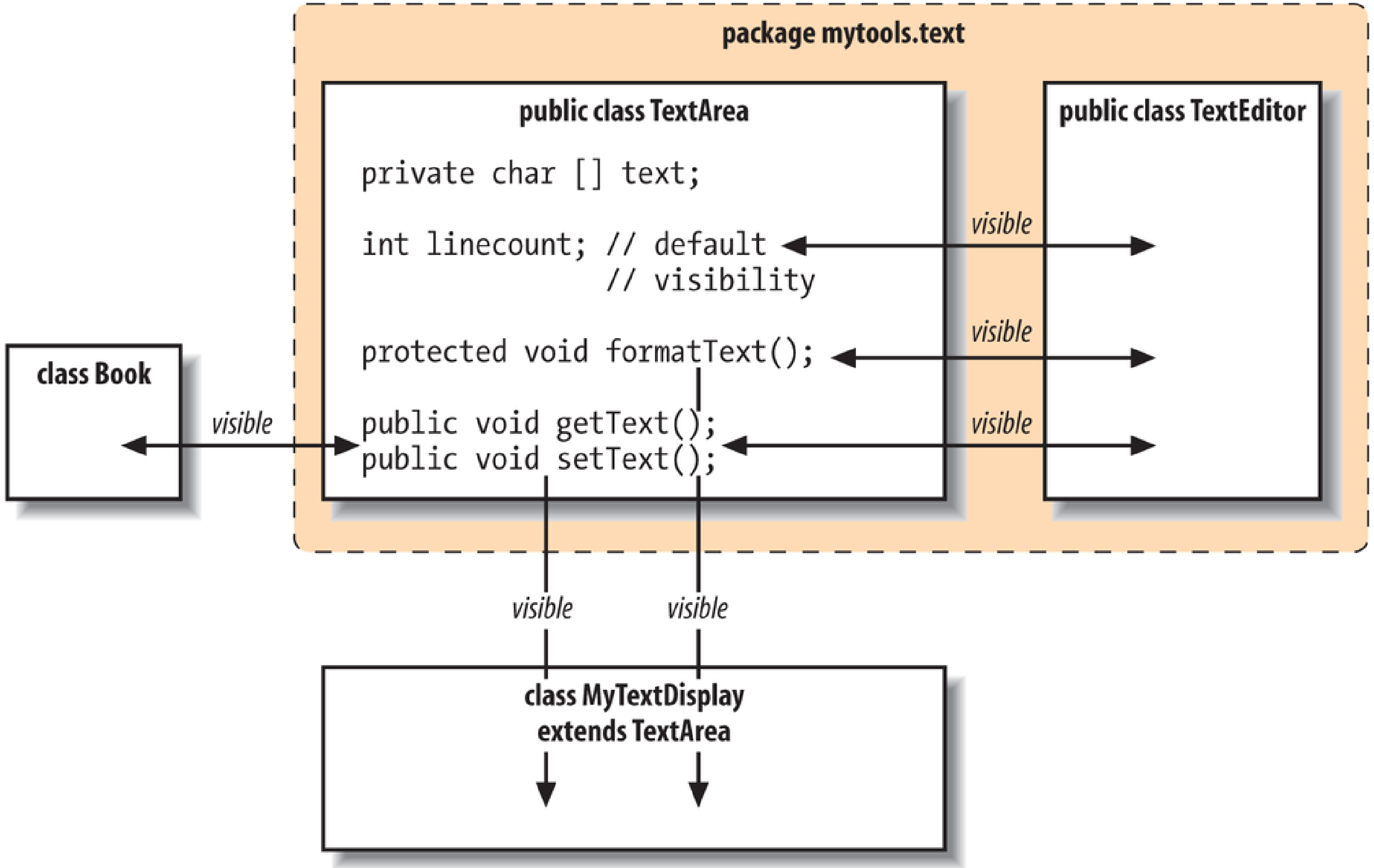
ENCAPSULATION
BOUNDARIES

ReD

Any method call or attribute accesses that that is not in the same class (or package) **crosses** an encapsulation boundary

You want to **minimize** these accesses - that's what we mean by "ReD"

So… expose (i.e. make public) the methods/attributes that client code really needs, and hide everything else



ReD: reducing dependency

THE
SINGLE RESPONSIBILITY PRINCIPLE (SRP)

> *A class should have one, and only one, reason to change.*
> *-- Robert C. Martin*

Each class should have **one responsibility**

- it **shouldn't** take on extra responsibilities
- ideally it should contain **all functionality needed** to support that responsibility

THE
OPEN/CLOSED PRINCIPLE (OCP)

> *Software entities (classes, modules, functions, etc.) should be*
> ***open for extension, but closed for modification.***
> *-- Robert C. Martin*

Initially, sounds contradictory

- don't you have to modify something to extend it?

Martin is talking about what should be easy and what should be hard when you're **adding functionality** to your software

FORMAL DEFINITION OF THE
LISKOV SUBSTITUTION PRINCIPLE (LSP)

If **B** is a subclass of **A**, you should be able to **put a B** in anywhere the program expects an **A**

- so, for example: **A myA = new B();**

This is true even if A is an **abstract class or interface**

The Java compiler knows that **all methods in A exist in B** too

- so there's nothing you can do with an A that the B won't support
- no reason not to allow **B to act in place of A**

THE
INTERFACE SEGREGATION PRINCIPLE (ISP)

> *Clients should not be forced to depend upon interfaces that they do not use.*
> *-- Robert C. Martin*

This seems obvious, but is surprisingly hard to do in practice

- your abstractions start out nice and clean, but it is hard to keep them that way over time

FIXING INTERFACE POLLUTION

Can fix this by **segregating interfaces**

- one for **basic** calculations
- one for **advanced** calculations

```
public interface BasicCalc {
        public double add();
        public double subtract();
        public double multiply();
        public double divide();
}

public interface AdvCalc {
        public double sin();
        public double cos();
        public double tan();
        public double log();
        public double sqrt();
}
```

The primary school version can implement **BasicCalc**

The advanced version can implement both: **BasicCalc** and **AdvCalc**

THE
DEPENDENCY INVERSION PRINCIPLE (DIP)

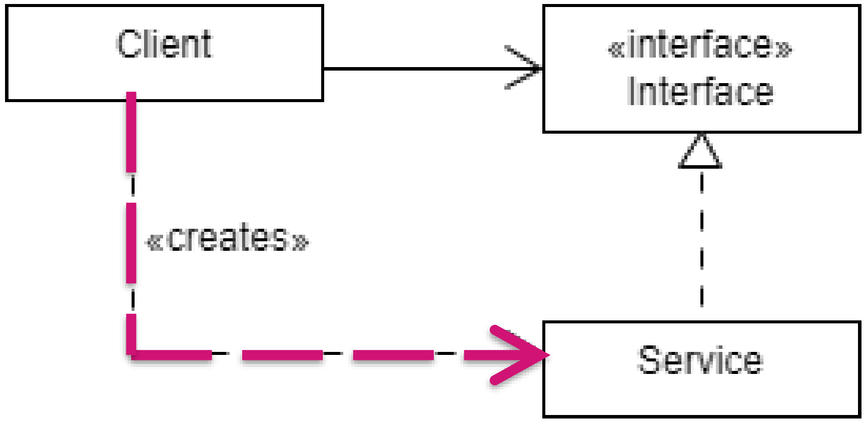
> *High-level modules should not depend on low-level modules. Both should depend on abstractions.*
>
> *Abstractions should not depend on details. Details should depend on abstractions.*
> *-- Robert C. Martin*

This is an "**inversion**" because if you are doing **top-down design**, you often end up with a high level module that calls methods in (i.e. depends on) low-level modules

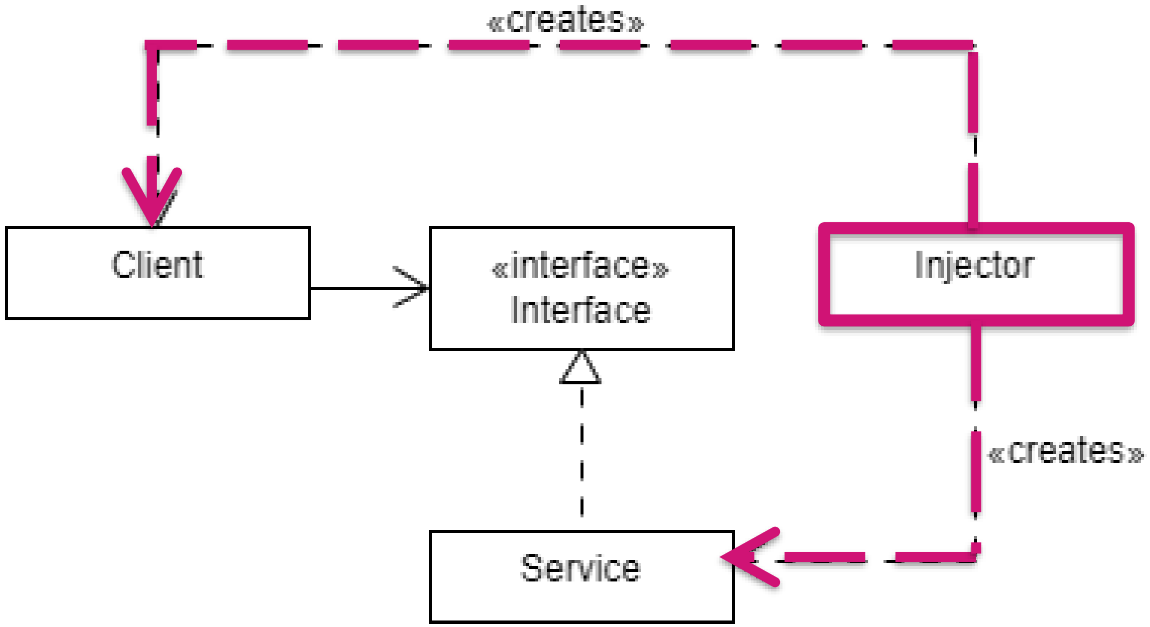
NOTE: We will cover this briefly here, and return to this principle in later lectures on abstraction

COMPARISON OF
CLASS DIAGRAMS

Dependency ***inversion***



Dependency ***injection***



TYPES OF
DEPENDENCY INJECTION

Constructor injection: an instance of the service is passed into the client's constructor

- the injector must be the class that instantiates the client

Setter injection: the client has a concrete setter that the injector can use to pass in the service instance

- can be used at any time; allows you to change the service of a running client
- but requires a public setter, might not be good for information hiding

Interface injection: the client implements an interface that allows the injector to pass in the service instance

- ends up being like setter injection but you can choose what your setter is called/are not restricted to a single setter

Summary

DRY: Don’t repeat yourself

Classes should be responsible for their own properties

Avoid excessive use of literals

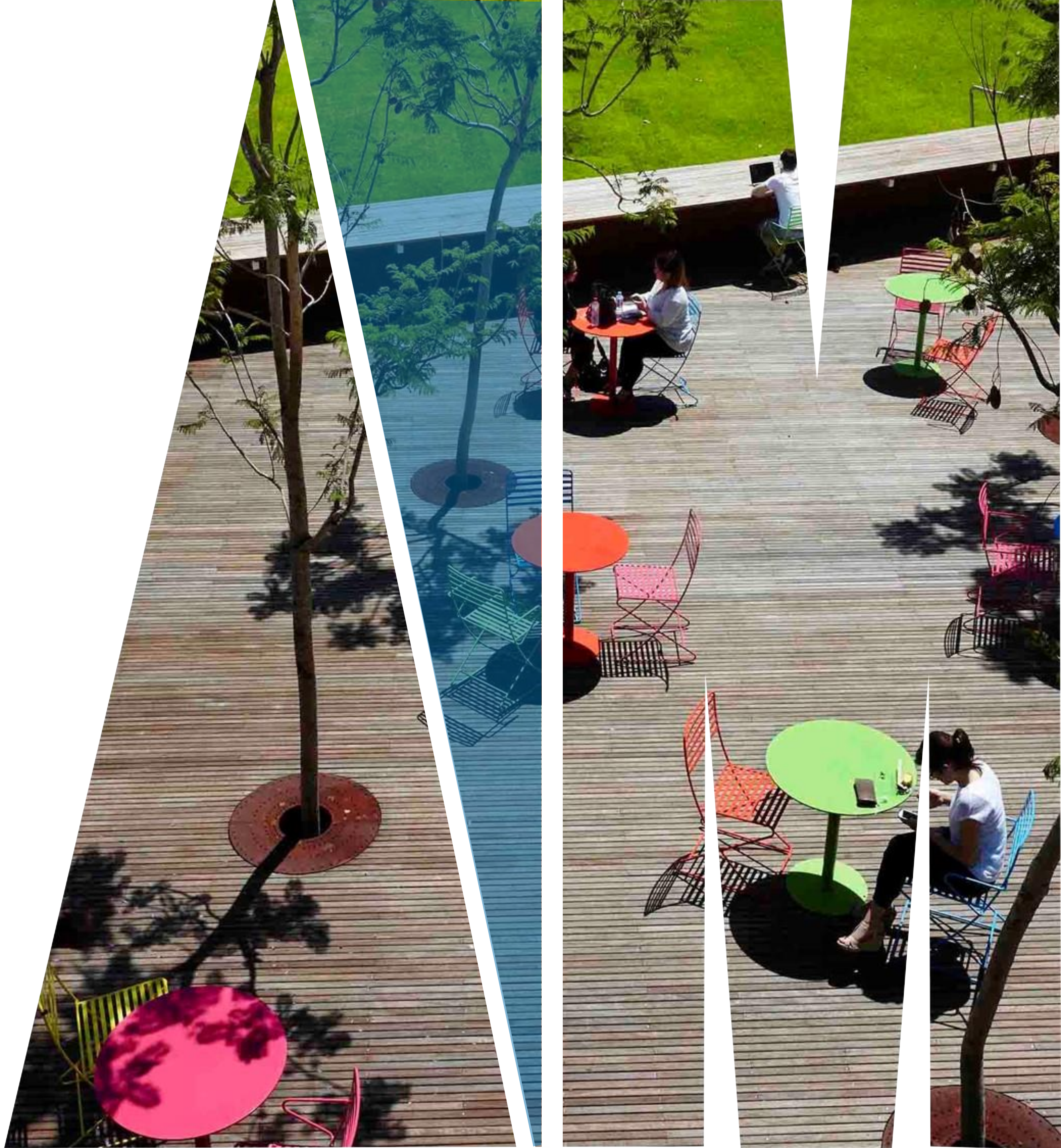
ReD: Reduce Dependencies

SOLID PRINCIPLES:

- ❑ SRP: Single Responsibility Principle
- ❑ OCP: Open/Close Principle
- ❑ LSP: Liskov Substitution Principle
- ❑ ISP: Interface Segregation Principle
- ❑ DIP: Dependency Inversion Principle

MONASH University

Thanks



MONASH University