CS2030 Lecture 5

Java Collections

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

Semester 1 2021 / 2022

Lecture Outline and Learning Outcomes

- Familiarity with the **Java Collections Framework**, particularly the `List` interface and `ArrayList` class
- Understand **autoboxing and unboxing** of primitives and its wrapper classes
- Appreciate how a list can be sorted using a `Comparator`, or by defining the natural order of the elements via the `Comparable` interface
- Be able to create packages and use the appropriate access modifiers

Java Collection: `ArrayList<T>`

- Java API provides **collections** to store related objects
  - abstraction: methods that organize, store and retrieve data
  - encapsulation: how data is being stored is hidden
- Example, `ArrayList<T>`
  - type parameter `T` 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") // ArrayList is Mutable!
$.. ==> true

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

jshell> for (String s : list) { System.out.println(s); }
one
two
```

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> 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**

Java Collections Framework

- Collections of type <E> 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

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

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.

`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` includes `ArrayList`, `LinkedList`, `Vector`, `Stack` (subclass of `Vector`), etc.
- `ArrayList<E>` is a sub-type of `List<E>`, so below is valid

```
List<Integer> list = new ArrayList<Integer>();
```

- Unlike arrays, given `S` is a sub-type of `T`, `ArrayList<S>` is **not** a sub-type of `ArrayList<T>`
  - `ArrayList<Object> list = new ArrayList<Integer>();` is invalid, even though `Integer` is a sub-type of `Object`
  - addresses the problem of heap pollution!

Storing Elements in an ArrayList

- Elements of sub-type `S` can be stored in an `ArrayList<T>`

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

jshell> list.add(1) // int autoboxed to Integer <: Object
$.. ==> true

jshell> list.add("one") // String <: Object
$.. ==> true

jshell> list
list ==> [1, "one"]
```

- Accessing the elements in the list

```
jshell> String foo(List<Object> list) {
   ...> String s = "";
   ...> for (Object obj : list) {  // syntactic sugar for iterators
   ...> s = s + "(" + obj + ")"; } // or obj.toString()
   ...> return s;
   ...> }
|  created method foo(List<Object>)

jshell> foo(list)
$.. ==> "(1)(one)"
```

Converting Between Arrays and Lists

- Converting from an array to a list

```
jshell> Integer[] arr = new Integer[]{1, 2, 3};
arr ==> Integer[3] { 1, 2, 3 }

jshell> List<Integer> list = Arrays.asList(arr);
list ==> [1, 2, 3]

jshell> List<Integer> list = Arrays.asList(1, 2, 3); // variable arguments
list ==> [1, 2, 3]
```

- Converting from a list to an array

```
jshell> list.toArray()
$.. ==> Object[3] { 1, 2, 3 }

jshell> list.toArray(new Integer[0]) // list.toArray(Integer[]::new)
$.. ==> Integer[3] { 1, 2, 3 }
```

- Converting from a primitive array to a list requires every element to be boxed; either write a loop or use streams!

```
jshell> int[] arr = new int[]{1, 2, 3}
arr ==> int[3] { 1, 2, 3 }

jshell> Arrays.stream(arr).boxed().collect(Collectors.toList())
$.. ==> [1, 2, 3]

jshell> Arrays.stream(arr).boxed().toArray(Integer[]::new)
$.. ==> Integer[3] { 1, 2, 3 }
```

List Sorting with a `Comparator`

- `List<E>` interface defines a `sort` method

```
default void sort(Comparator<? super E> c)
```

- Interface with `default` method indicates that `List<E>` comes with a default sort implementation (an impure interface)
  - A class that implements the interface need not implement `sort` again, unless the class wants to override the method
- `sort` takes in a `Comparator` object with a `compare` method

```
jshell> List<Integer> list = Arrays.asList(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> list.sort(new IntComp())

jshell> list
list ==> [3, 2, 1]
```

Sorting via Natural Order

- *Natural order* of elements of type `T` is defined by implementing the `compareTo` method (of the `Comparable<T>` interface)

```
class Circle implements Comparable<Circle> {
    private double radius;

    Circle(double radius) {
        this.radius = radius;
    }

    @Override
    public int compareTo(Circle other) {
        if (this.radius < other.radius) {
            return -1;
        } else if (this.radius > other.radius) {
            return 1;
        } else {
            return 0;
        }
    }

    @Override
    public String toString() {
        return "Circle with radius " + radius;
    }
}
```

Sorting via Natural Order

- Clearly, there can only be one `compareTo` method, and hence one natural order, defined in the `Circle` class
- Passing `null` into the sort method
  - sorts the elements based on the natural order as defined in the element's `compareTo` method

```
jshell> List<Circle> list = Arrays.asList(new Circle(2.0), new Circle(1.0))
list ==> [Circle with radius 2.0, Circle with radius 1.0]

jshell> list.sort(null)

jshell> list
list ==> [Circle with radius 1.0, Circle with radius 2.0]
```

- Once the `sort` method is invoked, the list changes state
  - `sort` has a **`void`** return type
  - Lists are *mutable* data structures!

Packages

- The Java API is organized into packages; we can organize our classes into packages too
- When discussing the abstraction barrier, we have been using **`private`** and default modifiers
- Other than these, there are **`protected`** and **`public`** modifiers
- Java adopts a **`package`** abstraction mechanism that allows the grouping of relevant classes/interfaces together under a *namespace*, just like `java.lang`
- The access level (most restrictive first) is given as follows:
  - **`private`** (visible to the class only)
  - default (visible to the package)
  - **`protected`** (visible to the package and all sub-classes)
  - **`public`** (visible to the world)

Creating Packages

- Include the **`package`** statement at the top of all source files that reside within the package, e.g.

  ```
  package cs2030.test;
  ```

- Include the **`import`** statement to source files outside the package, e.g.

  ```
  import cs2030.test.SomeClass;
  ```

- Compile the Java files using

  ```
  $ javac -d . *.java
  ```

- `cs2030/test` directory created with same-package class files stored within

Creating Packages

```
==> Base.java <==
package cs2030.test;
public class Base {
    private void foo() { }
    protected void bar() { }
    void baz() { }
    public void qux() { }
    private void test() {
        this.foo();
        this.bar();
        this.baz();
        this.qux();
    }
```

```
==> InsidePackageClient.java <==
package cs2030.test;
class InsidePackageClient {
    private void test() {
        Base b = new Base();
        b.bar();
        b.baz();
        b.qux();
    }
}
```

```
==> InsidePackageSubClass.java <==
package cs2030.test;
class InsidePackageSubClass extends Base {
    private void test() {
        super.bar();
        super.baz();
        super.qux();
    }
}
```

```
==> OutsidePackageClient.java <==
import cs2030.test.Base;
class OutsidePackageClient {
    private void test() {
        Base b = new Base();
        b.qux();
    }
}
```

```
==> OutsidePackageSubClass.java <==
import cs2030.test.Base;
class OutsidePackageSubClass extends Base {
    private void test() {
        super.bar();
        super.qux();
    }
}
```

Access Modifiers and Their Accessibility

Access Modifiers ->	private	Default/no-access	protected	public
Inside class	Y	Y	Y	Y
Same Package Class	N	Y	Y	Y
Same Package Sub-Class	N	Y	Y	Y
Other Package Class	N	N	N	Y
Other Package Sub-Class	N	N	Y	Y