1 The Java Collections Library

In real day-to-day Java programming, programmers do not usually create their own dynamic structures. The Java core library provides several structures that are fit for most purposes. Now that you understand pointers, inheritance, and generics, it is the right time to introduce them.

The so-called Java Collections Library is a series of interfaces and classes in package `java.util`. In order to use them, you will need to import them. For example, to use interface `List` and class `ArrayList` you will need to add to the beginning of your class file the following lines:

```
import java.util.ArrayList;
import java.util.List;
```

The benefits of using the dynamic structures from the Java Collections Library are twofold. First, it will save effort on your part because you do not have to create these structures again and again. Second, and more important, it will make it easier for your code to communicate with other people's code because you will be using the same interfaces and classes.

Important: where to get more information?

The Java Collections Library is well documented on the main JavaDoc. You can find it by looking up "java collections" in your favourite search engine. You can also look up specific classes or interfaces easily, e.g. "java list" will lead you to the documentation on the `List` interface.

1.1 Collection

A `Collection<E>` is just a group of elements. This is the most general interface. This interface is extended by `List<E>`, `Set<E>`, and `Queue<E>`.

1.2 Set

A `Set<E>` is a collection that cannot contain duplicate elements. If an element is added to a set that already contains it, nothing happens. The most commonly used implementation is `HashSet<E>`. A typical idiom to create a new set is:

```
Set<MyClass> classSet = new HashSet<MyClass>();
```

When the order of the elements is important (e.g. for iterating through the interface, see below), a useful implementing class may be `TreeSet<E>`.

1.3 List

A `List<E>` is a collection of elements, maybe with duplicate elements. A list provides methods to access elements at specific indeces (`.get(int)`) and to search for the index of an element (e.g. `indexOf(Object)`). A list usually keeps the order in which elements are added to it; This means that elements that were added earlier come before elements that were added later.

The most commonly-used implementation is `ArrayList<E>`, although `LinkedList<E>` may be useful in some situations. A typical idiom to create a new list is:

```
List<MyClass> classList = new ArrayList<MyClass>();
```

There is also a class `Vector<E>` that is an old, synchronised, more complicated implementation of interface `List<E>`. It should not be used (except maybe in multi-threaded applications). `Vector<E>` was part of Java 1.0[1]; since the Collections Library was introduced in Java 2, `ArrayList<E>` (and `LinkedList<E>`) is a better fit for most situations.

[1] Strictly speaking, it was Vector because Java 1.0 did not have generics.

1.4 Queue

A `Queue<E>` is a FIFO structure in which the first elements coming in are the first elements coming out. The exception to this rule may be the use of priorities, which allows some elements to come out before other elements that have been longer in the queue. The most common implementations of queues are `LinkedList<E>` and `PriorityQueue<E>`. A typical idiom to create a new queue is:

```
Queue<MyClass> classQueue = new LinkedList<MyClass>();
```

1.5 Stack

A `Stack<E>` is a LIFO structure in which the last element coming in is the first element coming out.

`Stack<E>` is a special case in Java because they existed before the Collections Library was introduced in Java 2 (like `Vector<E>`, its parent class) and it is a specific class, instead of being an interface that is implemented by one or more classes. A typical idiom to create a new stack is:

```
Stack<MyClass> classStack = new Stack<MyClass>();
```

1.6 Map

A `Map<K,V>` is an object that links pairs of keys and values. In the Java Collections Library, a map cannot link the same key to two different values, i.e. it cannot contain duplicate keys. The most typical implementation is `HashMap<K,V>` (in some cases, a `TreeMap` may be useful, e.g. if the order of keys is important). A typical idiom to create a new map is:

```
Map<KeyClass,ValueClass> classMap = new HashMap<KeyClass,ValueClass>();
```

1.7 Operations with collections

All operations with collections are documented in their respective JavaDocs. The operations that add or remove elements from the collection are those most commonly used. There are also methods to add or remove a lot of elements in bulk (e.g. add all the element in a set to a list), and to convert from collections to arrays and viceversa. You should look a the JavaDoc and get familiarised with them. We are going to comment here only on two special cases, but very important to know.

1.7.1 Iterators

Every collection implements the method `iterator()`. This method returns an `Iterator<T>` for the collection.

An iterator can be used to move through the collection. Iterators implement two (sometimes three) simple methods:

hasNext() Returns true if there are more elements to go through.

next() Moves to the next element.

A typical idiom to use iterators is as follows:

```
// Assuming there is a collection (list, set, etc) of MyClass called "elements"
for (Iterator itr = elements.iterator(); itr.hasNext(); ) {
    MyClass next = itr.next();
    next.doSomething();
}
```

First, you get an iterator for your collection. Then, while there are more elements to look at, you repeat the loop: take next element, do something with it, then go for the next...

1.7.2 for-each

Since Java 5, the above loop can be written more succintly as:

```
// Assuming there is a collection (list, set, etc) of MyClass called "elements"
for (MyClass next : elements) {
    next.doSomething();
}
```

This is read "for each object of class MyClass in 'elements', do...". Much shorter and clearer, isn't it?

It is important to bear in mind, though, that there is a bit of magic happening behind the scenes when you use a for-each loop. The objects on the right-hand side of the colon (":") must implement the `Iterable` interface or be an array. If it is an `Iterable` (any class, as long as it implements that interface), then Java will convert this construct into the idiom in Section 1.7.1; if it is an array, Java will convert it into a loop that uses an integer index to traverse it.