CS2030 Lecture 5

Generics and Variance of Types

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

Semester 1 2019 / 2020

Lecture Outline

- Abstraction principle again
- Generics
  - Generic classes
  - Auto-boxing and unboxing
  - Sub-typing and variance of generic types
    - Wildcards
    - Get–Put Principle / PECS
  - Generic methods
- Java Collections Framework
  - `Collection` / `List` interfaces
  - `Comparator` functional interface

Continuing on Our Abstraction Journey...

```java
public class CircleQueue {
    private Circle[] circles;
    private int front;
    private int back;

    public CircleQueue(int size) {
        circles = new Circle[++size];
        front = back = 0;
    }

    public int numOfCircles() {
        return back - front;
    }

    public boolean isFull() {
        return numOfCircles() ==
            circles.length - 1;
    }

    public boolean isEmpty() {
        return numOfCircles() == 0;
    }

    private int nextIndex(int index) {
        return (index + 1) % circles.length;
    }

    public void add(Circle circle) {
        if (!isFull()) {
            circles[back] = circle;
            back = nextIndex(back);
        } else {
            throw new IllegalStateException();
        }
    }

    public Circle remove() {
        Circle circle = null;
        if (!isEmpty()) {
            circle = circles[front];
            circles[front] = null;
            front = nextIndex(front);
        }
        return circle;
    }
}
```

- What if we now want a queue of points now?

Abstraction Principle Revisited

- Using the `Object` type

```
public class Queue {
    private Object[] elemts;
    private int front;
    private int back;

    public ObjectQueue(int size) {
        elemts = new Object[++size];
        front = back = 0;
    }

    public int numOfObjects() {
        return back - front;
    }

    public boolean isFull() {
        return numOfObjects() ==
            elemts.length - 1;
    }

    public boolean isEmpty() {
        return numOfObjects() == 0;
    }

    private int nextIndex(int index) {
        return (index + 1) % elemts.length;
    }

    public void add(Object elemt) {
        if (!isFull()) {
            elemts[back] = elemt;
            back = nextIndex(back);
        } else {
            throw new IllegalStateException();
        }
    }

    public Object remove() {
        Object elemt = null;
        if (!isEmpty()) {
            elemt = elemts[front];
            elemts[front] = null;
            front = nextIndex(front);
        }
        return elemt;
    }
}
```

Designing a “Generic” Queue

- The following program fragment results in a compilation error:

```
CircleQueue cq = new CircleQueue(10);
cq.add(new Circle(1.0));
cq.add(new Circle(2.0));
while (!cq.isEmpty()) {
    System.out.println(cq.remove().getArea());
}
```

- This indicates the need for an explicit type-cast

```
System.out.println(((Circle) q.remove()).getArea());
```

- What if rather than adding circles, `Point` objects are added to the queue?
  - There is no compilation error
  - But a runtime error `ClassCastException` ensues

Generic Type

"a type or method to operate on objects of various types while providing compile-time type safety"

```java
Queue<Circle> circleQueue = new Queue<Circle>()
```

- Generic typing is also known as **parametric polymorphism**
- For backward compatibility, Java implements generic typing via **type erasure**
  - The type argument is erased during compile time
  - Type parameter is replaced with either
    - `Object` if it is unbounded, or
    - the bound if it is bounded, more on this later...

Generic Type

- Implementing a **generic class** queue:

```
public class Queue<T> {
    private Object[] elements;
    private int front;
    private int back;

    public Queue(int size) {
        elements = new Object[++size];
        front = back = 0;
    }

    public int numOfObjects() {
        return back - front;
    }

    public boolean isFull() {
        return numOfObjects() ==
            elements.length - 1;
    }

    public boolean isEmpty() {
        return numOfObjects() == 0;
    }

    private int nextIndex(int index) {
        return (index + 1) % elements.length;
    }

    public void add(T element) {
        if (!isFull()) {
            elements[back] = element;
            back = nextIndex(back);
        } else {
            throw new IllegalStateException();
        }
    }

    public T remove() {
        Object element = null;
        if (!isEmpty()) {
            element = elements[front];
            elements[front] = null;
            front = nextIndex(front);
        }
        @SuppressWarnings("unchecked")
        T elem = (T) element;
        return elem;
    }
}
```

This prevents the warning from happening as , i the implementor have assured that the error will not happen

Auto-boxing and Unboxing

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

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

jshell> numbers.add(1)
$4 ==> true

jshell> numbers.add(0, 2)
$5 ==> true

jshell> for (int i : numbers) System.out.println(i * 10)
20
10
```

- Placing an `int` value into `ArrayList<Integer>` causes it to be **auto-boxed**
- Getting an `Integer` object out of `ArrayList<Integer>` causes the `int` value inside to be **(auto-)unboxed**

Java Collection: `ArrayList<T>`

- Java API provides **collections** to store related objects
  - provides methods that organize, store and retrieve data
  - there is no need to know how data is being stored
- Example, `ArrayList<T>`:

```java
import java.util.ArrayList;

class Queue<T> {
    private ArrayList<T> objects;
    private int maxObjects;

    public Queue(int size) {
        objects = new ArrayList<>();
        maxObjects = size;
    }

    public boolean isFull() {
        return maxObjects ==
                objects.size();
    }

    public boolean isEmpty() {
        return objects.isEmpty();
    }
```

```java
    public void add(T object) {
        if (!isFull()) {
            objects.add(object);
        } else {
            throw new IllegalStateException();
        }
    }

    public T remove() {
        if (!isEmpty()) {
            return objects.remove(0);
        }
        return null;
    }
}
```

T takes on the closest specification of type which
called from jshell or from main

Variance of Types

- Recall in LSP, if S is a sub-class of T, then object of type T can be replaced with that of type S without changing the desirable property of the program
- Moreover, S is a **sub-type** of T if a piece of code written for variables of type T can be safely used on variables of type S
- Let `S` and `T` represent classes or interfaces, and `S <: T` denote `S` being a sub-type of `T`
  - **Covariant**: `S[] <: T[]`

    ```
    Shape[] shapes = new Circle[10];
    ```

  - **Covariant**: `S<E> <: T<E>`

    ```
    List<Point> points = new ArrayList<Point>();
    ```

    The conventional way of declaring a arraylist
  - **Invariant**: Neither `C<S> <: C<T>` nor `C<T> <: C<S>`

    ```
    ArrayList<Shape> shapes = new ArrayList<Circle>(); //error
    ```

Wildcards

- Since neither `C<S> <: C<T>` (nor `C<T> <: C<S>`), a parameterized type is used with the same type argument, e.g. `ArrayList<Circle> circles = new ArrayList<Circle>(10);` or simply `ArrayList<Circle> circles = new ArrayList<>(10);`
- How do we then sub-type among generic types, in the spirit of `Shape[] shapes = new Circle[10]`;
- The answer is to use the wildcard `?` such as `ArrayList<?> anyList = new ArrayList<Circle>();`
- Even though `?` seems analogous to type `Object`, the **wildcard is not a type**
  - cannot declare a class of parameterized type `?`
  - use when specifying type of variable, field or parameter

Bounded Wildcards

- Suppose we have the following classes:
  - `public class FastFood`
  - `public class Burger extends FastFood`
  - `public class CheeseBurger extends Burger`
- Let's construct a method `getBurger`

```
static void getBurger(List<Burger> burgerProducer) {
    for (Burger burger : burgerProducer) {
        System.out.println(burger);
    }
}
```

- We can call the method as such

```
    List<Burger> burgers = new ArrayList<>();
    burgers.add(new Burger());
    :
    getBurger(burgers);
```

Upper-Bounded Wildcards

- Can we pass `List<FastFood>` or `List<CheeseBurger>` objects without changing the method body of `getBurger`?
  - Other than `Burger`, what other food can be a `Burger`?
    - ▷ A `CheeseBurger` is also a type of `Burger`

Anything that inherits from a Burger can be accepted

- So `Burger` can form an upper bound of the wildcard
- Change the parameterized type of the argument to

```
static void getBurger(List<? extends Burger> burgerProducer) {
    for (Burger burger : burgerProducer) {
        System.out.println(burger);
    }
}
```

- `? extends` is covariant:

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

Lower-Bounded Wildcards

- Now let's construct a method `putBurgers`

```
static void putBurger(List<Burger> burgerConsumer) {
    burgerConsumer.add(new Burger());
}
```

- Invoke the method as such

```
    List<Burger> burgers = new ArrayList<>();
    putBurgers(burgers);
```

- Can we pass `List<FastFood>` or `List<CheeseBurger>` objects without changing the method body of `putBurgers`?
  - Who, other than `Burger` consumers, like `Burgers`?
    - `FastFood` consumers also consume `Burgers`
- So `Burger` now forms a lower bound of the wildcard

Lower-Bounded Wildcards

- The only change needed is the the parameterized type

  Only Burgers and above are accepted

  ```
  static void putBurger(List<? super Burger> burgerConsumer) {
      burgerConsumer.add(new Burger());
  }
  ```

- `? super` is contravariant:

  if `S <: T`, then `C<T>` ~~`<:`~~ `:>` `C<? super S>`

- Can we change the method implementation to

  ```
  static void putBurger(List<? super Burger> burgerConsumer) {
      burgerConsumer.add(new FastFood());
  }
  ```

- What about a method that gets and puts into a `Burger` list?
  - Simply

    ```
    static void getAndPutBurger(List<Burger> burgers)
    ```

Get–Put Principle

- With wildcards, we can now do the following:

```
List<FastFood> fastFoodList = new ArrayList<>();
List<CheeseBurger> cheeseBurgerList = new ArrayList<>();

cheeseBurgerList.add(new CheeseBurger());
getBurger(cheeseBurgerList);

putBurger(fastFoodList);
System.out.println(fastFoodList);
```

- To summarize,
  - *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

Generic Methods

- Consider the following:

```
Integer[] nums = {19, 28, 37};
System.out.println(max3(nums));
```

- Other than using `Integer` class, can define generic methods

```
public static <T extends Comparable<T>> T max3(T[] nums) {
    T max = nums[0];

    if (nums[1].compareTo(max) > 0) {
        max = nums[1];
    }

    if (nums[2].compareTo(max) > 0) {
        max = nums[2];
    }

    return max;
}
```

Java Collections Framework

- Collections contain references to objects (elements) of type <E>, or objects of sub-type of <E>
- Collection-framework interfaces declare operations to be performed generically on various type of collections

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.

Java Collections Framework

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.
void	trimToSize()	Trims the capacity of this `ArrayList` instance to be the list's current size.

- Methods specified in interface `Collection<E>`
  - `size`, `isEmpty`, `contains`, `add(E)`, `remove(Object)`, `clear`
- Methods specified in interface `List<E>`
  - `indexOf`, `get`, `set`, `add(int, E)`, `remove(int)`,

`Collection<E>` Interface

- *Generic interface* parameterized with a type parameter `E`
- `toArray(T[])` is a generic method; the caller is responsible for passing the right type[1]
- `containsAll`, `removeAll`, and `retainAll` has parameter type `Collection<?>`, we can pass in a Collection of any reference type to check for equality
- `addAll` has parameter declared as `Collection <? extends E>`; we can only add elements that are upper-bounded by `E`

```java
public interface Collection<E>
        extends Iterable<E> {
    boolean add(E e);
    boolean contains(Object o);
    boolean remove(Object o);
    void clear();
    boolean isEmpty();
    int size();
    Object[] toArray();
    <T> T[] toArray(T[] a);
    boolean addAll(Collection<? extends E> c);
    boolean containsAll(Collection<?> c);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    :
}
```

[1] Otherwise, an ArrayStoreException will be thrown

`List<E>` Interface

- `List<E>` interface extends `Collection<E>`
  - For implementing a collection of possibly duplicate objects where element order matters
  - Classes that implement `List<E>` include `ArrayList` and `LinkedList`: `List<Circle> circles = new ArrayList<>();`
  - `circles` declared with `List<Circle>` to support possible future modifications to `LinkedList`
- `List<E>` interface also specifies a sort method

  `default void sort(Comparator<? super E> c)`
- Interface with `default` method indicates that `List<E>` comes with a default sort implementation
  - A class that implements the interface need not implement it again, unless the class wants to override the method

Comparator

- `sort` method takes in an object `c` with a generic **functional interface** `Comparator<? super E>`
  - `compare(o1, o2)` should return `0` if the two elements are equals, a negative integer if `o1` is "less than" `o2`, and a positive integer otherwise

```java
import java.util.Comparator;

public class NumberComparator implements Comparator<Integer> {
    @Override
    public int compare(Integer s1, Integer s2) {
        return s1 - s2;
    }
}


List<Integer> nums = new ArrayList<>();
nums.add(3);
nums.add(1);
nums.add(2);
nums.sort(new NumberComparator());
System.out.println(nums);
```

Anonymous Inner Class

- Rather than creating another class, use an anonymous inner class definition instead

```
List<Integer> nums = new ArrayList<>();
nums.add(3);
nums.add(1);
nums.add(2);
nums.sort(new Comparator<Integer>() {
    @Override
    public int compare(Integer s1, Integer s2) {
        return s1 - s2;
    }
});
System.out.println(nums);
```

ie a lambda function from python

- And it can potentially be defined even simpler...

Lecture Summary

- □ Appreciate the use of Java generics in classes and methods
- □ Understand autoboxing and unboxing involving primitives and its wrapper classes
- □ Understand parametric polymorphism and sub-typing mechanism, e.g. given `Burger <: FastFood`
  - covariant: `Burger[] <: FastFood[]`
  - covariant: `C<Burger> <: C <? extends FastFood>`
  - contravariant: `C<FastFood> <: C<? super Burger>`
  - invariant: Neither `C<Burger> <: C<FastFood>` nor `C<FastFood> <: C<Burger>`
- □ Appreciate the Get–Put principle and PECS
- □ Familiarity with the Java Collections Framework