CS2030 Lecture 3

The Interface

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

Semester 1 2023 / 2024

Lecture Outline and Learning Outcomes

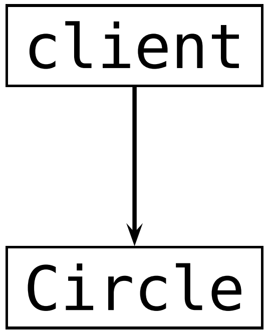
- Understand the need for a *contract* between the client and implementer
- Be able to define an **interface** and implement it in a class
- Appreciate that a class can implement multiple interfaces
- Familiarity with the *Java Collections Framework*
- Be able to make use of interfaces specified in the Java API
- Appreciate the use of `Iterable` and `Iterator` interfaces for iterating elements in a collection
- Appreciate the use of a `Comparator` interface that allows for a `compare` method to be defined for the purpose of ordering elements

Preamble

- Define a class `Circle` with radius and `getArea()` method

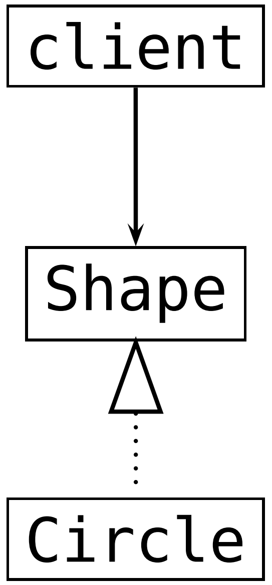
```
class Circle {
    private final double radius;

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

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

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

```
jshell> new Circle(1.0).getArea()
$.. ==> 3.141592653589793

jshell> double findVolume(Circle circle, double height) {
   ...>     return circle.getArea() * height;
   ...> }
|  created method findVolume(Circle,double)

jshell> findVolume(new Circle(1.0), 10.0)
$.. ==> 31.41592653589793
```



Contract

- The client is dependent on the implementation of `Circle`
- If `Circle` changes its implementation, the client will break!
  - e.g. renaming `getArea()` to `area()` instead
- To safeguard the interests of the client, establish a contract for both client and implementer to adhere to, e.g.
  - define the contract `Shape` that specifies the `getArea()` method
  - all implementers of `Shape` must define the method specifications of the contract
  - all clients of `Shape` should program to the contract, not the implementers



Defining an Interface as a Contract

- An interface *specifies public behaviours (methods)*

  ```
  interface Shape {
      public double getArea(); // getArea specification in the contract
  }
  ```

  to be *defined in the implementation class*

  ```
  class Circle implements Shape { // use the implements keyword
      ...
      public double getArea() { // implement public getArea() from Shape
          return Math.PI * this.radius * this.radius;
      }
      ...
  ```

- Interface methods are implicitly **`public`**
- An interface cannot be instantiated

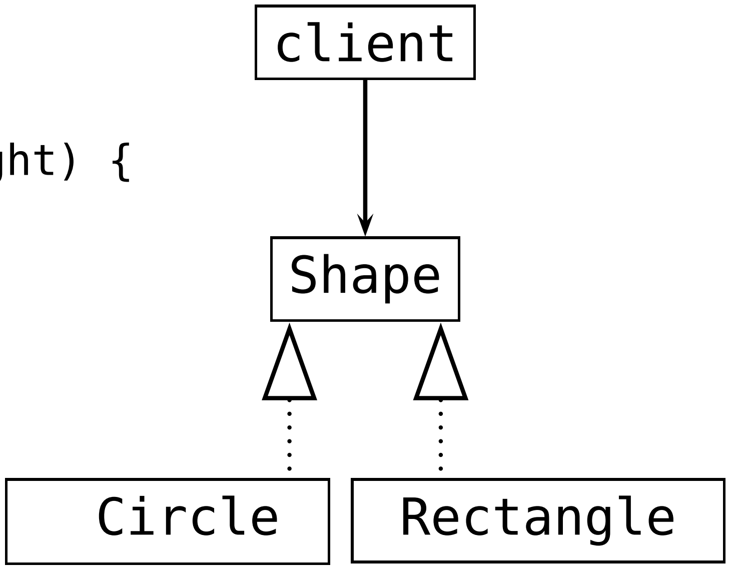
  ```
  jshell> new Shape()
  |  Error:
  |  Shape is abstract; cannot be instantiated
  |  new Shape()
  |  ^---------^
  ```

- Exercise: define `Rectangle` class that implements `Shape`

Is-A Relationship

- Circle is a Shape; Rectangle is a Shape

```
jshell> Circle circle = new Circle(1.0)
circle ==> Circle with radius 1.0

jshell> Shape shape = circle // Circle is-a Shape
shape ==> Circle with radius 1.0

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

jshell> shape = new Rectangle(2.0, 3.0) // Rectangle is-a Shape
$.. ==> Rectangle 2.0 x 3.0

jshell> shape.getArea()
$.. ==> 6.0

jshell> double findVolume(Shape shape, double height) {
   ...>     return shape.getArea() * height;
   ...> }
|  created method findVolume(Shape,double)

jshell> findVolume(new Circle(1.0), 10.0)
$.. ==> 31.41592653589793

jshell> findVolume(new Rectangle(2.0, 3.0), 10.0)
$.. ==> 60.0
```



Implementing Multiple Interfaces

- Implementing behaviours specified in multiple interfaces

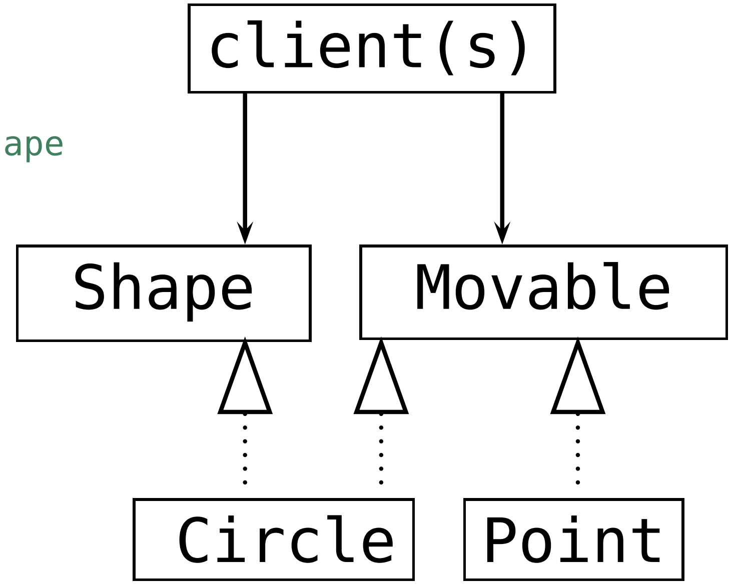
```java
interface Movable {
    // moving a Movable returns another Movable
    public Movable moveBy(double x, double y);
}

class Circle implements Shape, Movable {
    private final Point centre;
    private final double radius;

    Circle(Point centre, double radius) {
        this.centre = centre;
        this.radius = radius;
    }

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

    public Movable moveBy(double x, double y) { // from Movable interface
        return new Circle(this.centre.moveBy(x, y), this.radius);
    }
    ...
```

- Exercise: make `Point` movable

Programming to Interfaces

- Circle *is a* Shape; Circle *is a* Movable; Point *is a* Movable

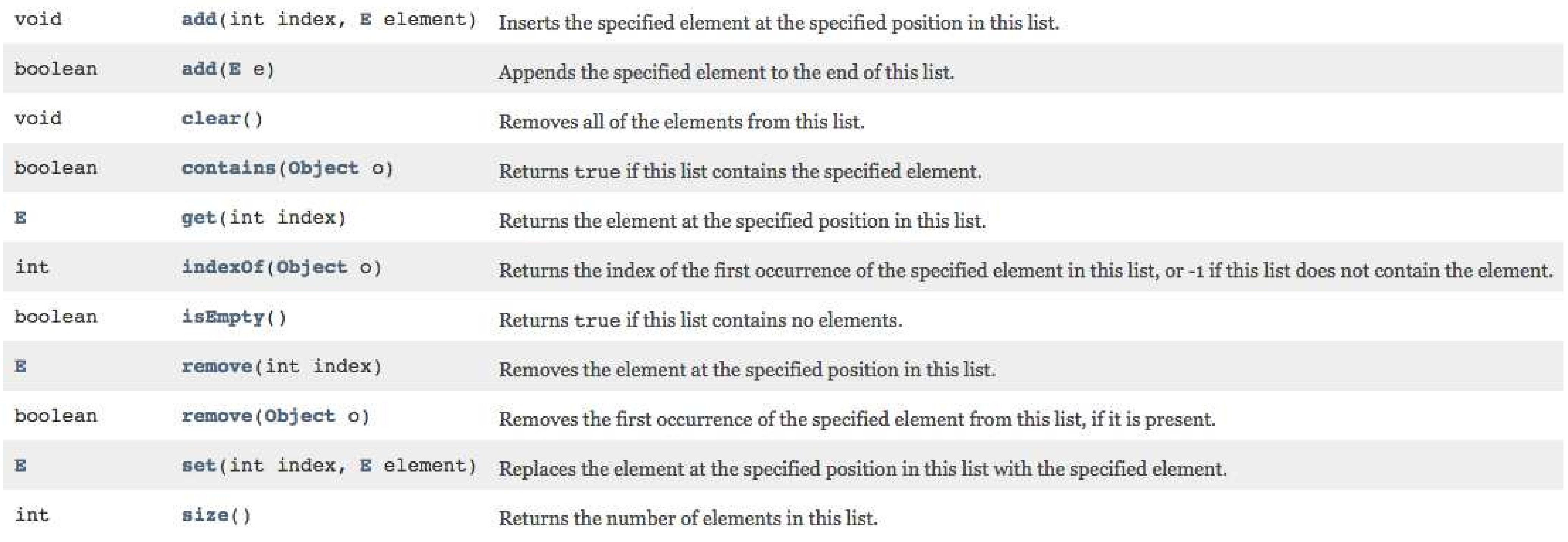
```
jshell> Circle c = new Circle(new Point(0.0, 0.0), 1.0)
c ==> Circle@(0.0, 0.0) with radius 1.0

jshell> Shape s = c
s ==> Circle@(0.0, 0.0) with radius 1.0

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

jshell> s.moveBy(1.0, 2.0) // moveBy is not specified in Shape
|  Error:
|  cannot find symbol
|    symbol:   method moveBy(double,double)
|  s.moveBy(1.0, 2.0)
|  ^------^

jshell> Movable m = c
m ==> Circle@(0.0, 0.0) with radius 1.0

jshell> m.moveBy(1.0, 2.0)
$.. ==> Circle@(1.0, 2.0) with radius 1.0

jshell> m.getArea() // getArea is not specified in Movable
|  Error:
|  cannot find symbol
|    symbol:   method getArea()
|  m.getArea()
|  ^-------^
```



- Exercise: make Rectangle movable

Java `List` Interface

- The Java API comprises many interfaces and classes
- Example: `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 throws exception (error): `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.java
      |        at (#1:1)
      ```

Java Collections Framework

- `List<E>` is an extension of parent interface `Collection<E>`

Interface	Description
Collection	The root interface in the collections hierarchy from which intefaces `List`, `Set`, `Queue`, ... are derived.
List	An ordered collection that can contain duplicate elements.
Set	A collection that does not contain duplicates.
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)`,

`Iterator` Interface

- Elements in a list can be looped successively (*iterable*)
- `Iterable` is the parent interface of `Collection`, and hence also the parent interface of `List`
  - `Iterable` interface specifies the `iterator()` method which returns an `Iterator`
    - `Iterator` is an interface that specifies the `next()` and `hasNext()` methods
- Any implementation of `List` (e.g. `ArrayList`) has to implement the `iterator()` method
  - `iterator()` returns an implementation of the `Iterator` interface that defines 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
```

List Sorting via the `Comparator` Interface

- There are many ways to sort a list / test if a list is sorted
- *Suppose* there is a `isSorted` method in `List`

```
boolean isSorted() {
    for (int i = 1; i < this.size(); i++) {
            if (/* how to compare this.get(i) with this.get(i-1)? */) {
        return false;
    }
    }
    return true;
}
```

- An implementation of a `Comparator<E>` interface is passed to the `sort` method
  - specifies `compare(x,y)` that returns `< 0` if x comes first; `> 0` if y comes first; or `0` if equal

```
boolean isSorted(Comparator<E> cmp) {
    for (int i = 1; i < this.size(); i++) {
        if (cmp.compare(this.get(i- 1), this.get(i)) > 0) {
            ...
```

Example: Sorting a List

- There is no `isSorted` method in `List`, but there is `sort`
  - e.g. 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 IntComp implements Comparator<Integer> {
   ...>    public int compare(Integer i, Integer j) {
   ...>       return i - j;
   ...>    }
   ...> }
|  created class IntComp

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

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

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

- How to sort
  - a list of integers in descending order?
  - a list of circles in increasing area?

Example: Sorting a List

- Sorting list of shapes in ascending order of area

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

jshell> shapes.add(new Rectangle(2.0, 3.0))
$.. ==> true

jshell> shapes.add(new Circle(1.0))
$.. ==> true

jshell> shapes
shapes ==> [Rectangle 2.0 x 3.0, Circle with radius 1.0]

jshell> shapes.sort(new ShapeAreaComp()) // how to define ShapeAreaComp()?

jshell> shapes
$.. ==> [Circle with radius 1.0, Rectangle 2.0 x 3.0] // state change!
```

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

```
jshell> ImList<Shape> shapes = new ImList<Shape>(). // using ImList
   ...> add(new Rectangle(2.0, 3.0)).
   ...> add(new Circle(1.0))
shapes ==> [Rectangle 2.0 x 3.0, Circle with radius 1.0]

jshell> shapes.sort(new ShapeAreaComp()) // creates a new sorted list
$.. ==> [Circle with radius 1.0, Rectangle 2.0 x 3.0]

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