# COMP2511

23T1 Week 4

WEDNESDAY 1PM - 4PM (W13B)

FRIDAY 11AM - 2PM (F11A)

Slides by Alvin Cherk (z5311001)

# Today

- The Functional Paradigm
- Refactoring
- **Introduction to Design Patterns**
- Strategy Pattern
- State Pattern
- Observer Pattern

- **Design Principles**
- **Design by Contract**
- **Streams & Lambdas**

# Law of Demeter

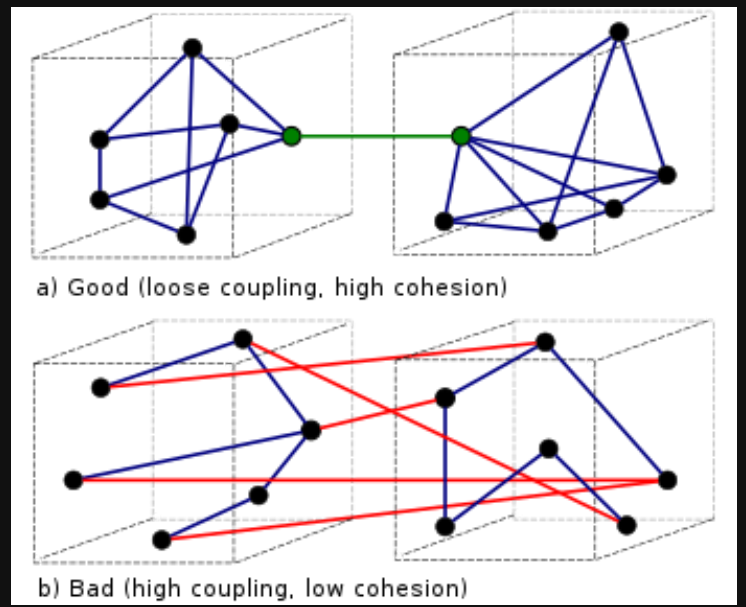
"Principle of least knowledge"

# Law of Demeter

What is it?

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.

It aims to achieve loose coupling in code.



# Law of Demeter

What does it actually mean?

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

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

*Caveat is that sometimes this is unavoidable

# Code Review

Law of Demeter

# Code Review

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).

```java
public class TrainingSystem {
    private List<Trainer> trainers;

    public LocalDate bookTraining(String employee, List<LocalDate> availability) {
        for (Trainer trainer : trainers) {
            for (Seminar seminar : trainer.getSeminars()) {
                for (LocalDate available : availability) {
                    if (seminar.getStart().equals(available) &&
                            seminar.getAttendees().size() < 10) {
                        seminar.getAttendees().add(employee);
                        return available;
                    }
                }
            }
        }
        return null;
    }
}
```

```java
/**
 * A trainer that runs in person seminars.
 */
public class Trainer {
    private String name;
    private String room;
    private List<Seminar> seminars;

    public List<Seminar> getSeminars() {
        return seminars;
    }
}
```

```java
/**
 * An in person all day seminar with a maximum of 10 attendees.
 */
public class Seminar {
    private LocalDate start;
    private List<String> attendees;

    public LocalDate getStart() {
        return start;
    }

    public List<String> getAttendees() {
        return attendees;
    }
}
```

```java
/**
 * An online seminar is a video that can be viewed at any time by employees. A
 * record is kept of which employees have watched the seminar.
 */
public class OnlineSeminar extends Seminar {
    private String videoURL;
    private List<String> watched;
}
```

How and why does it violate this principle?

What other properties of this design are not desirable?

```java
public class TrainingSystem {
    public List<Trainer> trainers;
    /**
     * Try to booking training for an employee, given their availability.
     *
     * @param employee
     * @param availability
     * @return The date of their seminar if booking was successful, null there
     * are no empty slots in seminars on the day they are available.
     */
    public LocalDate bookTraining(String employee, List<LocalDate> availability) {
        for (Trainer trainer : trainers) {
            LocalDate booked = trainer.book(employee, availability);
            if (booked != null)
                return booked;
        }
        return null;
    }
}
```

```java
/**
 * An in person all day seminar with a maximum of 10 attendees.
 */
public class Seminar {
    private LocalDate start;
    private List<String> attendees;
    public LocalDate getStart() {
        return start;
    }

    /**
     * Try to book this seminar if it occurs on one of the available days and
     * isn't already full
     * @param employee
     * @param availability
     * @return The date of the seminar if booking was successful, null otherwise
     */
    public LocalDate book(String employee, List<LocalDate> availability) {
        for (LocalDate available : availability) {
            if (start.equals(available) &&
                    attendees.size() < 10) {
                attendees.add(employee);
                return available;
            }
        }
        return null;
    }
}
```

```java
/**
 * A trainer that runs in person seminars.
 */
public class Trainer {
    private String name;
    private String room;
    private List<Seminar> seminars;

    public List<Seminar> getSeminars() {
        return seminars;
    }

    /**
     * Try to book one of this trainer's seminars.
     * @param employee
     * @param availability
     * @return The date of the seminar if booking was successful, null if the
     * trainer has no free slots in seminars on the available days.
     */
    public LocalDate book(String employee, List<LocalDate> availability) {
        for (Seminar seminar : seminars) {
            LocalDate booked = seminar.book(employee, availability);
            if (booked != null)
                return booked;
        }
        return null;
    }
}
```

```java
/**
 * An online seminar is a video that can be viewed at any time by
employees. A
 * record is kept of which employees have watched the seminar.
 */
public class OnlineSeminar extends Seminar {
    private String videoURL;
    private List<String> watched;
}
```

1. **`TrainingSystem`** no longer has knowledge of **`Seminar`**
2. Each class has their own responsibility (good cohesion)

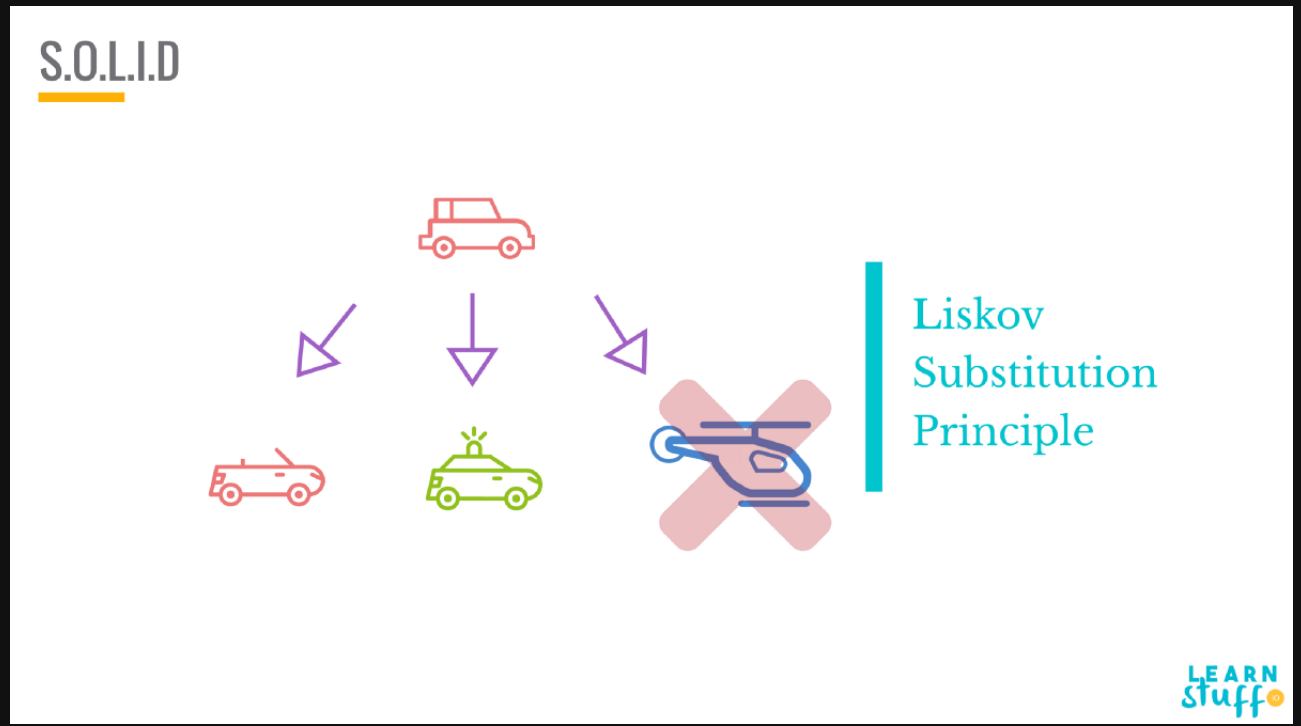
# Liskov Substitution Principle

# Liskov Substitution Principle

What is it?

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

```java
/**
 * An in person all day seminar with a maximum of 10 attendees.
 */
public class Seminar {
    private LocalDate start;
    private List<String> attendees;

    public LocalDate getStart() {
        return start;
    }

    public List<String> getAttendees() {
        return attendees;
    }
}
```

```java
/**
 * An online seminar is a video that can be viewed at any time by employees. A
 * record is kept of which employees have watched the seminar.
 */
public class OnlineSeminar extends Seminar {
    private String videoURL;
    private List<String> watched;
}
```

Where does **`OnlineSeminar`** violate LSP?

**`OnlineSeminar`** doesn't require a list of attendees

# Streams

# Streams

Streams abstract away the details of data structures and allows you to access all the values in the data structure through a **common interface**

```
1 List<String> strings = new ArrayList<String>(Arrays.asList(new String[] {"1", "2", "3", "4", "5"}));
2 for (String string : strings) {
3    System.out.println(string);
4 }
```

```
1 List<String> strings = new ArrayList<String>(Arrays.asList(new String[] {"1", "2", "3", "4", "5"}));
2 strings.stream().forEach(x -> System.out.println(x));
```

```
1 Map<String, Integer> map = new HashMap<>();
2 map.put("One", 1);
3 map.put("Two", 2);
4 map.put("Three", 3);
5 map.entrySet().stream().forEach(x -> System.out.printf("%s, %s\n", x.getKey(), x.getValue()));
```

# Streams

Common uses of streams are:

- forEach
- filter
- map
- reduce

Sort of similar to the Array prototypes/methods in JavaScript

# Optional<type>

If a variable can be null, use **`Optional<>`**.

Try to never set variables to be null, as you can get **`NullPointerExceptions`** which aren't fun to deal with

```
1  public class OptionalExample {
2      public static void main(String[] args) {
3          List<String> strings = new ArrayList<String>(Arrays.asList(new String[] { "1", "2", "3", "4", "5" }));
4          Optional<String> res = strings.stream().filter(x -> x.equals("1")).findAny();
5
6          if (res.isPresent()) {
7              System.out.println(res.get());
8          } else {
9              // It doesn't exist
10             // Handle error?
11         }
12     }
13 }
```

**`.findAny()`** actually returns **`Optional<type>`**

# Code Demo

Streams

# Code Demo

Convert the following to use streams

```
1  package stream;
2
3  import java.util.ArrayList;
4  import java.util.Arrays;
5  import java.util.List;
6  import java.util.stream.Collectors;
7
8  public class App {
9      public static void main(String[] args) {
10         List<String> strings = new ArrayList<String>(Arrays.asList(new String[] { "1", "2", "3", "4", "5" }));
11         for (String string : strings) {
12             System.out.println(string);
13         }
14
15         List<String> strings2 = new ArrayList<String>(Arrays.asList(new String[] { "1", "2", "3", "4", "5" }));
16         List<Integer> ints = new ArrayList<Integer>();
17         for (String string : strings2) {
18             ints.add(Integer.parseInt(string));
19         }
20         System.out.println(ints);
21     }
22
23 }
```

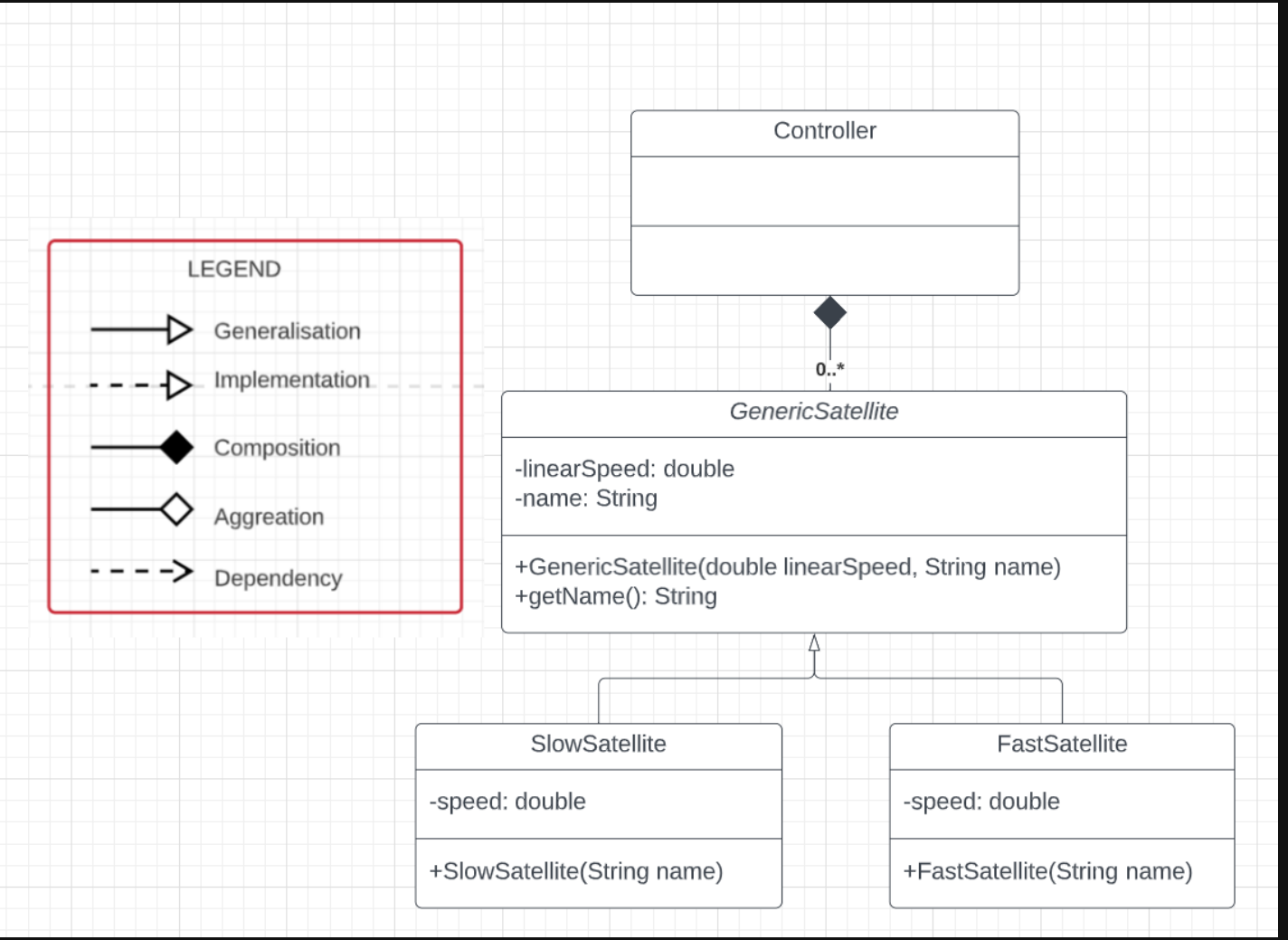
```java
package stream;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class App {
    public static void main(String[] args) {
        List<String> strings = new ArrayList<String>(Arrays.asList(new String[] { "1", "2", "3", "4", "5" }));
        // Same thing
        strings.stream().forEach(x -> System.out.println(x));
        // Use if there is more than one line of code needed in lambda
        strings.stream().forEach(x -> {
            System.out.println(x);
        });

        List<String> strings2 = new ArrayList<String>(Arrays.asList(new String[] { "1", "2", "3", "4", "5" }));
        List<Integer> parsedStrings = strings2.stream().map(x -> Integer.parseInt(x)).collect(Collectors.toList());
        strings2.stream().map(x -> Integer.parseInt(x)).forEach(x -> System.out.println(x));
    }
}
```

# Further Streams Example



```java
package stream;

public abstract class GenericSatellite {
    private double linearSpeed; // in KM/minute
    private String name; // name of satellite

    public GenericSatellite(double linearSpeed, String name) {
        this.linearSpeed = linearSpeed;
        this.name = name;
    }

    public String getName() {
        return this.name;
    }

    public double getLinearSpeed() {
        return this.linearSpeed;
    }

    @Override
    public String toString() {
        return "{" +
                "name='" + getName() + "'" +
                "}";
    }

}
```

```java
package stream;

public class FastSatellite extends GenericSatellite {
    private static final double speed = 100.0;

    public FastSatellite(String name) {
        super(speed, name);
    }
}
```

```java
package stream;

public class SlowSatellite extends GenericSatellite {
    private static final double speed = 50.0;

    public SlowSatellite(String name) {
        super(SlowSatellite.speed, name);
    }
}
```

```java
package stream;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

public class Controller {
    private List<GenericSatellite> satellites = new ArrayList<>();

    public void addSatellite(GenericSatellite satelliteToAdd) {
        this.satellites.add(satelliteToAdd);
    }

    public List<GenericSatellite> getSatelliteList() {
        return this.satellites;
    }

    public static void main(String[] args) { ... }

}
```

```java
public static void main(String[] args) {
    Controller c = new Controller();
    c.addSatellite(new FastSatellite("Fast Satellite 1"));
    c.addSatellite(new FastSatellite("Fast Satellite 2"));
    c.addSatellite(new SlowSatellite("Slow Satellite 1"));

    List<GenericSatellite> allSatellites = c.getSatelliteList();
    System.out.println("All: " + allSatellites);
    // [{name='Fast Satellite 1'}, {name='Fast Satellite 2'}, {name='Slow Satellite 1'}]

    // What if I just want FastSatellites only?
    // Method 1, normal for-in/for-each
    List<GenericSatellite> fast1 = new ArrayList<>();
    for (GenericSatellite x : allSatellites) {
        if (x instanceof FastSatellite) {
            fast1.add(x);
        }
    }
    System.out.println("Just fast: " + fast1);
    // [{name='Fast Satellite 1'}, {name='Fast Satellite 2'}]

    // Method 2, streams
    List<GenericSatellite> fast2 = allSatellites.stream().filter(x -> x instanceof FastSatellite).collect(Collectors.toList());
    System.out.println("Just fast: " + fast2);
    // [{name='Fast Satellite 1'}, {name='Fast Satellite 2'}]

    // Same, but I typecast at the same time
    List<FastSatellite> fast3 = allSatellites.stream().filter(x -> x instanceof FastSatellite).map(x -> (FastSatellite) x)
            .collect(Collectors.toList());
    System.out.println("Just fast: " + fast3);
    // [{name='Fast Satellite 1'}, {name='Fast Satellite 2'}]
}
```

```java
public static void main(String[] args) {
    Controller c = new Controller();
    c.addSatellite(new FastSatellite("Fast Satellite 1"));
    c.addSatellite(new FastSatellite("Fast Satellite 2"));
    c.addSatellite(new SlowSatellite("Slow Satellite 1"));

    List<GenericSatellite> allSatellites = c.getSatelliteList();
    System.out.println("All: " + allSatellites);
    // [{name='Fast Satellite 1'}, {name='Fast Satellite 2'}, {name='Slow Satellite 1'}]

    // What if im trying to search for something?
    // Look for satellite with name == "Fast Satellite 2"
    // Method 1, normal for-in/for-each
    GenericSatellite g1 = null;

    for (GenericSatellite x : allSatellites) {
        if (x.getName().equals("Fast Satellite 2")) {
            g1 = x;
            break;
        }
    }
    System.out.println(g1);
    // {name='Fast Satellite 2'}

    // Method 2, streams
    GenericSatellite g2 = allSatellites.stream().filter(x -> x.getName().equals("Fast Satellite 2")).findFirst().orElse(null);
    System.out.println(g2);
    // {name='Fast Satellite 2'}

    // Now I search for "Fast Satellite 3", which doesn't exist
    GenericSatellite g3 = allSatellites.stream().filter(x -> x.getName().equals("Fast Satellite 3")).findFirst().orElse(null);
    System.out.println(g3);
    // null
}
```

# Design By Contract

# 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

```
1  public class Calculator {
2      public static Double add(Double a, Double b) {
3          return a + b;
4      }
5
6      public static Double subtract(Double a, Double b) {
7          return a - b;
8      }
9
10     public static Double multiply(Double a, Double b) {
11         return a * b;
12     }
13
14     public static Double divide(Double a, Double b) {
15         return a / b;
16     }
17
18     public static Double sin(Double angle) {
19         return Math.sin(angle);
20     }
21
22     public static Double cos(Double angle) {
23         return Math.cos(angle);
24     }
25
26     public static Double tan(Double angle) {
27         return Math.tan(angle);
28     }
29 }
```

```java
public class Calculator {
    /**
     * @preconditions a, b != null
     * @postconditions a + b
     */
    public static Double add(Double a, Double b) {
        return a + b;
    }

    /**
     * @preconditions a, b != null
     * @postconditions a - b
     */
    public static Double subtract(Double a, Double b) {
        return a - b;
    }

    /**
     * @preconditions a, b != null
     * @postconditions a * b
     */
    public static Double multiply(Double a, Double b) {
        return a * b;
    }

    /**
     * @preconditions a, b != null, b != 0
     * @postconditions a / b
     */
    public static Double divide(Double a, Double b) {
        return a / b;
    }

    /**
     * @preconditions angle != null
     * @postconditions sin(angle)
     */
    public static Double sin(Double angle) {
        return Math.sin(angle);
    }

    /**
     * @preconditions angle != null
     * @postconditions cos(angle)
     */
    public static Double cos(Double angle) {
        return Math.cos(angle);
    }

    /**
     * @preconditions angle != null, angle != Math.PI / 2
     * @postconditions tan(angle)
     */
    public static Double tan(Double angle) {
        return Math.tan(angle);
    }
}
```

# Liskov Substitution Principle

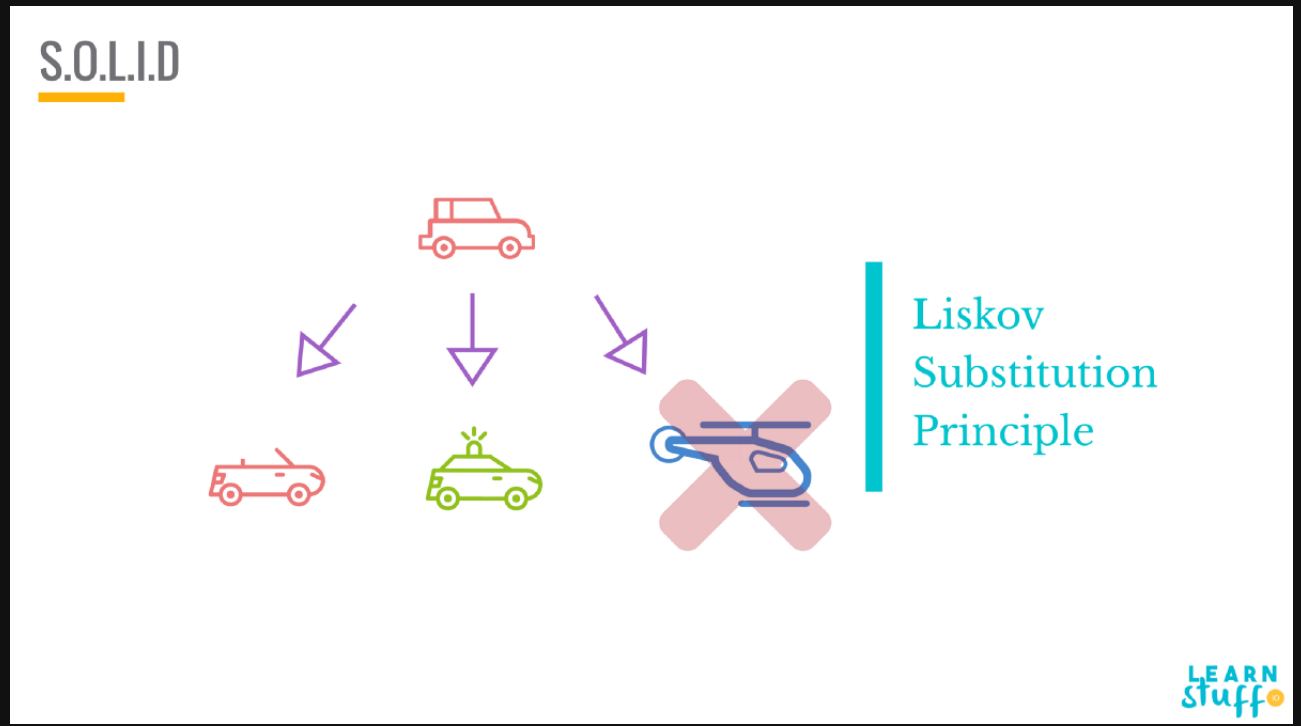
Precondition Weakening & Postcondition Strengthening

# Liskov Substitution Principle

What is it?

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



# 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.

# Feedback

https://forms.gle/R4sMTTQzPC4vqXSN8