
CS2030 Lecture 5

Java Generics

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

Semester 2 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
- Be able to define generic classes and generic methods
- 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
- 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**

Delegation Pattern: ImList

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

```
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);
    }
    ...
}
```


Generic Method

- Defining a generic method without associating to any object

```
jshell> <T> ImList<T> of(T t) { // return ImList<T> of one element
...>     return new ImList<T>().add(t);
...> }
| created method of(T)
jshell> of(1)
$.. ==> [1]
jshell> of("one")
$.. ==> [one]
```

- Generic methods are useful as static factory methods in a class

```
class ImList<E> {
    private final ArrayList<E> elems;

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

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

    static <E> ImList<E> of() { // note declaration of <E> for the method
        return new ImList<E>();
    }

    static <E> ImList<E> of(List<E> elems) { // note declaration of <E> for the method
        return new ImList<E>(elems);
    }
}

jshell> ImList.of() // type-inferred
$.. ==> []
jshell> ImList.<Integer>of() // type-witnessed
$.. ==> []
jshell> ImList.<Integer>of(List.of(1,2,3))
$.. ==> [1, 2, 3]
```

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);  
    if (index < this.size()) newList.elems.remove(index); // guard  
    return newList;  
}
```

```
jshell> ImList<Integer> list12 = ImList.<Integer>of(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]
```

- Exercise: define the iterator method

Generics and Substitutability

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

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

- Generics is invariant*; type parameters must match!

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

* 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 E`

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

jshell> ImList<Rectangle> rects = ImList.<Rectangle>of().
...> 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 = ImList.<Object>of().
...> 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)
|                                     ^_._^
```

- Likewise, use upper bounded wildcards in `ImList` constructor and `of` method that takes in a list, `List<? extends E> elems`

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

? **super** is *contravariant*: if `S <: T`, then `C<T> <: C<? super S>`

ImmutableList<E>: sort Method

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

```
jshell> ImmutableList<Shape> shapes = ImmutableList.<Shape>of().  
...> 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]
```

- Notice that ImmutableList::sort returns a new sorted list

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 of 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())  
|           ^-----^
```