CS2030 Lecture 6

Generics and Collections

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

Semester 1 2018 / 2019

Lecture Outline

- Abstraction principle
- Java Collection example: `ArrayList`
- Generics
  - Generic classes
  - Sub-typing
    - Wildcards
    - PECS
  - Generic methods
- Java Collections Framework
  - `Collection` / `List` interfaces
  - `Comparator` functional interface

Moving Up the Abstraction Ladder...

```
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

- Suppose we want to use class `Queue` to manage the following:

```
CircleQueue cq = new CircleQueue(10);
cq.add(new Circle(new Point(1, 1), 1));
cq.add(new Circle(new Point(2, 2), 2));
while (!cq.isEmpty()) {
    System.out.println(cq.remove().getArea());
}
```

- We require an explicit type-cast

```
Queue q = new Queue(10);
q.add(new Circle(new Point(1, 1), 1));
q.add(new Circle(new Point(2, 2), 2));
while (!q.isEmpty()) {
    System.out.println(((Circle) q.remove()).getArea());
}
```

- What if we add other shapes, i.e. `Circles`, `Rectangles`, etc. to `Queue`?

Collections and the `ArrayList`

- Java API provides **collections** to store groups of related objects together
  - provides methods that organize, store and retrieve data
  - there is no need to know how data is being stored
- `ArrayList<E>`: type parameter `E` replaced with type argument to support **parameterized types**, e.g. `ArrayList<Circle>`

```
ArrayList<Circle> = new ArrayList<Circle>();
```

- **Generic classes**: classes that allow some type parameter
- Convention: `T` for type; `E` for element; `K` for key; `V` for value
- Diamond notation `<>` lets the compiler infer the element type from the declaration; the following is equivalent

```
ArrayList<Circle> numbers = new ArrayList<>();
```

- Some commonly used methods of `ArrayList` include:

Collections and the `ArrayList`

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)`,

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**

Using ArrayList

- No explicit typecasting is needed

```
ArrayList<Circle> circleList = new ArrayList<>();
circleList.add(new Circle(new Point(1, 1), 1));
circleList.add(new Circle(new Point(2, 2), 2));
while (!circleList.isEmpty()) {
    System.out.println(circleList.remove(0).getArea());
}
```

- How about designing our own collection to support the following?

```
queue.add(new Circle(new Point(1, 1), 1));
queue.add(new Circle(new Point(2, 2), 2));
while (!queue.isEmpty()) {
    System.out.println(queue.remove().getArea());
}
```


Design Our Own Collection

- Cast non-generic type to generic: `(T[]) new Object[++size]`

```
class Queue<T> {
    private T[] elemts;
    private int front;
    private int back;

    @SuppressWarnings("unchecked")
    public Queue(int size) {
        elemts = (T[]) new Object[++size];
        front = back = 0;
    }

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

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

    public boolean isEmpty() {
        return numOfElements() == 0;
    }
```

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

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

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


Design Our Own Collection

- Alternative using `ArrayList`
- Generic typing is also known as **parametric polymorphism**

```
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();
    }
```

```
    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;
    }
}
```


From Sub–Class to Sub–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`
  - Is `S[] <: T[]`?
    e.g. `Shape[] shapes = new Circle[10];`
  - Is `S<E> <: T<E>`?
    e.g. `List<Point> points = new ArrayList<Point>();`
  - Is `C<S> <: C<T>`?
    e.g. `ArrayList<Shape> shapes = new ArrayList<Circle>();`

Wildcards

- Since neither `C<S> <: C<T>` (nor `C<T> <: C<S>`), a parameterized type must be used with the same type argument
  e.g. `ArrayList<Circle> circles = new ArrayList<Circle>(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 `readBurgers` in class `Main`

```
static void readBurger(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());
:
readBurger(burgers);
```


Upper-Bounded Wildcards

- Can we include `FastFood` or `CheeseBurger` objects without changing the method body of `readBurgers`?
  - That is to say, other than `Burger`, what other food can be a `Burger`?
    - A `CheeseBurger` is also a type of `Burger`
- So `Burger` can form an upper bound of the wildcard
- Change the parameterized type of the argument to

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

- `? extends Burger` means any type that extends from `Burger`, including itself

Lower-Bounded Wildcards

- Now let's construct a method `addBurgers` in class `Main`

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

- Invoke the method as such

```
ArrayList<Burger> burgerLovers = new ArrayList<>();
:
addBurgers(burgerLovers);
```

- Can we include `FastFood` or `CheeseBurger` objects without changing the method body of `addBurgers`?
  - In other words, other than `Burger` consumers, what other food consumers like `Burgers`?
    - `FastFood` consumers also consume `Burgers`

Lower-Bounded Wildcards

- So `Burger` now forms a lower bound of the wildcard
- The only change needed is the the parameterized type

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

- `? super Burger` means any type that `Burger` extends from (i.e. super-type of `Burger`), including itself
- The declaration `List<? super Burger> list` is not about the type of elements that can be assigned to the list, so `list.add(new Fastfood())` is wrong.
- Rather, it is about what type of list can take a `Burger` object.
- To summarize,
  - use `extends` to read items from a **producer** collection
  - use `super` to write items into a **consumer** collection

Producer Extends Consumer Super

- With wildcards, we can now do the following:

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

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

addBurger(fastFoodList);
System.out.println(fastFoodList);

$ java Main
CheeseBurger@e6ea0c6
[Burger@6a38e57f]
```

- What about a method that is both reads from and writes into a `Burger` list? Simply

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


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.

`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`

```
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

```
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);
```


Lecture Summary

- Appreciate higher-level abstraction thinking and design
- 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[]`
  - invariant: `C<Burger>` and `C<FastFood>`
  - covariant: `C<Burger> <: C <? extends FastFood>`
  - contravariant: `C<FastFood> <: C<? super Burger>`
- Appreciate PECS and accompanying notions of upper and lower bound wildcards
- Familiarity with the Java Collections Framework