CS2030 Lecture 4

Interface: Contract Between Classes

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

Semester 2 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
- 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

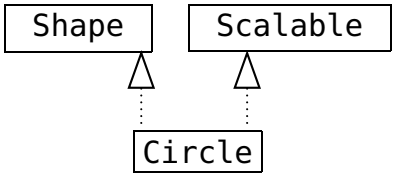
```
interface Scalable {
    Scalable scale(int factor);
}

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

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

Case Study: Java `List` Interface

- `List<E>` *generic* interface
  - specifies a contract for implementing a *collection* of possibly duplicate objects of type `E` with element order

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.

`List` Implementations

- Classes that implement `List` can be
  - mutable: e.g. `ArrayList`, `LinkedList`, `Vector`

```
jshell> List<Integer> list = new ArrayList<Integer>()
list ==> []

jshell> list.add(1)
$.. ==> true

jshell> list.get(0)
$.. ==> 1
```

  - immutable: e.g. `AbstractImmutableList` using `List.of(..)`
    - Read-access is allowed: `get`, `size`, `isEmpty`, ...

```
jshell> List.of(1, 2, 3).get(0)
$.. ==> 1
```

    - Write-access is **not** allowed: `add`, `remove`, `set`, `sort`...

```
jshell> List.of(1, 2, 3).add(4)
|  Exception java.lang.UnsupportedOperationException
|        at ImmutableCollections.uoe (ImmutableCollections.java:72)
|        at ImmutableCollections$AbstractImmutableCollection.add (ImmutableCollections.jav
|        at (#1:1)
```


Java Collections Framework

- `List<E>` *inherits* from a parent interface `Collection<E>`

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.

  - 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)`,

List Sorting Using a Comparator

- Example: sorting a list of shapes by
  - ascending order of area
  - descending order of perimeter
  - ...
- A possible (but highly unlikely) `sort` method for `List<E>`

```
void sort(Comparator<E> cmp) { // using bubble sort as an example
    for (int i = 0; i < this.size(); i++) {
        for (int j = i + 1; j < this.size() - 1; j++) {
            if (cmp.compare(this.get(i), this.get(j)) .. ) {
                ...
```

- Implementation of a `Comparator<E>` interface is passed to the `sort` method that specifies *how* two elements are compared
  - `compare(x,y)` should return < 0 if `x` comes first; > 0 if `y` comes first; or 0 otherwise

Example: `Comparator<Integer>`

- Sorting a list of integers in ascending order

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

jshell> class IntCompAsc implements Comparator<Integer>
   ...> public int compare(Integer i, Integer j) { return i - j; }}
|  created class IntComp

jshell> new IntCompAsc().compare(1, 2)
-1

jshell> list.sort(new IntCompAsc()) // ArrayList is mutable! :(

jshell> list
list ==> [1, 2, 3]
```

- Sorting a list of integers in descending order

```
jshell> list
list ==> [1, 2, 3]

jshell> class IntCompDsc implements Comparator<Integer>
   ...> public int compare(Integer i, Integer j) { return j - i; }}
|  created class IntCompDsc

jshell> new IntCompDsc().compare(1, 2)
1

jshell> list.sort(new IntCompDsc()) // or list.sort(new IntCompAsc().reversed())

jshell> list
list ==> [3, 2, 1]
```

Example: `Comparator<Shape>`

- Example: define `ShapeAreaComp` as an implementation of the `Comparator<Shape>` interface

```
jshell> class ShapeAreaComp implements Comparator<Shape> {
   ...>    public int compare(Shape s1, Shape s2) {
   ...>       double diff = s1.getArea() - s2.getArea();
   ...>       if (diff < 0) {
   ...>            return -1;
   ...>       } else if (diff > 0) {
   ...>            return 1;
   ...>       } else {
   ...>            return 0;
   ...>       }
   ...>    }
   ...> }
|  created class ShapeAreaComp

jshell> new ShapeAreaComp().compare(new Circle(1), new Rectangle(2, 3))
$.. ==> -1

jshell> new ShapeAreaComp().compare(new Rectangle(2, 3), new Rectangle(3, 2))
$.. ==> 0
```


Sorting `List<E>` using `Comparator<E>`

- Sorting list of shapes in ascending order of area

```
jshell> List<Shape> shapes = new List<Shape>()
shapes ==> []

jshell> shapes.add(new Rectangle(2, 3))
$.. ==> true

jshell> shapes.add(new Circle(1))
$.. ==> true

jshell> shapes
shapes ==> [Rectangle 2 x 3, Circle with radius 1]

jshell> shapes.sort(new ShapeAreaComp())

jshell> shapes
$.. ==> [Circle with radius 1, Rectangle 2 x 3] // state change!
```

- `ImList` has an *effect-free* sort implementation!

```
jshell> ImList<Shape> shapes = new ImList<Shape>(). // using ImList
   ...> add(new Rectangle(2, 3)).
   ...> add(new Circle(1))
shapes ==> [Rectangle 2 x 3, Circle with radius 1]

jshell> shapes.sort(new ShapeAreaComp()) // creates a new sorted list
$.. ==> [Circle with radius 1, Rectangle 2 x 3]

jshell> shapes // state remains unchanged
$.. ==> [Rectangle 2 x 3, Circle with radius 1]
```


`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
```