# CS2030 Lecture 4

## Interface: Contract Between Classes

Henry Chia (hchia@comp.nus.edu.sg)

Semester 1 2022 / 2023

# Lecture Outline and Learning Outcomes

- Be able to define and implement an **interface**
- Understand when to use inheritance and when to implement an interface
- Understand how inheritance and interfaces can both support polymorphism and substitutability
- Be able to define an **abstract class** for the purpose of inheritance
- Understand the **SOLID principles** and their application in the design of object-oriented software
- Familiarity with the *Java Collections Framework*
- Be able to make use of interfaces specified in the Java API

# Designing Circles and Rectangles as Shapes

- Define `Shape` as a parent class of `Circle` and `Rectangle` with corresponding properties and `getArea()` methods

```
class Shape {
    double getArea() { return -1.0; }
}

class Circle extends Shape {
    private final int radius;

    Circle(int radius) {
        this.radius = radius;
    }

    @Override
    double getArea() {
        return Math.PI * radius * radius;
    }

    @Override
    public String toString() {
        return "Circle with radius " +
            this.radius;
    }
}
```

```
class Rectangle extends Shape {
    private final int width;
    private final int height;

    Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }

    @Override
    double getArea() {
        return width * height;
    }

    @Override
    public String toString() {
        return "Rectangle " + this.width +
            " x " + this.height;
    }
}
```

```
jshell> new Shape() // does not make sense to create a Shape object!
$.. ==> Shape@68be2bc2

jshell> new Shape().getArea() // ???
$.. ==> -1.0
```

# Defining an Interface as a Contract

- Shape is not an object; it should only *specify behaviours* (or methods) to be defined in the implementation class
- Implementing the Shape interface as a "contract"

```
interface Shape {
    double getArea(); // specify getArea as a method of the contract
}
```

- Interface methods are implicitly **public**, hence overriding implementation methods are defined with the same access

```
class Circle implements Shape { // use the implements keyword
    private final int radius;

    Circle(int radius) {
        this.radius = radius;
    }

    @Override
    public double getArea() { // implement the contract method specification
        return Math.PI * this.radius * this.radius;
    }
    ...
```

# Implementing Multiple Interfaces

- Implementing behaviours specified in multiple interfaces

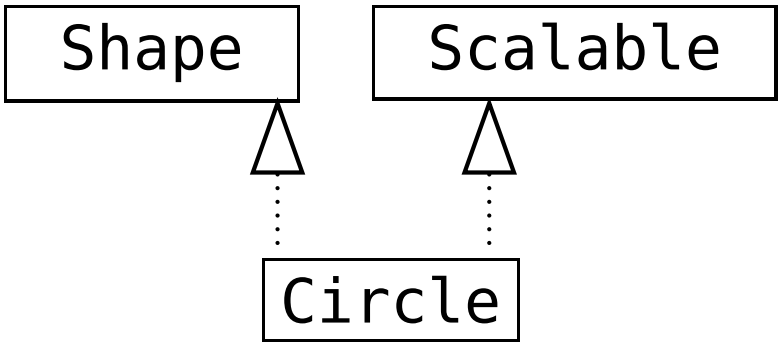
```
interface Scalable {
    Scalable scale(int factor);
}

class Circle implements Shape, Scalable {
    private final int radius;

    Circle(int radius) {
        this.radius = radius;
    }

    @Override
    public double getArea() { // implementing getArea from Shape
        return Math.PI * this.radius * this.radius;
    }

    @Override
    public Circle scale(int factor) { // implementing scale from Scalable
        return new Circle(this.radius * factor);
    }
    ...
```

- Unlike interfaces, a child class **cannot** extend from multiple parents; `class A extends B, C {...}` is invalid!

# Is-A Relationship Revisted

- An implementation class is *substitutable* for its interface
  - `Circle` *is a* `Shape`; `Circle` *is a* `Scalable`

```
jshell> Circle c = new Circle(1)
c ==> Circle with radius 1

jshell> Shape s = c
s ==> Circle with radius 1

jshell> s.getArea()
$.. ==> 3.141592653589793

jshell> s.scale(2) // scale is not defined in Shape
|  Error:
|  cannot find symbol
|    symbol:   method scale(int)
|  s.scale(2)
|  ^-----^

jshell> Scalable k = c
k ==> Circle with radius 1

jshell> k.scale(2)
$.. ==> Circle with radius 2

jshell> k.getArea() // getArea is not defined in Scalable
|  Error:
|  cannot find symbol
|    symbol:   method getArea()
|  k.getArea()
|  ^-------^
```



# From Concrete Class to Interfaces

- **Concrete class** defines the actual implementation with data (properties) and behaviour (methods)
- **Interface** specifies methods to be implemented, with no data
- **Abstract class** is a trade off between the two
  - can have properties to be inherited by child classes
  - can have some methods defined; hence cannot instantiate

```
abstract class FilledShape {
    protected final Color color;

    FilledShape(Color color) {
        this.color = color;
    }

    // declare method as abstract
    abstract double getArea();

    Color getColor() {
        return this.color;
    }
}
```

```
class Circle extends FilledShape {
    private final int radius;

    Circle(int radius, Color color) {
        super(color);
        this.radius = radius;
    }

    @Override
    double getArea() {
        return Math.PI * radius * radius;
    }
}
```

- Multiple inheritance, even for abstract classes, is not allowed

In CS2030, we use only "pure" interfaces. fyi, as of Java 8 "impure" interfaces can include default methods with implementations.

# SOLID Principles in OO Design

- **Single responsibility principle**:

  *a class should have only one reason to change*

  — *Robert C. Martin (Uncle Bob)*

- **Liskov substitution principle**:

  *Let $\phi(x)$ be a property provable about objects $x$ of type $T$. Then $\phi(y)$ should be true for objects $y$ of type $S$ where $S$ is a subtype of $T$.*

  — *Barbara Liskov*

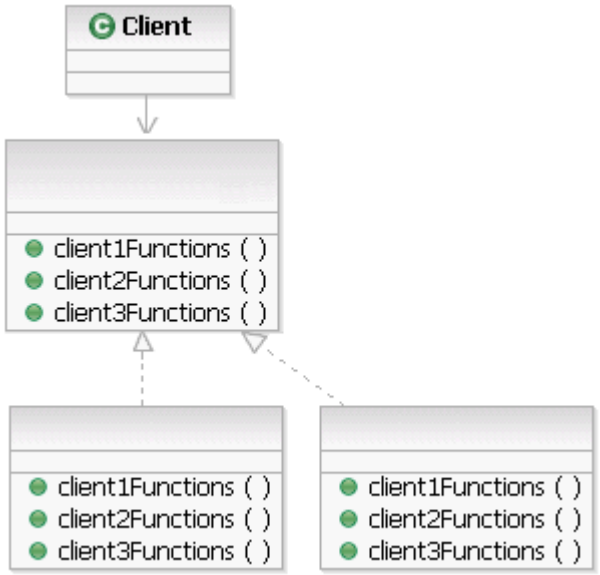
  - If $S$ is a *subtype* of $T$ (denoted `S <: T`), then an object of type $T$ can be replaced by that of type $S$ *without changing the desirable property* of the program

# SOLID Principles in OO Design

- **Open–closed principle**:

  classes should be *open for extension, but closed for modification* — *Bertrand Meyer*

```
jshell> class A { void foo() { } }
|  created class A

jshell> void client(A a) { a.foo(); }
|  created method client(A)

jshell> client(new A())

jshell> class B extends A { }
|  created class B

jshell> class C extends A { @Override void foo() { } }
|  created class C

jshell> class D extends B { @Override void foo() { } }
|  created class D

jshell> client(new B()) // client does not need modification

jshell> client(new C()) // C:foo() invoked

jshell> client(new D()) // D:foo() invoked
```



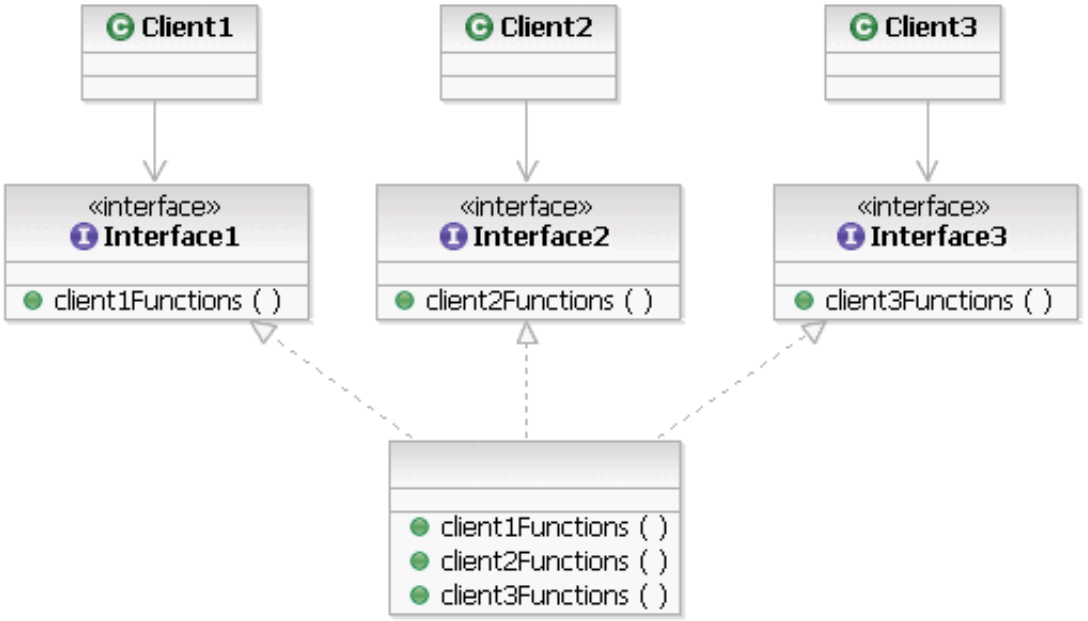
# SOLID Principles in OO Design

- **Interface segregation principle**:

  *no client should be forced to depend on methods it does not use.*

  — *Uncle Bob*

```
jshell> Circle circle = new Circle(1)
circle ==> Circle with radius 1

jshell> double client1(Shape s) {
   ...>     return s.getArea();
   ...> }
|  created method client1(Shape)

jshell> Scalable client2(Scalable k) {
   ...>     return k.scale(2);
   ...> }
|  created method client2(Scalable)

jshell> client1(circle)
$.. ==> 3.141592653589793

jshell> client2(circle)
$.. ==> Circle with radius 2
```



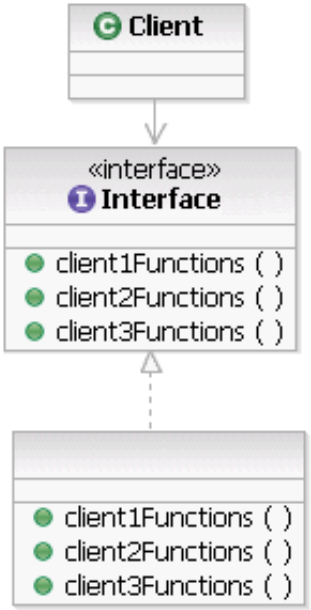
# SOLID Principles in OO Design

- **Dependency inversion principle**:

  *Program to an interface, not an implementation.*

  — *GoF*

```
jshell> /list Shape

   1 : interface Shape { // Shape is the contract
           double getArea();
       }
jshell> Shape s = new Circle(1)
s ==> Area 3.14 and perimeter 6.28

jshell> class Circle implements Shape { // Circle follows contract specs
   ...>    private final int radius;
   ...>    public double getArea() {
   ...>       return Math.PI * this.radius * this.radius;
   ...>    }
   ...> }
|  created class Circle

jshell> double client(Shape s) { // client codes according to contract
   ...>     return s.getArea();
   ...> }
|  created method client(Shape)

jshell> client(circle)
$.. ==> 3.141592653589793
```



# The `List` Interface

- `List` interface specifies a contract for implementing a *collection* of possibly duplicate objects with element order
- Classes that implement `List` includes:
  - public-facing: e.g. `ArrayList`, `LinkedList`, `Vector`

    ```
    List<Integer> list = new ArrayList<Integer>();
    ```
  - non-public-facing: e.g. `AbstractImmutableList`

    ```
    List<Integer> list = List.of(1);
    ```
- `List` *inherits* from a parent interface `Collection`
  - Java API provides collections to store related objects
    - provides methods that organize, store and retrieve data
    - there is no need to know how data is being stored

# Java Collections Framework

- Collection-framework interfaces declare operations to be performed generically on various type of collections

| Interface | Description |
|---|---|
| Collection | The root interface in the collections hierarchy from which interfaces Set, Queue and List are derived. |
| Set | A collection that does not contain duplicates. |
| List | An ordered collection that can contain duplicate elements. |
| Map | A collection that associates keys to values and cannot contain duplicate keys. |
| Queue | Typically a first-in, first-out collection that models a waiting line; other orders can be specified. |

- Collections of *generic* type <E> contain references to objects (elements) of type E

# Java Collections Framework: `List<E>`

| | | |
|---|---|---|
| void | **add**(int index, **E** element) | Inserts the specified element at the specified position in this list. |
| boolean | **add**(**E** e) | Appends the specified element to the end of this list. |
| void | **clear**() | Removes all of the elements from this list. |
| boolean | **contains**(**Object** o) | Returns `true` if this list contains the specified element. |
| **E** | **get**(int index) | Returns the element at the specified position in this list. |
| int | **indexOf**(**Object** o) | Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element. |
| boolean | **isEmpty**() | Returns `true` if this list contains no elements. |
| **E** | **remove**(int index) | Removes the element at the specified position in this list. |
| boolean | **remove**(**Object** o) | Removes the first occurrence of the specified element from this list, if it is present. |
| **E** | **set**(int index, **E** element) | Replaces the element at the specified position in this list with the specified element. |
| int | **size**() | Returns the number of elements in this list. |

- Methods specified in interface `Collection<E>`
  - `size()`, `isEmpty()`, `contains(Object)`, `add(E)`, `remove(Object)`, `clear()`
- Additional methods specified in interface `List<E>`
  - `indexOf(Object)`, `get(int)`, `set(int, E)`, `add(int, E)`, `remove(int)`,

# `Iterator` Interface

- Elements in a list can be looped successively via an *iterator*
- `Iterator` is the parent interface of `Collection`, and hence also the parent interface of `List`
  - `Iterator` interface specifies the `iterator()` method which returns an `Iterator`
    - `Iterator` is an interface that specifies the `next()` and `hasNext()` methods
- Any implementation of `List`, say `ArrayList`, has to implement the `iterator()` method which returns an implementation of the `Iterator` interface, say `Itr`
  - must define the `next()` and `hasNext()` methods

# `Iterator` Interface

- Using `Iterator`'s `hasNext()` and `next()` methods to iterate over list elements

```
jshell> List<Integer> list = List.of(1, 2, 3)
list ==> [1, 2, 3]

jshell> Iterator<Integer> iter = list.iterator()
iter ==> java.util.ImmutableCollections$ListItr@20e2cbe0

jshell> while (iter.hasNext()) { // Iterator is mutable!
   ...>     int i = iter.next(); // or Integer i = iter.next();
   ...>     System.out.print(i + " ");
   ...> }
1 2 3
```

- Using the enhanced **`for`** construct as syntactic sugar

```
jshell> List<Integer> list = List.of(1, 2, 3)
list ==> [1, 2, 3]

jshell> for (int i : list) {
   ...>     System.out.print(i + " ");
   ...> }
1 2 3
```