COMP2511

WEEK 4

Would you rather give up bread or rice?

IMPORTANT NOTICE

- Assignment-i is due NEXT WEEK!
- Assignment-ii pairs will be finalised by tomorrow. Make sure you have filled out the form.

ASSIGNMENT TIPS

- Use .equal() for String (or Class) instead of ==
- Avoid magic numbers, use final variables
- Use the super constructor to set variables
- Use instanceof for type comparison
- Polymorphism is preferred over type checking to perform a specific action

ASSIGNMENT TIPS

```java
if (s.getClass().equals(Rectangle.class)) { // v1: kinda ok
  // do something only on exactly the Rectangle class
}

if (s.getType().equals("Rectangle")) { // v2: very bad
  // do something on all rectangles
}

if (s instanceof Rectangle) { // v3: good
  // do something on all rectangles
}
```

ASSIGNMENT TIPS

```java
// Example: What not to do
public abstract class Shape {
    public abstract String getType();
}

public class Rectangle extends Shape {}

public class Square extends Rectangle {
    public static void main(String[] args) {
        List<Shape> shapes = new ArrayList<>();
        shapes.add(new Rectangle());
        shapes.add(new Square());
        for (Shape s : shapes) {
            if (s.getType().equals("Rectangle")) {
                // calculate the area this way
            } else if (s.getType().equals("Square")) {
                // calculate the area a different way
            }
        }
    }
}
```

```java
// Example: What to do
public abstract class Shape {
    public abstract String getType();
    public abstract double area();
    // ^ declare method in superclass
}

public class Rectangle extends Shape {
    public double area() {
        // calculate the area this way
    }
}

public class Square extends Rectangle {
    public double area() {
        // calculate the area a different way
    }
    public static void main(String[] args) {
        List<Shape> shapes = new ArrayList<>();
        shapes.add(new Rectangle());
        shapes.add(new Square());
        for (Shape s : shapes) {
            s.area(); // no more type checking
        }
    }
}
```

AGENDA

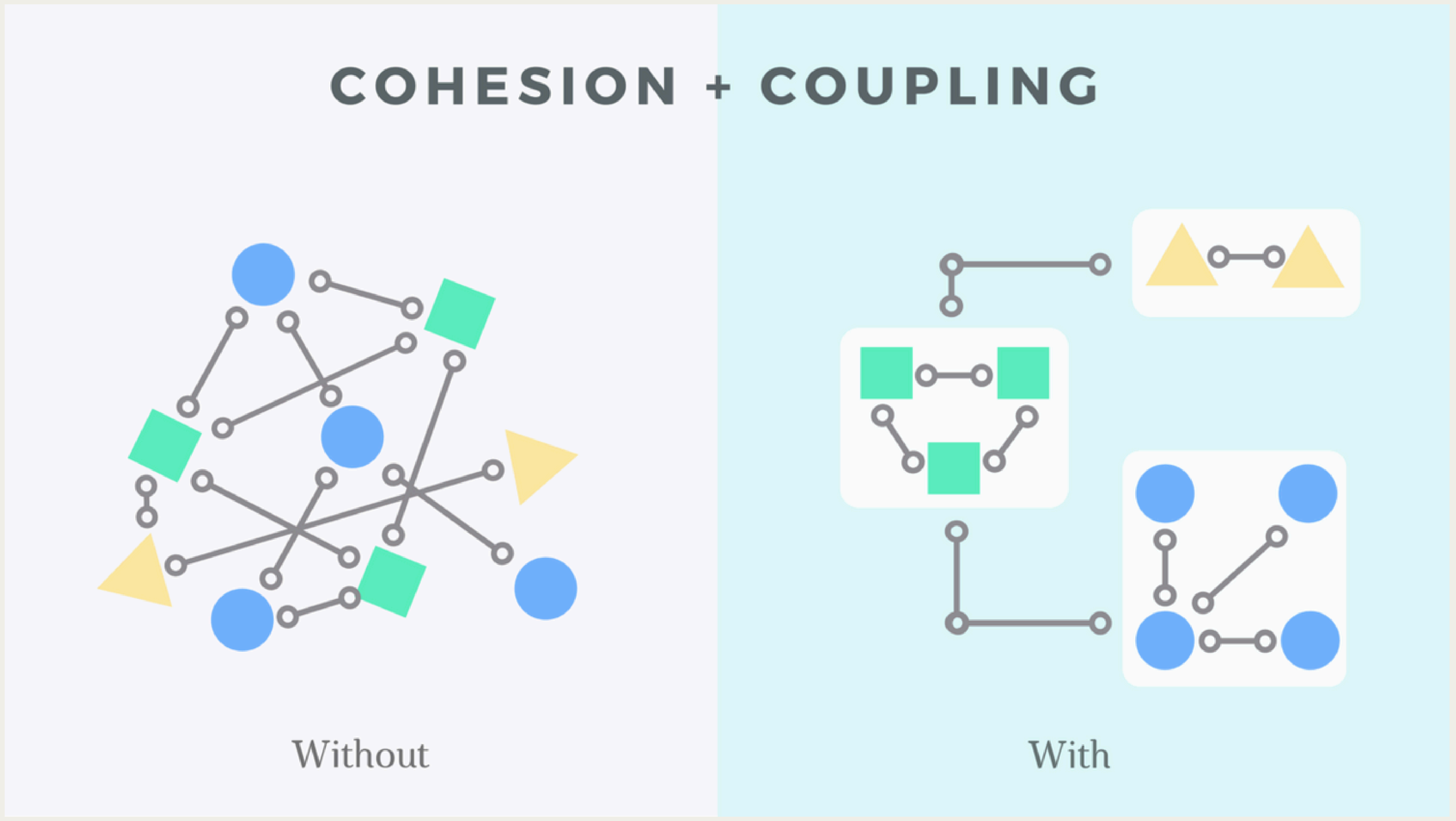
- Design Principles
- Steams and Lambdas
- Design by Contract

Law of Demeter

“Principle of least knowledge”

LAW OF DEMETER

Law of Demeter (aka principle of least knowledge) is a **design guideline** that says that an **object** should **assume as little as possible knowledge** about the structures or properties of other objects.



LAW OF DEMETER

A method in an object should only invoke methods of:

- The object itself
- The object passed in as a parameter to the method
- Objects instantiated within the method
- Any component objects
- And not those of objects returned by a method

E.g., don't do this

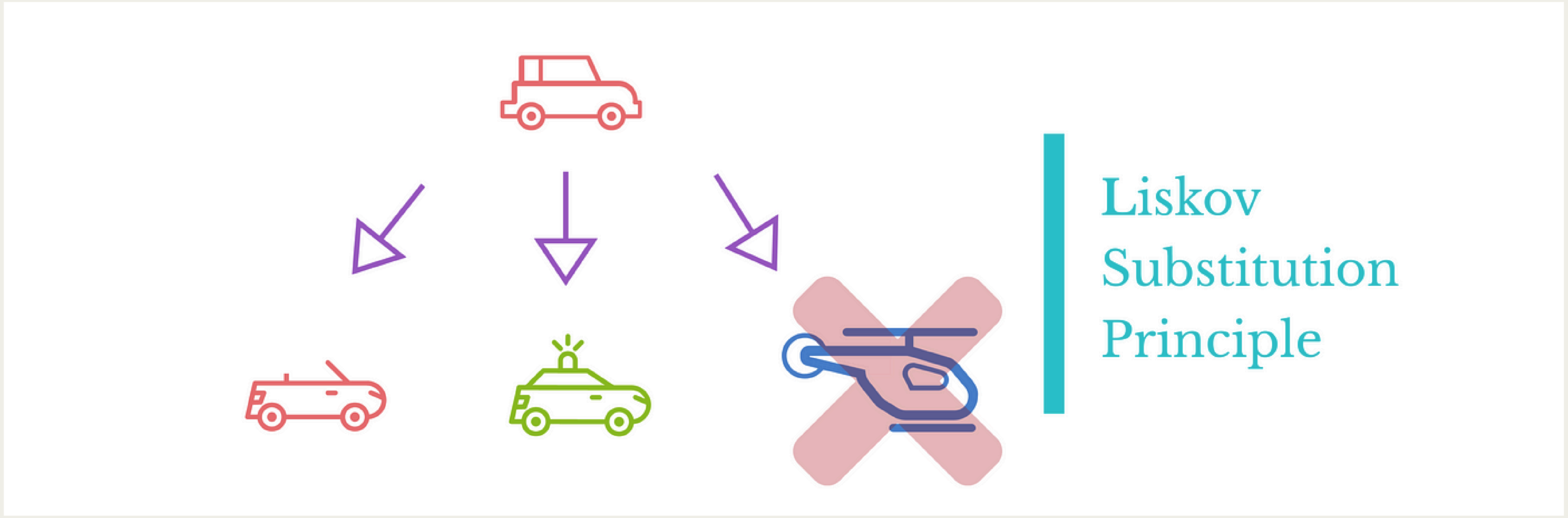
```
object.get(name).get(thing).remove(node)
```

*Caveat is that sometimes this is unavoidable

LISKOV SUBSTITUTION PRINCIPLE

Liskov Substitution Principle (LSP) states that objects of a **superclass** should be **replaceable** with objects of its **subclasses without breaking the application**.

*inheritance arrows are the other way around



LISKOV SUBSTITUTION PRINCIPLE

Solve the problem without inheritance

- Delegation - delegate the functionality to another class
- Composition - reuse behaviour using one or more classes with composition

Design principle: Favour composition over inheritance.

If you favour composition over inheritance, your software will be more flexible, easier to maintain, extend.

Streams

via Code Example

STREAMS

Common uses of streams are:

- forEach
- filter
- map
- reduce

Sort of similar to the Array prototypes/methods in JavaScript

Design By Contract

What is it?

DESIGN BY CONTRACT

At the design time, responsibilities are clearly assigned to different software elements, clearly documented and enforced during the development and using unit testing and/or language support.

- Clear demarcation of responsibilities helps prevent redundant checks, resulting in simpler code and easier maintenance
- Crashes if the required conditions are not satisfied. May not be suitable for highly availability applications

DESIGN BY CONTRACT

Every software element should define a specification (or a contract) that govern its transaction with the rest of the software components.
A contract should address the following 3 conditions:

1. Pre-condition - what does the contract expect?
2. Post-condition - what does that contract guarantee?
3. Invariant - What does the contract maintain?

DESIGN BY CONTRACT - QUESTIONS

1. Discuss briefly as a class how you have used Design by Contract already in previous courses.
2. Discuss how Design By Contract was applied in the Blackout assignment.
3. Will you need to write unit tests for something that doesn't meet the preconditions? Explain why.

DESIGN BY CONTRACT - PRECONDITION WEAKING

- An implementation or redefinition (method overriding) of an inherited method must comply with the inherited contract for the method
- Preconditions may be weakened (relaxed) in a subclass, but it must comply with the inherited contract
- An implementation or redefinition may lesson the obligation of the client, but not increase it

from 0 <= theta <= 90 to 0 <= theta <= 180 is weakening

[0, 90] => [0, 180]

Why?

LSP. I should be able to use the subclass's implementation in place of my super class.

LABORATORY

MARKING SESSION