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 today (and kinda over the weekend). Make sure you have filled out the form.

IMPORTANT NOTICE

F15B_ALBATROSS	Vinay Arkalgud	Ryan Saab
F15B_BLUEBIRD	Marcus Chan	Dabin Liu
F15B_CROW	Oscar Deng	Avhinab Koirala
F15B_DOVE	Marco Gava	Jatin Kumar Sharma
F15B_EAGLE	Ali Haidar	Zhendong Tang
F15B_FALCON	Justin Han	Varun Penumalli
F15B_GALAH	Andrew Hu	Jason Li
F15B_HERON	Lachlan Kwok	Alicia Tan
F15B_IBIS	Duncan Lai	Connor Li
F15B_JACKDAW	Ethan Loi	Tony Wong
F15B_KINGFISHER	Harshith Narayanarao	Ruiying Ren
F15B_LORIKEET	Jai Seth	Smit Shah

WHICH ONE?

```
void solve() {
    int n;
    cin >> n;

    int sum = 0;
    for (int i = 0; i < n; ++i)
{       int num;
        cin >> num;
        if (i % 2) sum -= num;
        else sum += num;
    }

    cout << sum << '\n';
}
```


```
void solve() {
    int n;
    cin >> n;

    int sum = 0;
    for (int i = 0; i < n; ++i)
{       int num;
        cin >> num;
        if (i % 2) sum -= num;
        else sum += num;
    }

    cout << sum << '\n';
}
```

```
void solve() {
    int n;
    cin >> n;

    int sum = 0;
    for (int i = 0; i < n; ++i)
{       int num;
        cin >> num;
        if (i % 2) sum -= num;
        else sum += num;
    }

    cout << sum << '\n';
}
```

WHICH ONE?

```
void solve() {
    int n;
    cin >> n;

    int sum = 0;
    for (int i = 0; i < n; ++i)
{       int num;
        cin >> num;
        if (i % 2) sum -= num;
        else sum += num;
    }

    cout << sum << '\n';
}
```

```
void solve() {
    int n;
    cin >> n;

    int sum = 0;
    for (int i = 0; i < n; ++i)
{       int num;
        cin >> num;
        if (i % 2) sum -= num;
        else sum += num;
    }

    cout << sum << '\n';
}
```

```
void solve() {
    int n;
    cin >> n;

    int sum = 0;
    for (int i = 0; i < n; ++i)
{       int num;
        cin >> num;
        if (i % 2) sum -= num;
        else sum += num;
    }

    cout << sum << '\n';
}
```

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

```
if (s.getClass().equals(Imposter.class)) { // v1: kinda ok
    // do something only on exactly the Imposter class
}
if (s.getType().equals("Imposter")) { // v2: very bad
    // do something on all imposters
}
if (s instanceof Imposter) { // v3: good
    // do something on all imposters
}
```

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
        }
    }
}
```

A G E N D A

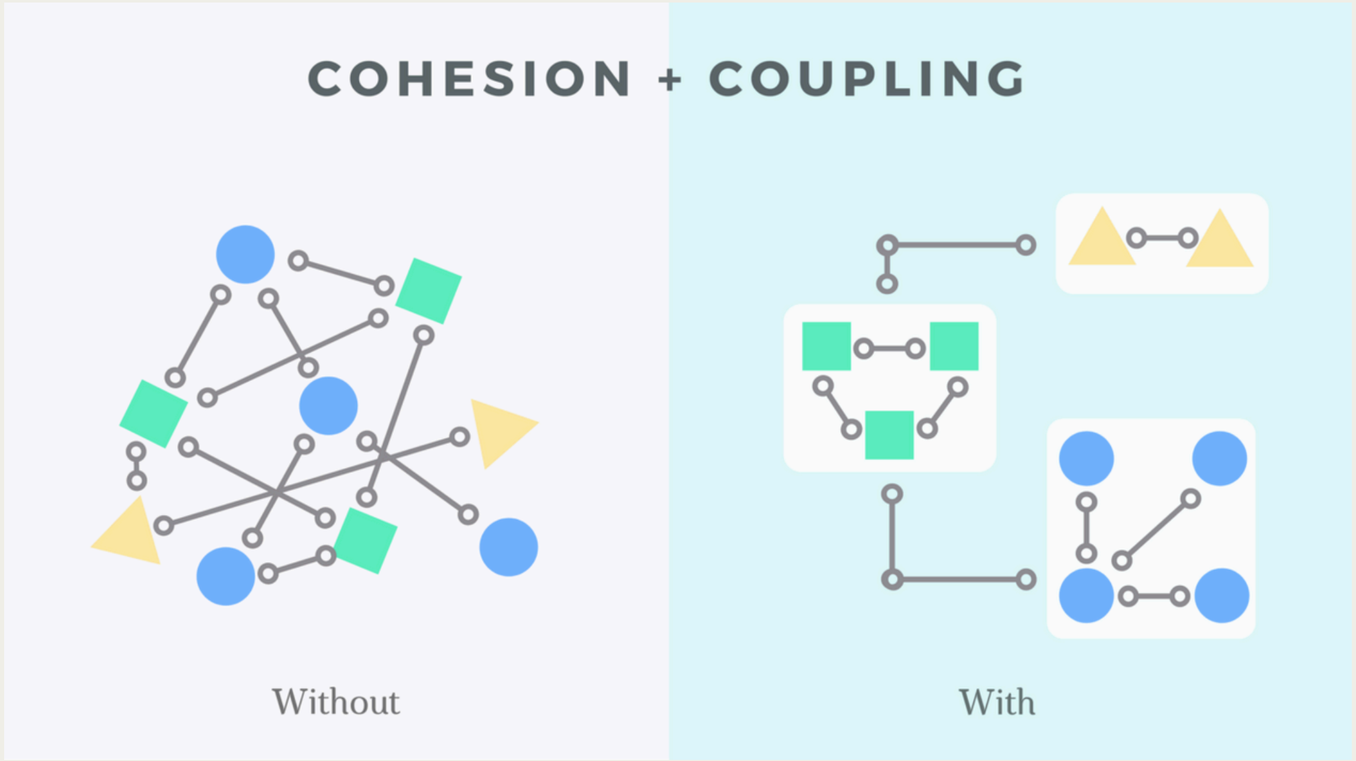
- Design Principles
- Steams and Lambdas
- Design by Contract

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

LAW OF DEMETER

In the **unsw.training** package there is some skeleton code for a training system.

- Every employee must attend a whole day training seminar run by a qualified trainer
- Each trainer is running multiple seminars with no more than 10 attendees per seminar

In the **TrainingSystem** class there is a method to book a seminar for an employee given the dates on which they are available. This method violates the principle of least knowledge (Law of Demeter).

1. How and why does it violate this principle?
2. In violating this principle, what other properties of this design are not desirable?
3. Refactor the code so that the principle is no longer violated. How has this affected other properties of the design?
4. More generally, are getters essentially a means of violating the principle of least knowledge? Does this make using getters bad design?

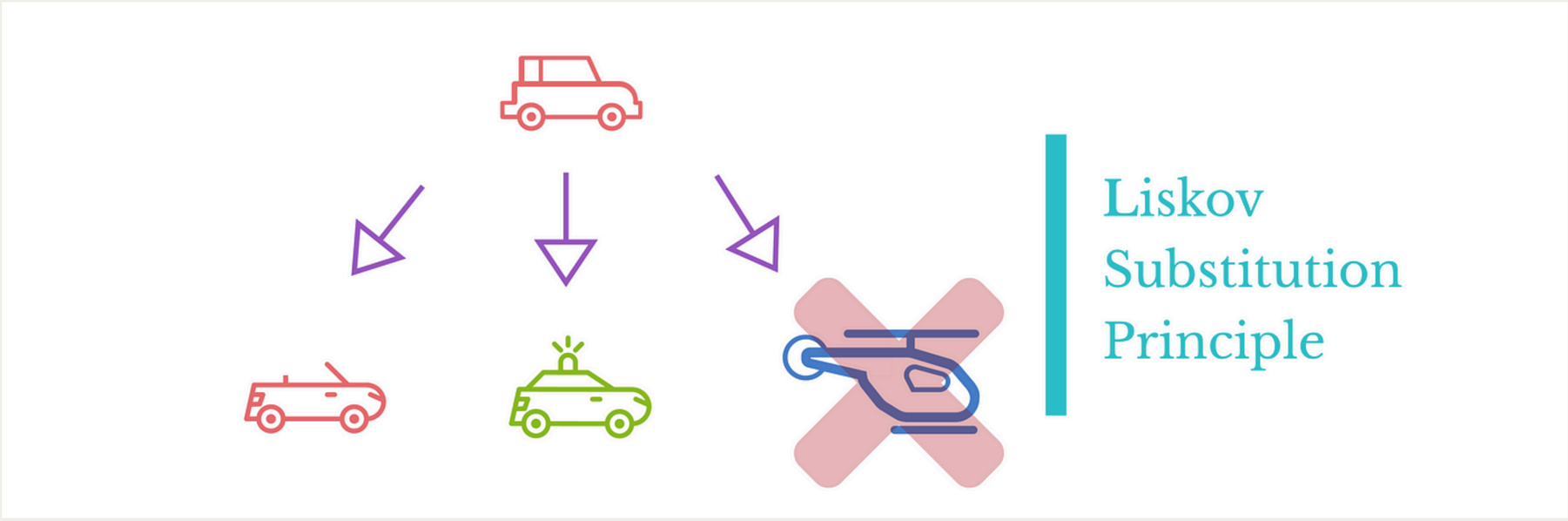
LAW OF DEMETER

- Getters which pass objects (not primitives) let clients use its methods
  - Violating principle of least knowledge
  - However, makes classes more reusable
  - What if it doesn’t have that functionality? Then you are using it to extend functionality. So good?
- Another example: having **getAttendees** in the seminar class.
  - Any client is able to modify this to have more than 10 attendees (everything is an object and passed by reference)
  - Unfortunately Java offers no good solutions to this problem
    - **getAttendees** has to create a copy of the list or
      - Needlessly copies data
    - Use **Collections.unmodifiableList(...)**
      - Still has the **add(...)** method but using it causes an exception
  - Other languages resolve this problem by having proper immutable or read-only lists.

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.

LISKOV SUBSTITUTION PRINCIPLE

Look at the **OnlineSeminar** class. How does this violate the Liskov Substitution Principle?

LISKOV SUBSTITUTION PRINCIPLE

Look at the **OnlineSeminar** class. How does this violate the Liskov Substitution Principle?

- **OnlineSeminar** does not require a list of attendees
- Would expect it to be “booked” like a **Seminar**
- Has “Is-A” relationship but doesn’t make sense here
    - But invalid inheritance once you take into account what the classes do and represent.

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. In the **Calculator** code, specify a contract for each of the functions. Hint: for the trig functions, look at the interface of the Math functions in the Java documentation. Key edge cases to consider:
    a. Dividing by zero
    b. tan(Math.PI / 2)
4. 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