

---

# CS2030 Lecture 5

## Java Generics

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

Semester 1 2022 / 2023

# Lecture Outline and Learning Outcomes

---

- Familiarity with the usage of a mutable `ArrayList` and how the delegation pattern is used to define an immutable list
- Understand **autoboxing and unboxing** of primitives and its wrapper classes
- Appreciate how **parametric polymorphism** supports the abstraction principle
- Be able to apply constructs involving Java **generics** to define generic classes
- Understand the implications of substitutability in generics and be able to apply upper- and lower- **bounded wildcards**

# Mutable ArrayList<E>

- ArrayList<E>: Java's mutable implementation of List<E>
  - type parameter E replaced with type argument to indicate the type of *elements* stored, e.g. ArrayList<String>
  - ArrayList<String> is a **parameterized type**

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

```
jshell> list.add("one")  
$.. ==> true
```

```
jshell> list.add("two")  
$.. ==> true
```

```
jshell> list.set(0, "three")  
$.. ==> "one"
```

```
jshell> list // ArrayList is mutable! :(  
list ==> [three, two]
```

# Type Arguments: Auto-boxing / Unboxing

- Only reference types allowed as type arguments; primitives need to be auto-boxed/unboxed, e.g. `ArrayList<Integer>`

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

```
jshell> list.add(1) // auto-boxing  
$.. ==> true
```

```
jshell> list.add(new Integer(2)) // explicit boxing  
$.. ==> true
```

```
jshell> int x = list.get(0) // auto-unboxing  
x ==> 1
```

- Placing a value of type `int` into `ArrayList<Integer>` causes it to be **auto-boxed**
- Getting a value out of `ArrayList<Integer>` results in a value of type `Integer`; assigning it to `int` variable causes it to be **auto-unboxed**

# (Immutable) Delegation Pattern: ImList

- Start by creating an immutable list *ImList of integers* by encapsulating a mutable `ArrayList` within the class
  - method implementations *delegated* to the `ArrayList`
  - create an empty `ImList`, or with elements from a `List`

```
import java.util.List;
import java.util.ArrayList;

class ImList {
    private final ArrayList<Integer> elems;

    ImList() { // creates an empty list
        this.elems = new ArrayList<Integer>();
    }

    ImList(List<Integer> elems) {
        this.elems = new ArrayList<Integer>(elems);
    }

    @Override
    public String toString() {
        return this.elems.toString();
    }
}
```

```
jshell> new ImList()
$.. ==> []

jshell> new ImList(List.of(1, 2))
$.. ==> [1, 2]
```

# ImList: get, size and isEmpty

- Define the get, size and isEmpty methods in ImList

```
Integer get(int index) {  
    return this.elems.get(index);  
}  
  
int size() {  
    return this.elems.size();  
}  
  
boolean isEmpty() {  
    return this.elems.isEmpty(); // or this.size() == 0  
}
```

```
jshell> new ImList().size()  
$.. ==> 0  
  
jshell> new ImList().isEmpty()  
$.. ==> true  
  
jshell> new ImList(List.of(1, 2, 3)).get(0)  
$.. ==> 1  
  
jshell> new ImList(List.of(1, 2, 3)).size()  
$.. ==> 3  
  
jshell> new ImList(List.of(1, 2, 3)).isEmpty()  
$.. ==> false
```

# ImList: add Method

- Define the add method which returns a new ImList
  - creates a copy of the original list before adding the element
  - uses the constructor that takes a List
    - ▷ an ArrayList is a List

```
ImList add(Integer elem) { // add elem to the back of a new elems
    ImList newList = new ImList(this.elems);
    newList.elems.add(elem);
    return newList;
}
```

```
jshell> ImList list12 = new ImList(List.of(1, 2))
list12 ==> [1, 2]
jshell> list12.add(3).add(4)
$.. ==> [1, 2, 3, 4]
jshell> list12.size()
$.. ==> 2
jshell> list12.add(3).size()
$.. ==> 3
jshell> list12
list12 ==> [1, 2] // ImList is immutable! :)
```

# Generic Class: `ImList<E>`

- **Generic `ImList<E>` class** to store elements of generic type `E`

```
import java.util.List;
import java.util.ArrayList;

class ImList<E> { // declare type parameter E
    private final ArrayList<E> elems;

    ImList() {
        this.elems = new ArrayList<E>();
    }

    ImList(List<E> elems) {
        this.elems = new ArrayList<E>(elems);
    }

    ImList<E> add(E elem) { // note return type of ImList<E>
        ImList<E> newList = new ImList<E>(this.elems);
        newList.elems.add(elem); // delegates add to ArrayList
        return newList;
    }

    E get(int index) {
        return this.elems.get(index);
    }
    ...
}
```



# Parametric Polymorphism

- Generic typing is also known as **parametric polymorphism**
- Like add method, set and remove can be similarly defined

```
ImList<E> set(int index, E elem) {  
    ImList<E> newList = new ImList<E>(this.elems);  
    newList.elems.set(index, elem);  
    return newList;  
}
```

```
ImList<E> remove(int index) {  
    ImList<E> newList = new ImList<E>(this.elems);  
    newList.elems.remove(index);  
    return newList;  
}
```

```
jshell> ImList<Integer> list12 = new ImList<Integer>(List.of(1, 2))  
list12 ==> [1, 2]  
jshell> list12.add(3).add(4).remove(2)  
$.. ==> [1, 2, 4]  
jshell> list12.add(3).add(4).remove(2).set(1, 5)  
$.. ==> [1, 5, 4]  
jshell> list12  
list12 ==> [1, 2]
```

# Generics and Substitutability

- `ImList<E>` can contain elements of type `T` or it's subclass `S`

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

- Are the following substitutable?

- `ImList<Shape> shapes = new ImList<Circle>(...)`
- `ImList<Circle> circles = new ImList<Shape>(...)`

- Generics is invariant\*; type parameters must match!

```
jshell> ImList<Shape> shapes = new ImList<Circle>()
Error:
incompatible types: ImList<Circle> cannot be converted to ImList<Shape>
ImList<Shape> shapes = new ImList<Circle>();
                        ^-----^
```

```
jshell> ImList<Circle> circles = new ImList<Shape>()
Error:
incompatible types: ImList<Shape> cannot be converted to ImList<Circle>
ImList<Circle> circles = new ImList<Shape>();
                        ^-----^
```

\* Given `S <: T`, neither `C<S> <: C<T>` (co-variance) nor `C<T> <: C<S>` (contra-variance) holds

# Upper Bounded Wildcard

- Define the `addAll` method that takes in elements of another `ImList` and adds to the end of the current `ImList`
- Suppose we have a `ImList<Shape>` object, what other types of `ImList` can `addAll` method take in?
  - another `ImList<Shape>`? Yes
  - `ImList<Circle>` or `ImList<Rectangle>`? Yes
  - `ImList<Object>`? No
- Use the upper bounded wildcard: `? extends T`

```
ImList<E> addAll(List<? extends E> list) {  
    ImList<E> newList = new ImList<E>(this.elems);  
    newList.elems.addAll(list);  
    return newList;  
}  
  
ImList<E> addAll(ImList<? extends E> list) {  
    return this.addAll(list.elems);  
}
```

`? extends` is *covariant*: if `S <: T`, then `C<S> <: C<? extends T>`

# ImList<E>: addAll Method

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

jshell> ImList<Rectangle> rects = new ImList<Rectangle>().
...> add(new Rectangle(4, 5))
rects ==> [Rectangle 4 x 5]

jshell> shapes.addAll(rects)
$.. ==> [Circle with radius 1, Rectangle 2 x 3, Rectangle 4 x 5]

jshell> shapes.addAll(shapes)
$.. ==> [Circle with radius 1, Rectangle 2 x 3, Circle with radius 1,
Rectangle 2 x 3]

jshell> ImList<Object> objs = new ImList<Object>().
...> add(new Circle(1)).
...> add("circle")
objs ==> [Circle with radius 1, circle]

jshell> shapes.addAll(objs)
| Error:
| incompatible types: ImList<java.lang.Object> cannot be converted to
| ImList<? extends Shape> shapes.addAll(objs)
|                                     ^__^
```

# List Sorting Using a Comparator

- Define a sort method that sorts the elements of `ImList<E>`  

```
ImList<E> 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)) > 0) {
                ...
            }
        }
    }
}
```
- 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

```
jshell> List<Integer> list = new ArrayList<Integer>(List.of(1, 2, 3))
list ==> [1, 2, 3]
jshell> class IntComp implements Comparator<Integer> {
    ...> public int compare(Integer i, Integer j) { return j - i; }
|   created class IntComp
jshell> new IntComp().compare(1, 2)
1
jshell> list.sort(new IntComp()) // ArrayList is mutable! :(
jshell> list
list ==> [3, 2, 1]
```

# Lower-Bounded Wildcard

- What are the possible ways to sort `ImList<Shape>`?
  - Sort by area of shape? Yes
  - Sort by radius of circles? No
  - Sort by length of `Object`'s `toString` method? Yes
- Use a lower bounded wildcard: `? super T`

```
import java.util.Comparator;

...
ImList<E> sort(Comparator<? super E> cmp) {
    ImList<E> newList = new ImList<E>(this.elems);
    newList.elems.sort(cmp);
    return newList;
}
```

- Notice that the actual sorting routine is delegated to the `ArrayList` where a similar `sort` method is defined

# ImmutableList<E>: sort Method

- Given shapes as an immutable list of type ImmutableList<Shape>

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

- Sorting by area of shape, i.e. via Comparator<Shape>

```
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> shapes.sort(new ShapeAreaComp())  
$.. ==> [Circle with radius 1, Rectangle 2 x 3]
```

# ImList<E>: sort Method

- Sorting by length of toString, i.e. via Comparator<Object>

```
jshell> class ObjectStringLengthComp implements Comparator<Object> {  
...>     public int compare(Object o1, Object o2) {  
...>         return o1.toString().length() - o2.toString().length();  
...>     }  
...> }  
| created class ObjectStringLengthComp  
  
jshell> shapes.sort(new ShapeAreaComp()).sort(new ObjectStringLengthComp())  
$.. ==> [Rectangle 2 x 3, Circle with radius 1]
```

- Sorting by radius if circle, i.e. via Comparator<Circle>

```
jshell> class CircleRadiusComp implements Comparator<Circle> {  
...>     public int compare(Circle c1, Circle c2) {  
...>         return c1.getRadius() - c2.getRadius(); // assuming Circle::getRadius() implemented  
...>     }  
...> }  
| created class CircleRadiusComp  
  
jshell> shapes.sort(new CircleRadiusComp())  
| Error:  
| incompatible types: CircleRadiusComp cannot be converted to java.util.Comparator<? super Shape>  
| shapes.sort(new CircleRadiusComp())  
|           ^-----^
```