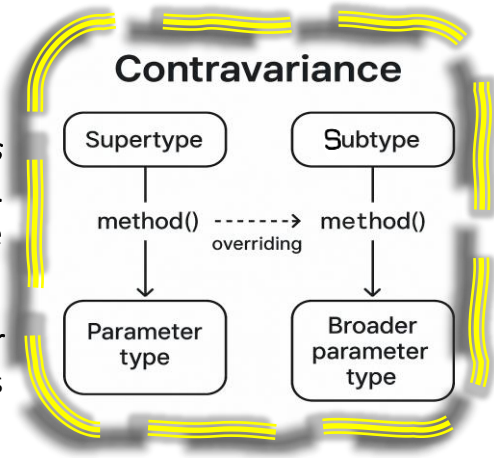
Guidelines to follow the Liskov Substitution Principle (LSP) properly:

- Signature rule → Method argument rule
  → Return type rule
  → Exception rule
- Property rule → Class invariant rule
  → History constraint rule
- Method rule → Pre-condition rule
  → Post-condition rule

Signature rule:

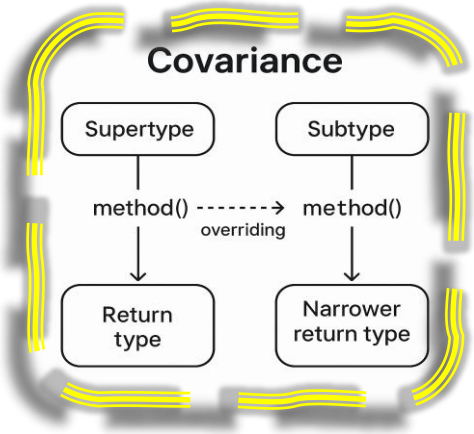
Method argument rule

- The ***child's method*** must accept ***inputs*** that are at least ***as flexible as*** the ***parent's method***. It can be ***equal*** or ***broader***, but never ***narrower***. This ensures we can safely use the child anywhere we have used the parent ***without causing*** unexpected ***errors***.
- Use of ***@Override*** annotation to the overwritten method, the ***compiler*** would give an ***error***, because the ***method signatures don't match***. This is how ***Java*** prevents this specific kind of violation.



Return type rule

- ***Covariance*** refers to a concept where a ***subclass*** is allowed to override a method and ***return*** a ***more specific type*** (i.e., a subtype) than the one declared in the ***superclass***.
- This is safe because wherever the ***superclass return type*** is expected, the subclass's ***narrower*** type will still work.



same/narrower

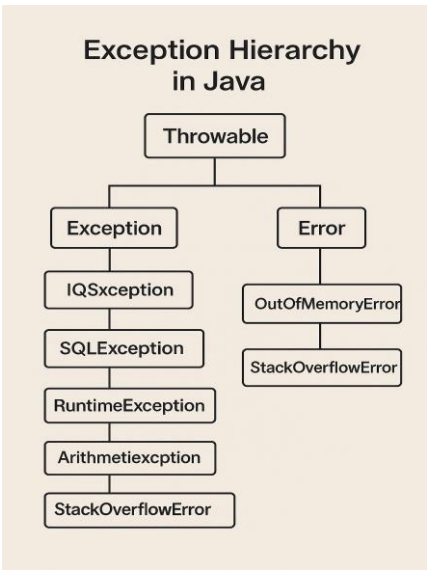
```
class Animal {
    Number numberOfLegs () {
        return 2;
    }
}
class Cow extends Animal {
    @Override
    Integer numberOfLegs () {
        return 4;
    }
}
```

Exception rule

- Child class can only throw the same exceptions as the parent class, or narrower (more specific) exceptions.
- Violating this rule can crash client code that relies on the parent's behaviour.

```
class Animal {
    Integer numberOfLegs () throws RuntimeException {
        throw new RuntimeException ();
    }
}
class Cow extends Animal {
    @Override
    Integer numberOfLegs () throws ArithmeticException {
        throw new ArithmeticException ();
    }
}
```

Same/Narrower



@Sibasundarj8

Property rule:

Class invariant rule

- A ***child class*** must ***preserve*** all class ***invariants*** of its ***parent class***. This means that any ***condition*** or ***rule*** that is ***always true*** for an ***object of the base class*** must also ***always remain true*** for an ***object of the subclass***.

- ***Subclasses*** must not ***break*** or ***relax*** any ***logical guarantees*** (invariants) defined by their ***base class***. The ***object*** must always ***maintain*** the same fundamental ***truths*** as its ***parent***.

```
class Rectangle {
   private Integer height, width;
   public void setHeight ( Integer height ) { this.height = height; }
   public void setWidth ( Integer width ) { this.width = width; }
   public Integer getArea () { return height * width; }
}
class Square extends Rectangle {
   private Integer height, width;
   public void setHeight ( Integer height ) {
      this.height = height;
      this.width = height;                    // Violating LSP
   }
   public void setWidth ( Integer width ) {
      this.height = width;
      this.width = width;
   }
   public Integer getArea () { return height * width; }
}
```

```
Throwable
    ├── Error
    │       └── OutOfMemoryError
    │       └── StackOverflowError
    │       └── ...
    └── Exception
            ├── IOException
            ├── SQLException
            ├── ...
            └── RuntimeException
                    ├── NullPointerException
                    ├── IndexOutOfBoundsException
                    ├── ArithmeticException
                    └── ...
```

History constraint rule

- This rule ensures that the ***"history"*** or ***state evolution*** of objects behaves the ***same*** in the ***child class*** as it would in the ***parent class***.

- The "history constraint" refers to **how the state of an object evolves over time** through method calls.

- Under LSP, ***a child class must preserve this evolution pattern*** → it should not allow clients to change object state in ways that ***were restricted in the parent class***.

```
class Bird {
   public void fly () {
      System.out.println("Flying");
   }
}
class Penguin extends Bird {

   @Override     // breaking the history constraints (LSP)
   public void fly () {
      throw new RuntimeException("penguin can't fly");
   }
}
```

```
class ImmutablePoint {
   final int x, y;
   ImmutablePoint(int x, int y) {
      this.x = x;
      this.y = y;
   }
   // No setters - state is immutable
}
class MutablePoint extends ImmutablePoint {
   MutablePoint(int x, int y) {super(x, y);}
   void setX(int newX) {}// changes internal state!
   void setY(int newY) {}// changes internal state!
}
```

Method rule:

Pre-condition rule

- A ***subclass*** must ***not strengthen*** the ***preconditions*** of the method it overrides from the ***superclass***.
- Means,
  - The subclass can ***weaken*** or ***keep*** the ***same preconditions***.
  - But it ***can't require more restrictive conditions*** for the method to work.
- This ensures that any code using a parent class can also use any of its child classes without knowing the difference.

```
class User {
   public String setPassword(String s) {
      if (s.length() < 8) {       // stronger password
         throw new RuntimeException("at least 8 characters");
      }
      return s;
   }
}
class AdminUser extends User {
   @Override
   public String setPassword(String s) {
      if (s.length() < 5) {       // makes password weaker
         throw new RuntimeException("at least 5 characters");
      }
      return s;
   }
}  // pre-condition of superclass is always stronger then subclass
```

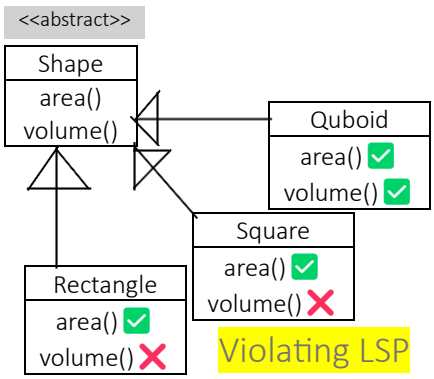
Post-condition rule

- A ***child class's*** methods ***cannot weaken*** the ***post-conditions*** established by its ***parent class***.
- Essentially, if a ***parent class guarantees*** a ***certain state*** after a method call, the ***child class*** must also ***guarantee at least that same state***, or a ***stronger*** one, when overriding that method.
- LSP ensures that ***subclasses*** maintain the ***behavioural contracts*** established by their ***superclasses***. If a method in a ***subclass*** behaves ***differently*** than ***expected***, it ***violates LSP*** and can lead to subtle and ***hard-to-debug*** issues in the application.

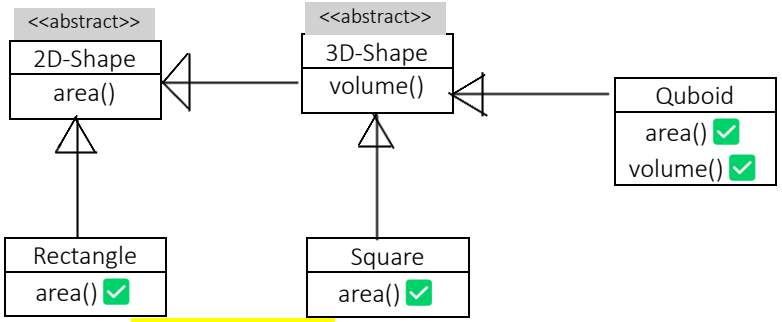
```
class Car {
   protected int speed;
   public Car (int acceleration) {
      this.speed += acceleration;
   }
   // We can reduce the speed by applying break
   public void applyBreak () {
      speed -= Math.min(speed, 30);
   }
   public int getSpeed (){
      return speed;
   }
}
class brokenCar extends Car {
   public brokenCar(int acceleration) {
      super(acceleration);
   }
   // violating the LSP speed remain same after applying break
   @Override
   public void applyBreak() {
      super.speed -= Math.min(speed, 0);
   }
}
```

Interface Segregation Principle (ISP):

- Many ***client-specific interfaces*** are ***better*** than ***one general-purpose interface***.
- A ***client*** should not be forced to ***depend*** on interfaces it ***does not use***.
- Rather than having ***one large interface*** that serves many purposes (and potentially has many methods), it's ***better to have several smaller***, more ***specific interfaces*** tailored to the needs of different clients. This improves ***maintainability***, ***modularity***, and ***flexibility***.



Violating LSP



Following LSP

@Sibasundarj8

```java
interface Shape {
    public void area () ;
    public void volume();
}
class Rectangle implements Shape{
    @Override
    public void area() {
        System.out.println("calculating");
    }
    @Override                          // Violating ISP
    public void volume() {// doesn't contain volume }
}
class Square implements Shape{
    @Override
    public void area() {
        System.out.println("calculating");
    }
    @Override
    public void volume() {doesn't contain volume }
}
class Quboid implements Shape{
    @Override
    public void area() {
        System.out.println("calculating");
    }
    @Override
    public void volume() {
        System.out.println("calculating");
    }
}
```

```java
interface Shape_2D {
    public void area () ;
}
interface Shape_3D extends Shape_2D {
    public void volume ();                    // Following ISP
}
class Rectangle implements Shape_2D {
    @Override
    public void area() {
        System.out.println("calculating");
    }
}
class Square implements Shape_2D {
    @Override
    public void area() {
        System.out.println("calculating");
    }
}
class Quboid implements Shape_3D {
    @Override
    public void area() {
        System.out.println("calculating");
    }
    @Override
    public void volume() {
        System.out.println("calculating");
    }
}
```

Dependency Inversion Principle (DIP):

- *DIP* states that ***high-level*** modules should ***not depend*** on ***low-level*** modules. Both should ***depend*** on ***abstractions***.
- It means that instead of ***high-level*** modules ***directly depending*** on ***concrete implementations*** of ***low-level*** modules, they should depend on ***interfaces*** or ***abstract classes***.
- This allows for ***flexibility*** and makes it easier to ***change*** or ***extend*** the system without affecting other parts of the code.

```java
class LightBulb {
    private boolean status = false;
    private void turnOn() {
        this.status = true;
        System.out.println("Light ON 💛");
    }
    private void turnOff() {
        this.status = false;
        System.out.println("Light Off 🖤");
    }
    public void trigger() {
        if (status) this.turnOff();
        else this.turnOn();
    }
}
class Switch {
    private final LightBulb bulb;
    Switch() {      // dependency injection
        this.bulb = new LightBulb();
    }
    public void operate() {
        this.bulb.trigger();
    }
}
```

```java
class User {
    public static void main(String[] args) {
        Switch sw = new Switch();
        sw.operate();
        sw.operate();
        sw.operate();
        sw.operate();
    }
}
```

It violates DIP.
Here is an issue when it comes to ***add*** a ***new fan*** or ***TV*** which could be ***controlled*** through ***switch logic implementation*** might be a little bit ***challenging***.
We have to add object of newly added device in the switch class and call methods separately and it ***violates open-close principle*** (***OCP***).

<<abstract>>

```
abstract class SwitchableDevice {
    protected boolean status = false;

    abstract protected void turnOn();

    abstract protected void turnOff();

    public void trigger() {
        if (status) this.turnOff();
        else this.turnOn();
    }
}
```

```
class LightBulb extends SwitchableDevice {

    @Override
    public void turnOn() {
        status = true;
        System.out.println("Light ON 💛");
    }

    @Override
    public void turnOff() {
        status = false;
        System.out.println("Light Off 🖤");
    }
}
```

```
class Fan extends SwitchableDevice {

    @Override
    public void turnOn() {
        status = true;
        System.out.println("Fan ON ☮💨");
    }

    @Override
    public void turnOff() {
        status = false;
        System.out.println("Fan Off ☮");
    }
}
```

It follows DIP,
Now we can easily *add* and *remove* devices without affecting any other part of code.

```
class Switch {
    private final SwitchableDevice3 device;

    Switch(SwitchableDevice device) { // dependency injection
        this.device = device;
    }

    public void operate() {
        device.trigger();
    }
}
```

```
class Main {
    public static void main(String[] args) {
        Switch sw1 = new Switch(new LightBulb());
        Switch sw2 = new Switch(new Fan());

        sw1.operate();
        sw2.operate();
        sw1.operate();
        sw2.operate();
    }
}
```

@Sibasundarj8