

# COMP2511

23T2 Week 4

Wednesday 1PM - 4PM (W13A)

Thursday 3PM - 6PM (H15B)

Friday 2PM - 5PM (F14B)

Slides by Alvin Cherk (z5311001)

# Assignment Tips

Use `.equals()` for String (or any Class) comparison instead of `=`

```
1 public class EqualsExample {
2     public static void main(String[] args) {
3         String s1 = "Hello"; // string literal
4         String s2 = new String("Hello"); // string class
5         System.out.println(s1 == s2); // false, as its comparing the addresses of s1 and s2
6         System.out.println(s1.equals(s2)); // true, compares the values at the addresses
7         //             ^ do this
8     }
9 }
```

Correctness + style issue

# Assignment Tips

Avoid the use of magic numbers, assign them to final variables  
(readability - style issue)

```
1 public class MagicNumbers {
2     List<String> files = new ArrayList<>();
3
4     public void doSomething() {
5         if (files.size() == 5) { // 5 here is a magic number
6                                 // What does it mean?
7         }
8     }
9 }

1 public class MagicNumbers {
2     List<String> files = new ArrayList<>();
3     private final static int MAX_FILES = 5;
4     // declare constant with good name ^
5     public void doSomething() {
6         if (files.size() == MagicNumbers.MAX_FILES) {};
7         double angle = 750; // ^ use constant here
8         angle = angle % 360; // This is more OK, since theres context
9     }
10 }
```

# Assignment Tips

Don't use **super.x** to set attributes in the super class in the subclass's constructor. Pass variable into super constructor as argument (single responsibility - design issue)

```
1 // Example: What not to do
2 public class SatelliteParent {
3     private String name;
4     private double height;
5
6     public SatelliteParent(String name) {
7         this.name = name;
8         // height is currently unset
9     }
10
11     public void setHeight(double height) {
12         this.height = height;
13     }
14 }
15
16 public class SatelliteSubclass extends SatelliteParent {
17     public SatelliteSubclass(String name, double height) {
18         super(name);
19         this.setHeight(height); // bad
20         this.height = height; // also bad if not private
21     }
22 }
```

# Assignment Tips

Don't use **super.x** to set attributes in the super class in the subclass's constructor. Pass variable into super constructor as argument (single responsibility - design issue)

```
1 // Example: What to do
2 public class SatelliteParent {
3     private String name;
4     private double height;
5
6     public SatelliteParent(String name, double height) {
7         this.name = name;
8         this.height = height;
9     }
10 }
11
12 public class SatelliteSubclass extends SatelliteParent {
13     private String somethingElse;
14
15     public SatelliteSubclass(String name, double height, String somethingElse) {
16         super(name, height);
17         this.somethingElse = somethingElse;
18     }
19 }
```

# Assignment Tips

Use **instanceof** for type comparison (open/close - design issue)

```
1 public abstract class Shape {
2     public abstract String getType();
3 }
4
5 public class Rectangle extends Shape {
6     public void doSomething(List<Shape> shapes) {
7         for (Shape s : shapes) {
8             if (s.getClass().equals(Rectangle.class)) { // v1: kinda ok
9                 // do something only on exactly the Rectangle class
10            }
11            if (s.getType().equals("Rectangle")) { // v2: very bad
12                // do something on all rectangles
13            }
14            if (s instanceof Rectangle) { // v3: good
15                // do something on all rectangles
16            }
17        }
18    }
19 }
20
21 public class Square extends Rectangle {}
```

# Assignment Tips

Polymorphism is preferred over typechecking to perform a specific action (open/closed - design issue)

```
1 // Example: What not to do
2 public abstract class Shape {
3     public abstract String getType();
4 }
5 public class Rectangle extends Shape {}
6
7 public class Square extends Rectangle {
8     public static void main(String[] args) {
9         List<Shape> shapes = new ArrayList<>();
10        shapes.add(new Rectangle());
11        shapes.add(new Square());
12        for (Shape s : shapes) {
13            if (s.getType().equals("Rectangle")) {
14                // calculate the area this way
15            } else if (s.getType().equals("Square")) {
16                // calculate the area a different way
17            }
18        }
19    }
20 }
```

# Assignment Tips

Polymorphism is preferred over typechecking to perform a specific action (open closed - design issue)

```
1 // Example: What to do
2 public abstract class Shape {
3     public abstract String getType();
4     public abstract double area();
5     // ^ declare method in superclass
6 }
7
8 public class Rectangle extends Shape {
9     public double area() {
10         // calculate the area this way
11     }
12 }
13
14 public class Square extends Rectangle {
15     public double area() {
16         // calculate the area a different way
17     }
18     public static void main(String[] args) {
19         List<Shape> shapes = new ArrayList<>();
20         shapes.add(new Rectangle());
21         shapes.add(new Square());
22         for (Shape s : shapes) {
23             s.area(); // no more type checking
24         }
25     }
26 }
```



# 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

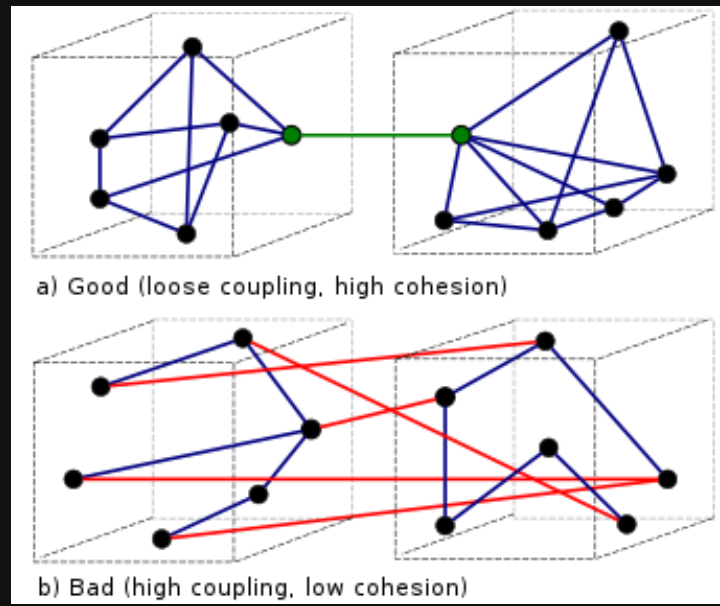
"Principle of least knowledge"

# 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).

```

1 public class TrainingSystem {
2     private List<Trainer> trainers;
3
4     public LocalDate bookTraining(String employee, List<LocalDate> availability) {
5         for (Trainer trainer : trainers) {
6             for (Seminar seminar : trainer.getSeminars()) {
7                 for (LocalDate available : availability) {
8                     if (seminar.getStart().equals(available) &&
9                         seminar.getAttendees().size() < 10) {
10                        seminar.getAttendees().add(employee);
11                        return available;
12                    }
13                }
14            }
15        }
16        return null;
17    }
18 }

```

```

1 /**
2  * An in person all day seminar with a maximum of 10 attendees.
3  */
4 public class Seminar {
5     private LocalDate start;
6     private List<String> attendees;
7
8     public LocalDate getStart() {
9         return start;
10    }
11
12    public List<String> getAttendees() {
13        return attendees;
14    }
15 }

```

```

1 /**
2  * A trainer that runs in person seminars.
3  */
4 public class Trainer {
5     private String name;
6     private String room;
7     private List<Seminar> seminars;
8
9     public List<Seminar> getSeminars() {
10        return seminars;
11    }
12 }

```

```

1 /**
2  * An online seminar is a video that can be viewed at any time
3  * by employees. A
4  * record is kept of which employees have watched the seminar.
5  */
6 public class OnlineSeminar extends Seminar {
7     private String videoURL;
8     private List<String> watched;
9 }

```

How and why does it violate this principle?

What other properties of this design are not desirable?

```

1 public class TrainingSystem {
2     public List<Trainer> trainers;
3     /**
4      * Try to booking training for an employee, given their availability.
5      *
6      * @param employee
7      * @param availability
8      * @return The date of their seminar if booking was successful, null there
9      * are no empty slots in seminars on the day they are available.
10    */
11    public LocalDate bookTraining(String employee, List<LocalDate> availability) {
12        for (Trainer trainer : trainers) {
13            LocalDate booked = trainer.book(employee, availability);
14            if (booked != null)
15                return booked;
16        }
17        return null;
18    }
19 }

1 /**
2  * An in person all day seminar with a maximum of 10 attendees.
3  */
4 public class Seminar {
5     private LocalDate start;
6     private List<String> attendees;
7     public LocalDate getStart() {
8         return start;
9     }
10
11    /**
12     * Try to book this seminar if it occurs on one of the available days and
13     * isn't already full
14     * @param employee
15     * @param availability
16     * @return The date of the seminar if booking was successful, null otherwise
17     */
18    public LocalDate book(String employee, List<LocalDate> availability) {
19        for (LocalDate available : availability) {
20            if (start.equals(available) &&
21                attendees.size() < 10) {
22                attendees.add(employee);
23                return available;
24            }
25        }
26        return null;
27    }
28 }

```

```

1 /**
2  * A trainer that runs in person seminars.
3  */
4 public class Trainer {
5     private String name;
6     private String room;
7     private List<Seminar> seminars;
8
9     public List<Seminar> getSeminars() {
10         return seminars;
11     }
12
13    /**
14     * Try to book one of this trainer's seminars.
15     * @param employee
16     * @param availability
17     * @return The date of the seminar if booking was successful, null if the
18     * trainer has no free slots in seminars on the available days.
19     */
20    public LocalDate book(String employee, List<LocalDate> availability) {
21        for (Seminar seminar : seminars) {
22            LocalDate booked = seminar.book(employee, availability);
23            if (booked != null)
24                return booked;
25        }
26        return null;
27    }
28 }

1 /**
2  * An online seminar is a video that can be viewed at any time by
3  * employees. A
4  * record is kept of which employees have watched the seminar.
5  */
6 public class OnlineSeminar extends Seminar {
7     private String videoURL;
8     private List<String> watched;
9 }

```

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



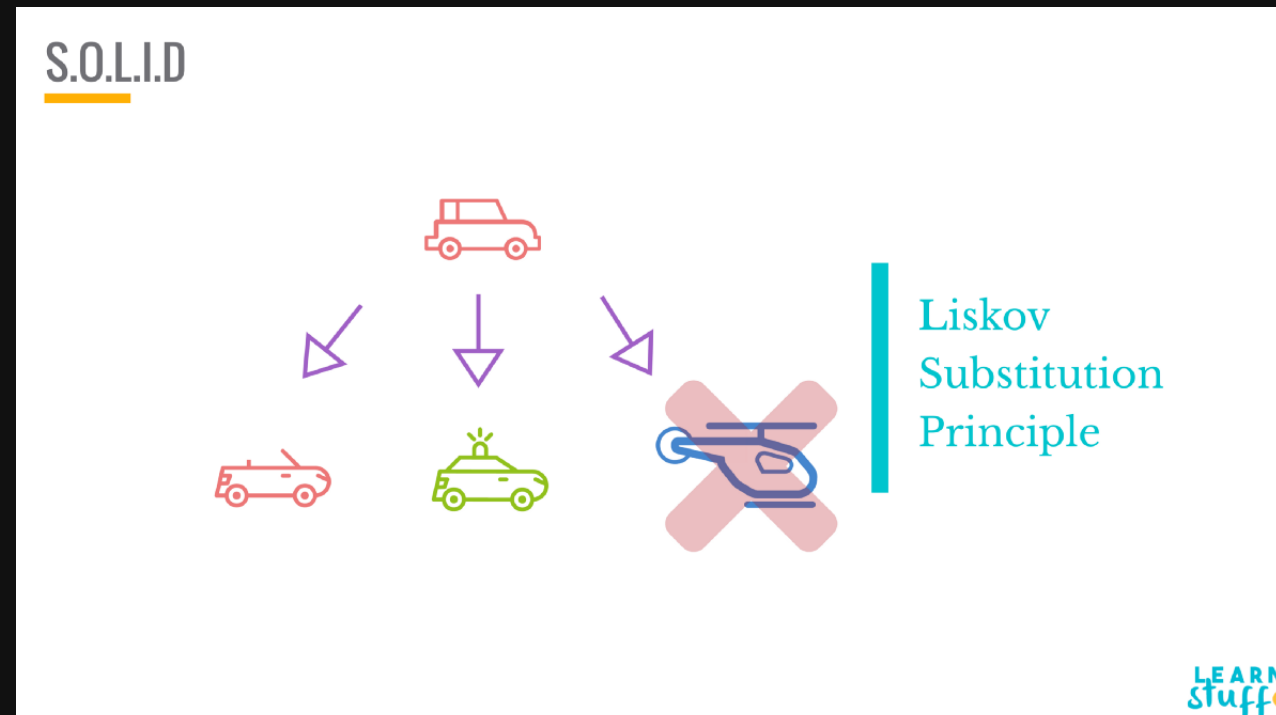
# Liskov Substitution Principle

# 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

```
1 /**
2  * An in person all day seminar with a maximum of 10 attendees.
3  */
4 public class Seminar {
5     private LocalDate start;
6     private List<String> attendees;
7
8     public LocalDate getStart() {
9         return start;
10    }
11
12    public List<String> getAttendees() {
13        return attendees;
14    }
15 }
```

```
1 /**
2  * An online seminar is a video that can be viewed at any time
3  * by employees. A
4  * record is kept of which employees have watched the seminar.
5  */
6 public class OnlineSeminar extends Seminar {
7     private String videoURL;
8     private List<String> watched;
9 }
```

Where does **OnlineSeminar** violate LSP?

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

<https://www.youtube.com/embed/hxGOiiR9ZKg?enablejsapi=1>

# 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

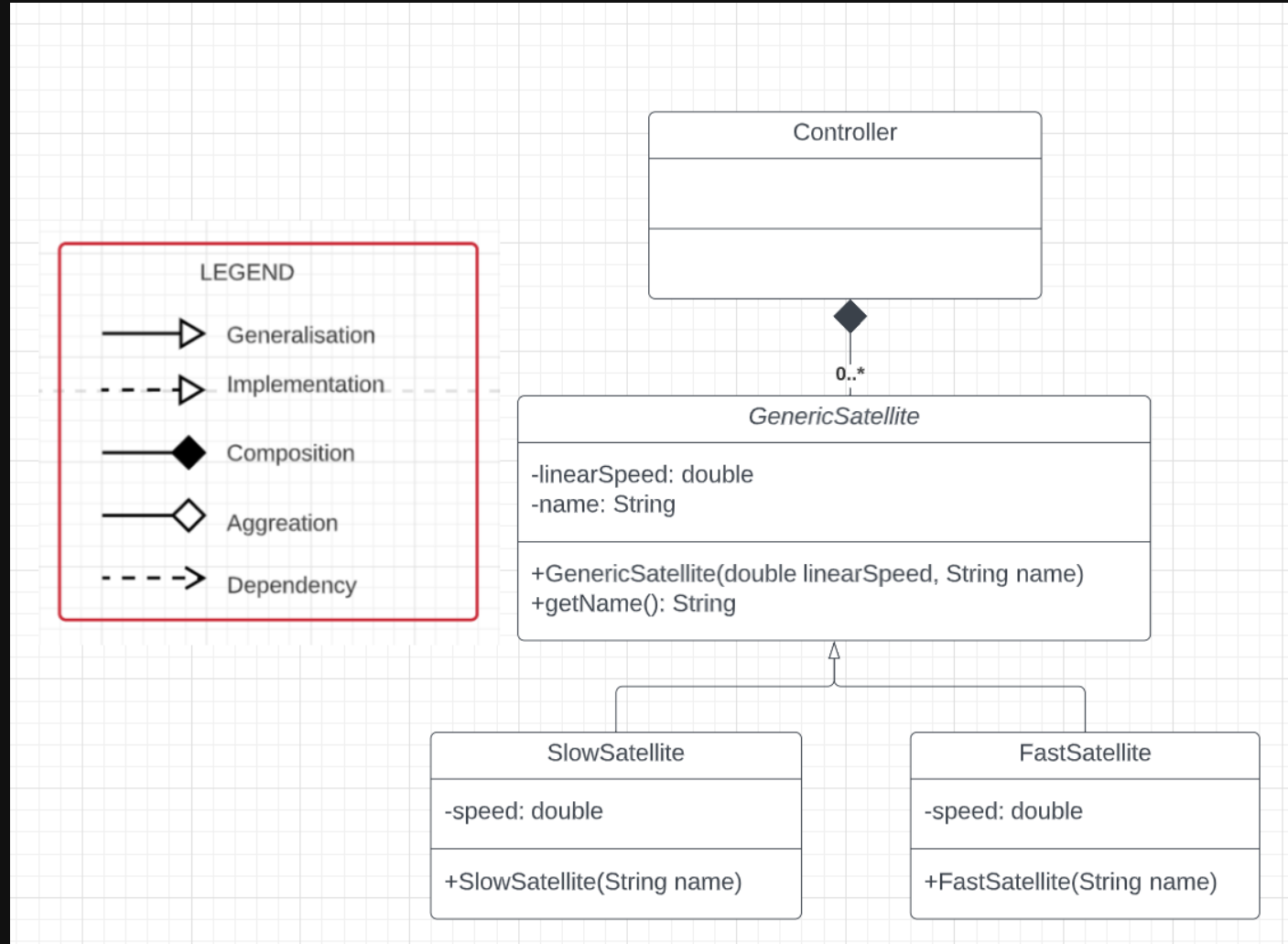
```
1 package stream;
2
3 import java.util.ArrayList;
4 import java.util.Arrays;
5 import java.util.List;
6 import java.util.stream.Collectors;
7
8 public class App {
9     public static void main(String[] args) {
10         List<String> strings = new ArrayList<String>(Arrays.asList(new String[] { "1", "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 }
```

```

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         // Same thing
12         strings.stream().forEach(x -> System.out.println(x));
13         // Use if there is more than one line of code needed in lambda
14         strings.stream().forEach(x -> {
15             System.out.println(x);
16         });
17
18         List<String> strings2 = new ArrayList<String>(Arrays.asList(new String[] { "1", "2", "3", "4", "5" }));
19         List<Integer> parsedStrings = strings2.stream().map(x -> Integer.parseInt(x)).collect(Collectors.toList());
20         strings2.stream().map(x -> Integer.parseInt(x)).forEach(x -> System.out.println(x));
21     }
22 }

```

# Further Streams Example



```

1 package stream;
2
3 public abstract class GenericSatellite {
4     private double linearSpeed; // in KM/minute
5     private String name; // name of satellite
6
7     public GenericSatellite(double linearSpeed, String name) {
8         this.linearSpeed = linearSpeed;
9         this.name = name;
10    }
11
12    public String getName() {
13        return this.name;
14    }
15
16    public double getLinearSpeed() {
17        return this.linearSpeed;
18    }
19
20    @Override
21    public String toString() {
22        return "{" +
23            "name='" + getName() + "'" +
24            "}";
25    }
26
27 }

```

```

1 package stream;
2
3 public class FastSatellite extends GenericSatellite {
4     private static final double speed = 100.0;
5
6     public FastSatellite(String name) {
7         super(speed, name);
8     }
9 }

```

```

1 package stream;
2
3 public class SlowSatellite extends GenericSatellite {
4     private static final double speed = 50.0;
5
6     public SlowSatellite(String name) {
7         super(SlowSatellite.speed, name);
8     }
9 }

```

```

1 package stream;
2
3 import java.util.ArrayList;
4 import java.util.List;
5 import java.util.stream.Collectors;
6
7 public class Controller {
8     private List<GenericSatellite> satellites = new ArrayList<>();
9
10    public void addSatellite(GenericSatellite satelliteToAdd) {
11        this.satellites.add(satelliteToAdd);
12    }
13
14    public List<GenericSatellite> getSatelliteList() {
15        return this.satellites;
16    }
17
18    public static void main(String[] args) { ... }
19
20 }

```

```

1 public static void main(String[] args) {
2     Controller c = new Controller();
3     c.addSatellite(new FastSatellite("Fast Satellite 1"));
4     c.addSatellite(new FastSatellite("Fast Satellite 2"));
5     c.addSatellite(new SlowSatellite("Slow Satellite 1"));
6
7     List<GenericSatellite> allSatellites = c.getSatelliteList();
8     System.out.println("All: " + allSatellites);
9     // [{name='Fast Satellite 1'}, {name='Fast Satellite 2'}, {name='Slow Satellite 1'}]
10
11     // What if I just want FastSatellites only?
12     // Method 1, normal for-in/for-each
13     List<GenericSatellite> fast1 = new ArrayList<>();
14     for (GenericSatellite x : allSatellites) {
15         if (x instanceof FastSatellite) {
16             fast1.add(x);
17         }
18     }
19     System.out.println("Just fast: " + fast1);
20     // [{name='Fast Satellite 1'}, {name='Fast Satellite 2'}]
21
22     // Method 2, streams
23     List<GenericSatellite> fast2 = allSatellites.stream().filter(x -> x instanceof FastSatellite).collect(Collectors.toList());
24     System.out.println("Just fast: " + fast2);
25     // [{name='Fast Satellite 1'}, {name='Fast Satellite 2'}]
26
27     // Same, but I typecast at the same time
28     List<FastSatellite> fast3 = allSatellites.stream().filter(x -> x instanceof FastSatellite).map(x -> (FastSatellite) x)
29         .collect(Collectors.toList());
30     System.out.println("Just fast: " + fast3);
31     // [{name='Fast Satellite 1'}, {name='Fast Satellite 2'}]
32 }

```

```

1 public static void main(String[] args) {
2     Controller c = new Controller();
3     c.addSatellite(new FastSatellite("Fast Satellite 1"));
4     c.addSatellite(new FastSatellite("Fast Satellite 2"));
5     c.addSatellite(new SlowSatellite("Slow Satellite 1"));
6
7     List<GenericSatellite> allSatellites = c.getSatelliteList();
8     System.out.println("All: " + allSatellites);
9     // [{name='Fast Satellite 1'}, {name='Fast Satellite 2'}, {name='Slow Satellite 1'}]
10
11     // What if im trying to search for something?
12     // Look for satellite with name == "Fast Satellite 2"
13     // Method 1, normal for-in/for-each
14     GenericSatellite g1 = null;
15
16     for (GenericSatellite x : allSatellites) {
17         if (x.getName().equals("Fast Satellite 2")) {
18             g1 = x;
19             break;
20         }
21     }
22     System.out.println(g1);
23     // {name='Fast Satellite 2'}
24
25     // Method 2, streams
26     GenericSatellite g2 = allSatellites.stream().filter(x -> x.getName().equals("Fast Satellite 2")).findFirst().orElse(null);
27     System.out.println(g2);
28     // {name='Fast Satellite 2'}
29
30     // Now I search for "Fast Satellite 3", which doesn't exist
31     GenericSatellite g3 = allSatellites.stream().filter(x -> x.getName().equals("Fast Satellite 3")).findFirst().orElse(null);
32     System.out.println(g3);
33     // null
34 }

```



# 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 }
```

```

1 public class Calculator {
2     /**
3      * @preconditions a, b != null
4      * @postconditions a + b
5      */
6     public static Double add(Double a, Double b) {
7         return a + b;
8     }
9
10    /**
11     * @preconditions a, b != null
12     * @postconditions a - b
13     */
14    public static Double subtract(Double a, Double b) {
15        return a - b;
16    }
17
18    /**
19     * @preconditions a, b != null
20     * @postconditions a * b
21     */
22    public static Double multiply(Double a, Double b) {
23        return a * b;
24    }
25
26    /**
27     * @preconditions a, b != null, b != 0
28     * @postconditions a / b
29     */
30    public static Double divide(Double a, Double b) {
31        return a / b;
32    }
33
34    /**
35     * @preconditions angle != null
36     * @postconditions sin(angle)
37     */
38    public static Double sin(Double angle) {
39        return Math.sin(angle);
40    }
41
42    /**
43     * @preconditions angle != null
44     * @postconditions cos(angle)
45     */
46    public static Double cos(Double angle) {
47        return Math.cos(angle);
48    }
49
50    /**
51     * @preconditions angle != null, angle != Math.PI / 2
52     * @postconditions tan(angle)
53     */
54    public static Double tan(Double angle) {
55        return Math.tan(angle);
56    }
57 }

```

# Liskov Substitution Principle

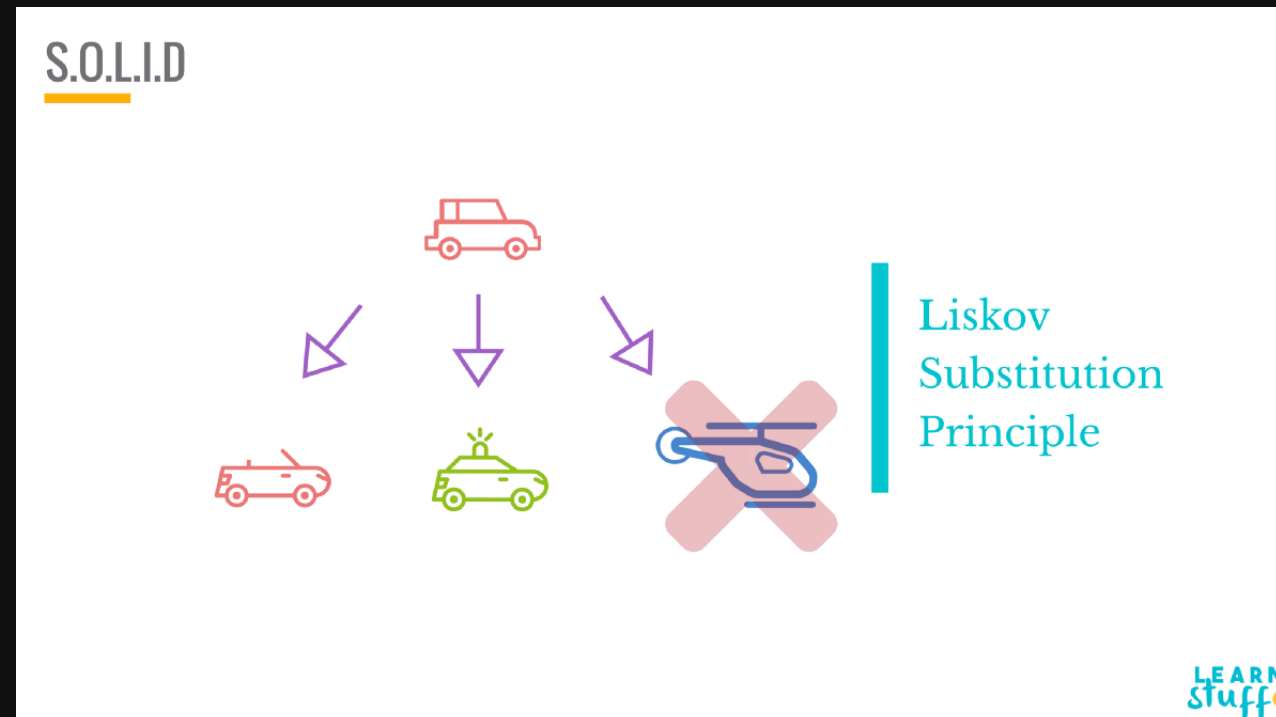
Precondition Weakening & Postcondition Strengthening

# 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 Weakening

- 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 lessen the obligation of the client, but not increase it

from  $0 \leq \theta \leq 90$  to  $0 \leq \theta \leq 180$  is weakening  
 $[0, 90] \Rightarrow [0, 180]$

Why?

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



# Feedback



<https://forms.gle/R4sMTTQzPC4vqXSN8>