

CS1020: DATA STRUCTURES AND ALGORITHMS I

Tutorial 3 – Collection, Array-Based List, Generics

(Week 5, starting 8 February 2016)

1. *Collection*<E> vs *List*<E>

Java defines 2 interfaces: `java.util.Collection` and `java.util.List`.

Collection: <http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html>

List: <http://docs.oracle.com/javase/7/docs/api/java/util/List.html>

Each interface is an abstraction – an idea without any implementation – and can be used as a data type.

(a) What is the relationship between a *Collection* and a *List*?

(b) What do they have in common?

(c) What extra functionality does *List* have?

Tip: Learn to compare, contrast, and draw links between different concepts/topics you've learnt! Hopefully, by the end of the semester, you would have mapped out and understood the relationships between all the data structures covered in this module.

Answers

(a)

A *List* **inherits** *Collection*. Every *List* **is a** *Collection* but not the other way round. Therefore, we expect a *List* to have additional functionality.

(b)

They both use **generics**. A *Collection* has `size()` method, returning the **number of elements** within it, and `isEmpty()` method. It typically has `add(E)`, `remove(Object)`, and `contains(Object)` methods. It also has **iterators** to move through all elements and possibly delete data.

(c)

Lists have a notion of sequence/ordering. Therefore, they typically provide additional functionality for **insertion/removal/search at particular indexes**, through `add(int, E)`, `remove(int)` and `indexOf(Object)` respectively. They also provide `get(int)` and `set(int, E)` methods for **access/modification** at a particular index.

Since there is a concept of ordering, Lists provide a **ListIterator**, which can move through all elements in **both directions**. The `ListIterator` *may possibly* allow insertion and modification, besides deletion.

2. *ArrayList<E> / Vector<E>*

A very “simple” way to compress a string of characters is to **group adjacent identical characters** into a unit. Each unit contains two pieces of information: the character, and its number of occurrences. In this way, much space is saved if the original string contains many duplicated characters side-by-side.

```
class Pair <T, S> {
    private T _objLeft;
    private S _objRight;

    public Pair(T pObjLeft, S pObjRight) {
        _objLeft = pObjLeft;
        _objRight = pObjRight;
    }
    public T getLeft() { return _objLeft; }
    public S getRight() { return _objRight; }
    public String toString() { // aids debugging
        return "(" + _objLeft + "," + _objRight + ")";
    }
}
```

- (a) Complete the method **compressString**(String input). The method returns an ArrayList containing pairs of (character, number of occurrences) in sequence.

For example, if we call

```
result = compressString("AAAAABBBAA");
```

result should be an ArrayList that contains three Pair references: {(‘A’, 5), (‘B’, 3), (‘A’, 2)}

```
class Zip {
    public _____ compressString(String input) {
        /* TODO: compress input and return ArrayList of pairs */
    }
}
```

- (b) Can we replace all occurrences of ArrayList<...> with Vector<...>?

Answers

(a)

```
public ArrayList<Pair<Character, Integer>> compressString(String input){

    ArrayList<Pair<Character, Integer>> pairList =
        new ArrayList<Pair<Character, Integer>>();

    if (input.length() == 0) // if no groups, return empty ArrayList
        return pairList;

    char toAdd = input.charAt(0);
    int counter = 1;
    for (int inputIdx = 1; inputIdx < input.length(); inputIdx++) {
        char curr = input.charAt(inputIdx);
        if (curr != toAdd) { // if new grouping found, add previous grp
            pairList.add(new Pair<Character, Integer>(toAdd, counter));
            toAdd = curr;
            counter = 0;
        }
        counter++;
    } // finally, add last group

    pairList.add(new Pair<Character, Integer>(toAdd, counter));
    return pairList;
}
```

(b)

The method signatures (and return types) of `ArrayList` and `Vector` are almost identical. Both implement the `List` interface. Since we **only use** the `add(E e)` **method from the `List` interface** here, `ArrayList` and `Vector` can be **interchanged without affecting the correctness** of the program.

There are **some differences** between `Vector` and `ArrayList`¹. We seldom need to use `Vector`², so you may want to learn `ArrayList` well, and just note the differences in the `Vector` class.

¹ See accepted answer of <http://stackoverflow.com/questions/2986296/what-are-the-differences-between-arraylist-and-vector> for the main differences, and the API docs for different methods, e.g. `capacity()`

² *Extra:* See accepted answer to <http://stackoverflow.com/questions/1386275/why-is-java-vector-class-considered-obsolete-or-deprecated> to find out why `Vector` may not be useful even in a multithreaded program

3. Generics

Examine the following classes and their relationships:

```
class Animal { ... }
class Flyer extends Animal { ... }
class Glider extends Flyer { ... }

class Writing<T> { ... }
class Book<T> extends Writing<T> { ... }
class Birdopedia extends Book<Flyer> { ... }
```

(a) Regarding generics, where should angled brackets <> appear, and where should they NOT appear?

Answer

They should appear when:

- **Defining** the class
- Inheriting from a generic class
- **Declaring a reference** to a generic class
- **Instantiating an object** that is of a generic class

Generally speaking, you should ensure you have not missed out angled brackets in those 3 places.

They should **NOT** appear when:

- Defining the generic class' **constructor**.

For example, in question 2, the constructor of Pair is:

```
public Pair(T pObjLeft, S pObjRight) {
```

and not

```
public Pair<...>(T pObjLeft, S pObjRight) {
```

- Using the generic type parameter as a datatype within the class that the type parameter is defined in.

For example,

```
class Zoo <E extends Animal> { // Defining the class, yes.

    private E[] animals; // No angled brackets here, because you
                        // are using the type parameter and not
                        // declaring one.
```

(b) Are each of these cases, independently, allowed?

Answers

	Y/N	Why or Why Not
Type specification		
Book book = new Book(...);	Yes but...	Bad practice. These are termed “raw types”, allowed for compatibility reasons. In effect, this defeats the purpose of using generics in the first place, because Book book is equivalent to Book<Object> book. The second example may cause problems when writing.
Book book = new Book<Animal>(...);		
Book<Animal> book = new Book(...);	Warn ...	Bad practice. The object data type is equivalent to Book<Object>. This example may cause problems when reading.
Book<Animal> book = new Book<Animal>(...);	Yes	A (Book<Animal>) reference points to a (Book<Animal>) object.
	Y/N	Why or Why Not
Typing relationship		
Book<Flyer> book = new Book<Animal>(...);	No	Error. Flyer is a subclass (subtype) of Animal, but Book<Flyer> is NOT a subtype of Book<Animal>, nor the other way round. Allowing such relationships may cause read or write problems respectively. <i>Read the table on the next page if interested</i>
Book<Animal> book = new Book<Flyer>(...);		
	Y/N	Why or Why Not
Inheritance as learned in lectures		
Writing<Flyer> writing = new Book<Flyer>(...);	Yes	Inheritance relationship in action. For all types X, a Book<X> is a Writing<X>, but not the other way round, so a Book<X> can take the place where a Writing<X> is expected.
Book<Flyer> book = new Writing<Flyer>(...);	No	
Inheriting generic class with type specified in class definition		
Book<Flyer> book = new Birdopedia(...);	Yes	Due to the definition of the Birdopedia class: Every Birdopedia is a Book<Flyer> and Writing<Flyer> Every Birdopedia is NOT a Book<Animal> Every Birdopedia is NOT a Book<Glider>

Extra: For those of you who want to **learn more**, what happens **IF** these were allowed?

(Java and C++ do not allow such typing relationships to exist for generics, but some other languages do.)

Relationship	Case	Prob
<code>Book<Flyer> book = new Book<Animal>(...);</code> Contravariant relationship Relationship is in the opposite direction as inheritance	Read	Book<T> may have an instance method <code>T find(...)</code>. <code>book.find(...)</code> would then return the address of a Flyer, when the Book may actually contain other animals (Animal not always a Flyer). On the other hand, writes are safe because all Flyers to be written (assuming <code>add(T chapter)</code>) are Animals.
<code>Book<Animal> book = new Book<Flyer>(...);</code> Covariant relationship Relationship is in the same direction as inheritance	Write	Book<T> may have a method <code>add(T chapter)</code> that adds a T object to a book's data. This means that, if we have <code>Animal c</code>, a cat, then <code>book.add(c)</code> adds a cat to a book of flyers (Animal not always a Flyer). On the other hand, reads are safe because all Flyer objects will be retrieved with Animal references (assuming <code>T find(...)</code>).

- Learn how to learn 🧠 -

Revise a concept
Think of a few cases
Test your understanding
Prepare well for tutorial