CS2030 Lecture 7

Generics

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

Semester 1 2021 / 2022

Lecture Outline and Learning Outcomes

- Be able to apply constructs involving Java **generics** in defining generic classes and methods
- Understand how **cross-barrier manipulation** can be achieved by having the client define a functionality and passed across the barrier to the implementor for execution
- Be able to define **anonymous inner classes** and **lambdas**
- Be able to define implementations of the `Comparator`, `Predicate`, `Function`, `BiFunction` **functional interfaces** having a *single abstract method*
- Be able to use **wildcard** and **bounded wildcards**
- Appreciate the *Get–Put principle* and *PECS*

Generic Class

- Let's define our own generic `ImList` class that is immutable with tasks delegated to an internal mutable `ArrayList`

```java
class ImList<T> { // scope of T is within the class
    private final ArrayList<T> list;

    ImList() {
        this.list = new ArrayList<T>();
    }

    ImList<T> add(T elem) { // add elem to the back
        ImList<T> newList = new ImList<T>(this.list);
        newList.list.add(elem);
        return newList;
    }

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

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

Generic Method

- Type parameter can also be scoped within a method

```
jshell> <T> List<T> arrayToList(T[] arr) { // scoped of T is within method
   ...> List<T> list = new ArrayList<T>();
   ...> for (T t : arr) { list.add(t); }
   ...> return list; }
|  created method arrayToList(T[])

jshell> List<String> strings = arrayToList(new String[]{"one", "two", "three"})
strings ==> [one, two, three]
```

- **static** methods are also defined as generic methods

```
    static <U> ImList<U> of(U[] array) { // scope of U is within the method
        ImList<U> newList = new ImList<U>();
        for (U u : array) {
            newList.list.add(u);
        }
        return newList;
    }

jshell> ImList.of(new Integer[]{1,2,3}) // type inferred
$.. ==> [1, 2, 3]

jshell> ImList.<Integer>of(new Integer[]{1,2,3}) // type witness.. encouraged!
$.. ==> [1, 2, 3]
```

Cross-Barrier Manipulation

- **Cross-barrier manipulation** — where the client defines a *functionality* that is passed to the implementor for execution
- For example, just like `sort` method in `List`, the `sort` method in `ImList` takes in a `Comparator<T>`

```
ImList<T> sort(Comparator<T> comp) {
    ImList<T> newList = new ImList<T>(this.list);
    newList.list.sort(comp);
    return newList;
}
```

- Defining a comparator as a concrete class

```
jshell> ImList<Integer> list123 = new ImList.<Integer>of(new Integer[]{1, 2, 3}
list123 ==> [1, 2, 3]

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

jshell> list123.sort(new IntComp())
$.. ==> [3, 2, 1]
```

Anonymous Inner Class

- Using an *anonymous inner class* instead

```
jshell> list123.sort(new Comparator<Integer>() {
   ...>     @Override
   ...>     public int compare(Integer i, Integer j) {
   ...>         return j - i;
   ...>     }
   ...> })
$.. ==> [3, 2, 1]
```

- Which part of the anonymous inner class is *really* useful?
  - Interface name (`Comparator`) does not add value
  - `Comparator` is a SAM (*single abstract method*) interface
    - there is only one abstract method in `Comparator`
    - method name `compare` does not add value
- *Functional Interface* — interface with single abstract method

Lambda Expression

- Lambda syntax: `(`*parameterList*`) -> {`*statements*`}`
  - inferred parameter type with body:
    `(x, y) -> { double d = x * y; return d;}`
  - body contains a single expression: `(x, y) -> x * y`
  - only one parameter: `x -> 2 * x`
  - no parameter: `() -> System.out.println("Lambda!")`
- Most importantly, methods can now be treated as values! 😎
  - assign lambdas to variables
  - pass lambdas as arguments to other methods
  - return lambdas from methods

```
jshell> list123.sort((Integer i, Integer j) -> { return j - i; })
$.. ==> [3, 2, 1]

jshell> list123.sort((i, j) -> j - i)
$.. ==> [3, 2, 1]
```

Filtering a List Using Predicate<T>

- Predicate<T> with SAM: `boolean test(T t)`

```java
interface Predicate<T> { // java.util.function.Predicate
   boolean test(T t);
}
```

- Define the method `filter(Predicate<T> pred)` in `ImList`

```java
import java.util.function.Predicate;
    ...
    ImList<T> filter(Predicate<T> pred) {
        ImList<T> newList = new ImList<T>();
        for (int i = 0; i < this.list.size(); i++) {
            T elem = this.list.get(i);
            if (pred.test(elem)) {
                newList = newList.add(elem);
            }
        }
        return newList;
    }

jshell> list123.filter(x -> x % 2 == 1)
$.. ==> [1, 3]
```

Mapping a List Using Function<T,R>

- `Function<T,R>` with SAM: `R apply(T t)`

```
interface Function<T,R> { // part of java.util.function.Function
   R apply(T t);
}
```

- Define the method `map(Function<T,R> mapper)` in `ImList`

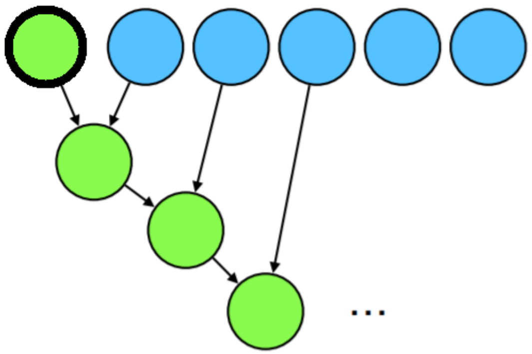
```
import java.util.function.BiFunction;
    ...
    <R> ImList<R> map(Function<T,R> mapper) { // scope of R is within method
        ImList<R> newList = new ImList<R>();
        for (int i = 0; i < this.list.size(); i++) {
            newList = newList.add(mapper.apply(this.list.get(i)));
        }
        return newList;
    }

jshell> ImList.<String>of(new String[]{"one", "TWO", "ThReE"}).
   ...> map(x -> x.toUpperCase()) // String -> String
$.. ==> [ONE, TWO, THREE]
jshell> ImList.<String>of(new String[]{"one", "TWO", "ThReE"}).
   ...> map(x -> x.length()) // String -> Integer
$.. ==> [3, 3, 5]
```

Reducing a List to a Value

- Iterate through elements of a list and reduce to a single value
- Uses `BiFunction<T,U,R>` with SAM: `R apply(T t, U u)`
- Define `reduce(U identity, BiFunction<U,T,U> mapper)` in `ImList`

```java
import java.util.function.BiFunction;
    ...
    <U> U reduce(U identity, BiFunction<U,T,U> acc) {
        for (int i = 0; i < this.list.size(); i++) {
            identity = acc.apply(identity, this.list.get(i));
        }
        return identity;
    }
```

```
jshell> ImList.<String>of(
   ...> new String[]{"one", "TWO", "ThReE"}).
   ...> reduce(0, (x,y) -> x + y.length())
$.. ==> 11
```



Substitutability Using Wildcard ?

- Consider defining an overriding `equals` method in `ImList`

```java
@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    } else if (obj instanceof ImList) {
        ImList<?> otherlist = (ImList<?>) obj;// ? is not a type!
        if (this.size() != otherlist.size()) {
            return false;
        } else {
            for (int i = 0; i < this.size(); i++) {
                // if (!this.get(i).equals(otherlist.get(i))) {
                if (!otherlist.get(i).equals(this.get(i))) {
                    return false;
                }
            }
            return true;
        }
    } else {
        return false;
    }
}
```

- `ImList<?> otherlist` can refer to *all* types of `ImList`s!
  - valid here because all types of objects can call `equals`

Bounded Wildcards

- Upper bounded wildcards: `? extends T`
  - All possible type `T` and its sub-types
- Lower bounded wildcards: `? super T`
  - All possible type `T` and its super-types
- Suppose we have the following classes:

```
class FastFood {
    int x;

    FastFood(int x) { this.x = x; }

    int fastfood() { return this.x; }

    public String toString() {
        return super.toString() +
            " " + this.x;
    }
}
```

```
class Burger extends FastFood {
    Burger(int x) {
        super(x);
    }

    int burger() { return this.x; }
}

class CheeseBurger extends Burger {
    CheeseBurger(int x) {
        super(x);
    }
}
```

Lower-Bounded Wildcards

- Consider filtering a list of burgers using `Predicate<Burger>`

```
jshell> ImList<Burger> burgers = new ImList<Burger>().
   ...> add(new Burger(1)).add(new CheeseBurger(2))
burgers ==> [Burger@210366b4 1, CheeseBurger@eec5a4a 2]

jshell> Predicate<Burger> pred = x -> x.burger() == 1
pred ==> $Lambda$15/0x00000001000af440@5b275dab

jshell> burgers.filter(pred)
$.. ==> [Burger@210366b4 1]
```

- What other predicates can we use to filter `burgers`?
  - `Predicate<FastFood>` since `Burger` is a `FastFood`
  - `Burger` is the lower bound
- Generalize the method signature of `ImList::filter` to

```
ImList<T> filter(Predicate<? super T> pred) {
```

```
jshell> burgers.filter((Predicate<FastFood>) x -> x.fastfood() == 2)
$.. ==> [CheeseBurger@eec5a4a 2]
```

Upper-Bounded Wildcard

- Consider extending a list with elements from another list

```
ImList<T> addAll(ImList<T> otherList) {
    ImList<T> newList = new ImList<T>(this.list);
    newList.list.addAll(otherList.list);
    return newList;
}
```

```
jshell> ImList<Burger> burgers = new ImList<Burger>().
   ...> add(new Burger(1)).add(new CheeseBurger(2))
burgers ==> [Burger@59f99ea 1, CheeseBurger@27efef64 2]

jshell> ImList<Burger> moreBurgers = new ImList<Burger>().
   ...> add(new Burger(3)).add(new CheeseBurger(4))
moreBurgers ==> [Burger@6f7fd0e6 3, CheeseBurger@47c62251 4]

jshell> burgers.addAll(moreBurgers)
$.. ==> [Burger@59f99ea 1, CheeseBurger@27efef64 2, Burger@6f7fd0e6 3,
CheeseBurger@47c62251 4]
```

- What other lists can we add to `burgers`?
  - `ImList<CheeseBurger>` since `CheeseBurger` is a `Burger`
  - `Burger` is the upper bound

Bounded Wildcards

- Generalize the method signature of `ImList::addAll` to

```
ImList<T> addAll(ImList<? extends T> otherList) {
```

```
jshell> ImList<CheeseBurger> cheeseBurgers = new ImList<CheeseBurger>().
   ...> add(new CheeseBurger(5)).add(new CheeseBurger(6))
cheeseBurgers ==> [CheeseBurger@66a3ffec 5, CheeseBurger@77caeb3e 6]

jshell> burgers.addAll(cheeseburgers)
$.. ==> [Burger@59f99ea 1, CheeseBurger@27efef64 2, CheeseBurger@66a3ffec 5,
CheeseBurger@77caeb3e 6]
```

- Exercise: how about the signature of `ImList::sort`?
- More general method signature of `ImList::map` is

```
<R> ImList<R> map(Function<? super T, ? extends R> mapper) {
```

- More general method signature of `ImList::reduce` is

```
<U> U reduce(U identity, BiFunction<U,? super T,U> acc) {
```

Variance of Types

- Let <: denote a sub-type (substitutability) relationship
  - Java arrays are *covariant*, $\texttt{C} <: \texttt{S} \Rightarrow \texttt{C[]} <: \texttt{S[]}$
  - Java generics is *invariant*,
    $\texttt{C} <: \texttt{S} \not\Rightarrow \texttt{ImList<C>} <: \texttt{ImList<S>}$ nor $\texttt{ImList<S>} <: \texttt{ImList<C>}$
  - Parameterized types are covariant,
    $\texttt{ArrayList} <: \texttt{List} \Rightarrow \texttt{ArrayList<C>} <: \texttt{List<C>}$
  - `? extends` is *covariant*
    $\texttt{C} <: \texttt{B} \implies \texttt{ImList<C>} <: \texttt{ImList<? extends B>}$
  - `? super` is *contravariant*
    $\texttt{B} <: \texttt{F} \implies \texttt{ImList<F>} <: \texttt{ImList<? super B>}$

- Get–Put Principle:
  - *covariant*: use **extends** to *get* items from a *producer**
  - *contravariant*: use **super** to *put* items into a *consumer**
  - *invariant*: use neither to get and put

***PECS**: Producer Extends Consumer Super