Programming in Java (23/24) – Day 9: Java Collections, more on generic types

1 The Java Collections Library

In real day-to-day Java programming, programmers do not usually create their own container data structures. The Java core library provides several structures that are fit for most purposes where some kind of container data structure is needed. Now that you understand pointers, inheritance, and generics, it is the right time to introduce them. Such container structures are perhaps the most prominent use case for interfaces and classes with generic types.

The *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 the following lines to the beginning of your class or interface file:

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

The benefits of using the container data 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 perhaps more importantly, 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.

Information for Java 17 is available here:

`https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/doc-files/coll-overview.html`

You can also look up specific classes or interfaces easily, e.g., "java list" will lead you to the documentation on the `List` interface.

2 Collections: overview

For an overview of several central interfaces and classes of the Collections framework, consider the *class diagram* in Figure 1,[1] using a notation very close to UML class diagrams.

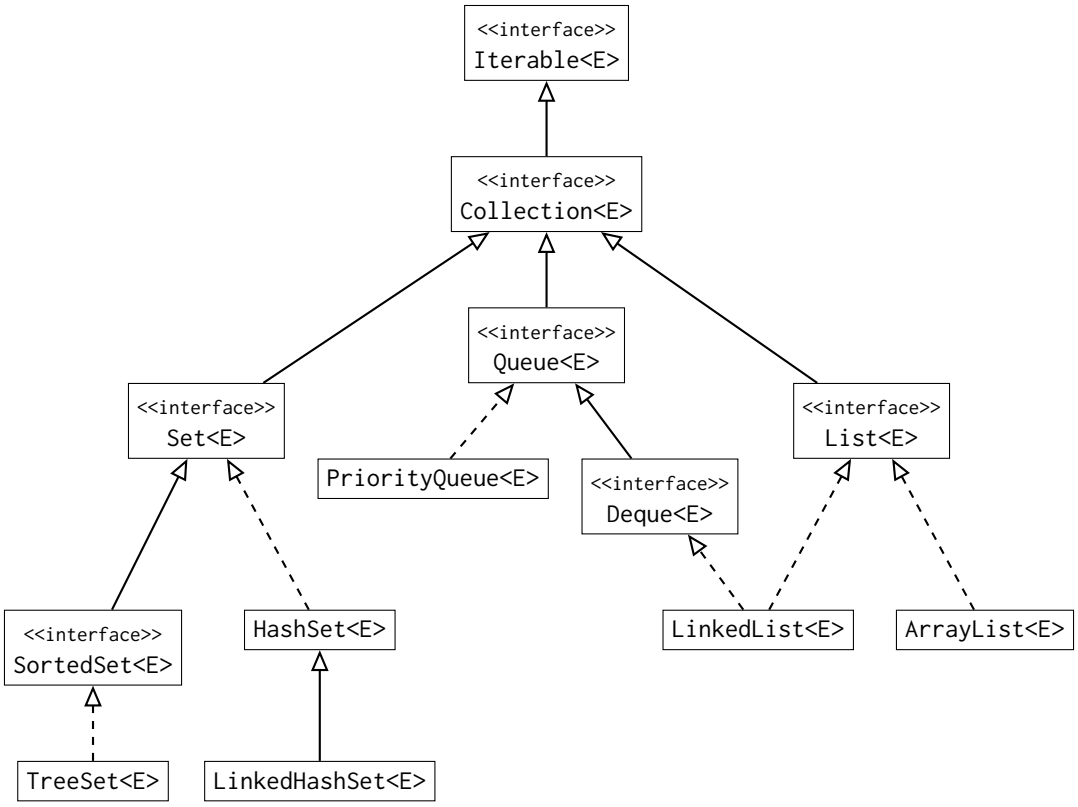

Figure 1: Several central interfaces and classes of the Java Collections framework using a UML-like notation.

As on earlier days, here a box represents either a class or an interface. If it is an interface, the box is labelled with `<<interface>>`. A solid arrow from a box to another box represents the "extends" relation, like the Java keyword **`extends`**. For example, class `LinkedHashSet<E>` extends class `HashSet<E>`, and interface `List<E>` extends interface `Collection<E>`. A dashed arrow from a box to another box represents the "implements" relation, like the Java keyword **`implements`**. For example, class `ArrayList<E>` implements interface `List<E>`.

[1] This diagram is necessarily incomplete. Specifically, it omits a number of abstract classes such as `AbstractCollection<E>`, `AbstractList<E>`, or `AbstractSet<E>` that provide part of the implementation of the concrete subclasses.

2.1 Iterable

The most general interface in our hierarchy is `Iterable<E>`. It provides a method `iterator()` that returns an object of type `Iterator<E>`. An `Iterator<E>` is an object that lets you visit elements of type `E` one by one. Usually these elements are taken from a container that can hold several elements, such as a `Collection<E>`. There is a separate interface `Iterable<E>` because an `Iterator<E>` does not *have* to be backed by a collection – we could also have an `Iterator<BigInteger>` that lets us visit all of the infinitely many prime numbers in ascending order, one by one, and that does the calculations for the next prime number on the fly.

However, visiting elements in a collection is probably the most typical use case for iterators. Iterators implement two (sometimes three) simple methods:

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

next() Moves to the next element and returns it.

A typical idiom (i.e., "the way it is usually said/done") to use iterators is as follows:

```
1    // Assuming there is a collection (list, set, etc) of MyClass called "elements"
2    for (Iterator<MyClass> iter = elements.iterator(); iter.hasNext(); ) {
3        MyClass next = iter.next();
4        next.doSomething();
5    }
```

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...

2.1.1 for-each

The above kind of loop was very common in Java up to version 4.[2] Since Java 5, the above loop is usually written more succinctly as:

```
1        // Assuming there is a collection (list, set, etc) of MyClass called "elements"
2        for (MyClass next : elements) {
3            next.doSomething();
4        }
```

This is read "for each object 'next' 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<MyClass>` interface or be an array. If it is an `Iterable<MyClass>` (any class, as long as it implements that `interface`), then Java will convert this construct into the idiom at the beginning of Section 2.1. If it is an array, Java will convert it into a loop that uses an integer index to traverse the array from index `0` all the way to index `length - 1`.

[2] Technically, the version number was 1.4 – it took Java's version numbering scheme a while to stabilise.

2.1.2 Deleting with iterators

When we are iterating over a collection, we may sometimes want to remove elements from that collection. The for-each loop lets us visit elements one by one, but it does not let us delete elements. Here we need to use a conventional iterator that also provides an implementation of the following method:

remove() Removes the element returned by the last call to `next()` on this iterator from the underlying collection.

```
1    // Assuming there is a collection (list, set, etc) of MyClass called "elements"
2    for (Iterator<MyClass> iter = elements.iterator(); iter.hasNext(); ) {
3        MyClass next = iter.next();
4        // Assuming we have a method shouldBeRemoved(MyClass)
5        // that returns whether we want to remove an element
6        if (shouldBeRemoved(next)) {
7            iter.remove();
8        }
9    }
```

It is important that you use the iterator to delete the element rather than modifying `elements` directly by using a mutator method of `elements`. The reason is that iterators usually stop working correctly if their underlying data structure gets modified "behind their back".

2.2 Collection

A `Collection<E>` is just a group of elements. This is the most general interface for our container data structures. It has methods to add and remove elements, to check if an element is present, to check a collection's size, and to convert it to an array. The `Collection<E>` interface is extended by `List<E>`, `Queue<E>`, and `Set<E>`.

2.3 List

A `List<E>` is a collection of elements, maybe with duplicate elements. This means that a list `[1]` and a list `[1,1]` are considered to be different by the `equals(Object)` method of implementations of interface `List<E>`. The order of elements also matters: a list `[1,2]` and a list `[2,1]` are considered to be different as well. A list provides methods to access elements at specific indices (`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. It is also possible to add elements at specific positions rather than at the end.

The most commonly-used implementation is `ArrayList<E>`, although `LinkedList<E>` may be useful in some situations where we want to add and remove at both ends of the list. A typical idiom to create a new list is:

```
1        List<MyClass> myList = new ArrayList<MyClass>();
```

Recall that since Java 7 we can instead just write:

```
1        List<MyClass> myList = new ArrayList<>();
```

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;[3] since the Collections Library was introduced in Java 2, `ArrayList<E>` (and `LinkedList<E>`) is a better fit for most situations.

Most implementations of the `List<E>` interface do not have specific requirements on their element type `E`. They normally use the method `equals(Object)` for type `E` (either inherited or overridden, but always available) to check whether two objects are "the same" or "different".

2.4 Queue

A `Queue<E>` is a FIFO (First In, First Out) structure in which the first elements going in are the first elements going 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:

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

2.5 Deque

A `Deque<E>` ("double-ended queue") allows for inserting and removing elements at both ends of the queue. It can be used not only as a queue, but also as a stack. A stack is a LIFO (Last In, First Out) structure in which the last element going in is the first element going out.[4] A typical idiom to create a new deque is:

```
1       Deque<MyClass> classStack = new LinkedList<>();
```

2.6 Set

Typical implementations of interface `List<E>` like `ArrayList<E>` or `LinkedList<E>` necessarily have a downside: checking whether a list contains some value typically requires checking for each and every list element whether it is equal to that value. Formally, for these implementations the worst-case time complexity for the `contains(Object)` method is in time linear in the size of the list.[5]

If all we want to track is whether some value "is known", a `Set<E>` may be a better alternative – typical implementations of this interface do not need to check every element of the set for equality with the value.

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

[4] You may wonder why there is no interface `Stack<E>`. `Stack<E>` is a special case in Java because it 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. Nowadays it is recommended to use an implementation of interface `Deque<E>` instead. Hindsight is 20/20, also in the design of core libraries – still, this example shows that it is often a good idea to introduce interfaces that may have several different implementations.

[5] This is an example for the *time complexity* of an algorithm. This subject is covered more in-depth in the *Fundamentals of Computing* module. As a software developer, you will need to be able to choose between different implementations of an interface based on different trade-offs between the implementations, such as the time complexity for their operations. For now, just follow the "rules of thumb" mentioned here.

A Set<E> is a collection that cannot contain (i.e., keep track of) duplicate elements. If you try to add an element to a set that already contains that element, nothing happens. The core Java libraries provide us with three widely used implementations of interface Set<E>.

- Class HashSet<E> uses the hashCode() method declared in class Object to get a quick "summary" of an object. This allows for an efficient check whether the set contains a specific object: if the hashCode() method is implemented sensibly in the class used for the elements, checking whether a HashSet<E> contains some value can be done in constant time (which is very good indeed).

  To work correctly, class HashSet<E> requires that the implementation of method hashCode() is *consistent* with method equals(Object):

  > If a.equals(b) holds, then also a.hashCode() == b.hashCode() must hold.[6]

  If your class just inherits the implementations of equals(Object) and hashCode() from class Object, this is the case, and you can put objects of your class into a HashSet<E> without problems. But if you choose to override method equals(Object), you must also remember to override method hashCode() in a consistent way – otherwise your hash set may give incorrect answers.

  A downside of HashSet<E> is that the iteration order is "chaotic": the order in which elements were inserted is not necessarily the order in which they are returned by the iterator, and the class makes no guarantees about the iteration order.

- For user-facing programs, an unpredictable iteration order is a bad idea and will make end users unhappy. This is why Java 4 introduced the class LinkedHashSet<E>, a subclass of HashSet<E>. It works essentially like a HashSet<E>, but additionally guarantees that an iterator over the set will return the elements in the order in which they had been inserted.[7] Usually a LinkedHashSet<E> is a better choice over a HashSet<E>.

  A LinkedHashSet<E> has the same requirements on the type of the elements as HashSet<E>.

- Finally, there is the interface SortedSet<E> with an implementation in class TreeSet<E>. Its iterator can return the elements of the set in the "natural order" of class E if E implements interface Comparable<E> and thus has a method compareTo(E) that returns whether this object comes before or after the other object (or that they are equal). Interface Comparable<E> is implemented by many classes in the Java libraries (e.g., `class String implements Comparable<String>`), and you can also implement it in your classes as well.

[6] Internally, class HashSet<E> uses the contraposition of the statement: if a.hashCode() != b.hashCode() does *not* hold, then a.equals(b) cannot hold either, and there is no point checking this statement. This drastically reduces the number of candidates that our HashSet<E> must check when we ask it whether it contains a certain value. The HashSet<E> groups the values that it stores by their hash codes, usually grouping values with "similar" hash codes together into a corresponding list. The query myHashSet.contains(value) only needs to check the contents of the single internal list in which value's hash code might be found – if value is not in this list, it is not anywhere in myHashSet.

[7] This is accomplished using an additional doubly-linked list through all its entries that stores the elements in insertion order.

Alternatively, the iterator can use the order specified by a `Comparator<E>` object with a method `compare(E, E)` that can tell which of the two objects should come first.[8]

If you want to use a `SortedSet<E>`, you must make sure that the order used for the set is consistent with `equals`:

if a and b are not **null**, then:

`a.compareTo(b) == 0` if and only if `a.equals(b)`, or
`comparator.compare(a,b) == 0` if and only if `a.equals(b)`.

A `TreeSet<E>` needs *logarithmic* time as a function of its size to check whether it contains a certain value. This is better than the linear time that a list would need, but worse than the constant time that a (linked) hash set would need.

So, if you do not need the order and automatic sorting provided by a `TreeSet<E>`, I would recommend you to use a `LinkedHashSet<E>` as your first choice if you need an implementation of interface `Set<E>`.

A typical idiom to create a new set is:

```
1        Set<MyClass> mySet = new LinkedHashSet<>();
```

3 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 `LinkedHashMap<K,V>` (in some cases, a `TreeMap<K,V>` may be useful, e.g., if the order of keys is important). A typical idiom to create a new map is:

```
1        Map<KeyClass,ValueClass> myMap = new LinkedHashMap<>();
```

The similarity of the names of these interfaces/classes for maps to the names of interfaces/classes for sets that we saw earlier is not a coincidence: for each of the interfaces and classes from Section 2.6 for a `Set<E>`, there is also a corresponding interface or class with type `Map<E,V>`, with the same requirements on type `E`.

[8] To be precise, it suffices if the comparator object can compare objects of a superclass of `E`: if we actually want to compare Dogs, it is enough if we can compare Animals. Java Generics allow us to specify this using a `Comparator<? super E>` wildcard. We will learn more on wildcards later today.

4 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 vice versa. You should look a the JavaDoc and get familiarised with them.

There is a useful helper class called `Collections` (note the final s) in package `java.util`. It provides several useful helper methods, e.g., for checking that two collections are disjoint, for sorting collections, for filling a list with a specific element, ...

5 Type wildcards

Recall our recurring example of an inheritance hierarchy where classes `Dog` and `Cat` are both direct subclasses of class `Animal`.

By Liskov's substitution principle, we can write:

```
1        List<Animal> intList = new ArrayList<Animal>();
```

This is fine: every `ArrayList<Animal>` can do the job specified by interface `List<Animal>`.

Now we may wonder: can we also write something like the following?

```
1        List<Animal> animalList = new ArrayList<Dog>();
```

That is, is `ArrayList<Dog>` a subtype of `List<Animal>`, or at least of `ArrayList<Animal>`?

It turns out, perhaps a bit surprisingly, that the answer is *no*. There is a good reason for that. Let's assume for a moment that the above was okay for the Java compiler. Then also the following would be accepted by the compiler:

```
1        List<Dog> dogList = new ArrayList<Dog>();
2        List<Animal> animalList = dogList; // The Java compiler actually says "no" here.
3        animalList.add(new Cat()); // Cat is a subclass of Animal, so this would be ok.
4        Dog d = dogList.get(0); // Type error! That's a Cat, not a Dog!
```

Instead, Java lets us write something similar:

```
1        List<? extends Animal> animalList = new ArrayList<Dog>();
```

We call "`List<? extends Animal>`" an *upper-bounded wildcard*: we can use any type between the "`<...>`", as long as it is a subtype of `Animal` (or `Animal` itself). In other words, `Animal` is an "upper bound" for the type of the list elements.

So should we always use "<? extends Foo>" instead of "<Foo>", then?[9]

The answer is, again, *no*. Consider the following example:

```
1  public static void addCat(List<? extends Animal> animalList) {
2      animalList.add(new Cat()); // The compiler complains here.
3  }
```

The compiler has a good reason to reject the above code. We are allowed to write the following:

```
1      List<Dog> dogList = new ArrayList<>();
2      addCat(dogList); // ok, because Dog extends Animal; would sneak in a Cat
3      Dog d = dogList.get(0); // meeouw!
```

However, if all we want is to *read* from the list (or other collection), it can make sense to use the wildcard `List<? extends Foo>` instead of `List<Foo>` for the formal parameter of our method. Here is an example using the class `Number` from the Java libraries, a superclass of many classes like `Integer`, `Double`, `BigInteger`, ...:

```
1  public static double sum(List<? extends Number> numberList) {
2      double runningTotal = 0;
3      for (Number n : numbers) {
4          // method doubleValue() is the reason for needing the
5          // "upper bound" Number for the generic type of numberList
6          runningTotal += n.doubleValue();
7      }
8      return runningTotal;
9  }
```

Now we can use method sum not only with arguments of type `List<Number>`, but also with arguments of types `List<Integer>`, `List<Double>`, `List<BigInteger>`, ... Thus, using a wildcard has made our method more widely applicable.

Analogously to upper-bounded wildcards, there is a concept called *lower-bounded wildcards*, written using `<? super Foo>` for some complex type `Foo`. Roughly speaking, lower-bounded wildcards like `List<? super Animal>` are useful if we want to make a method more widely applicable that *writes* into a container:

```
1  public static void addAnimal(List<? super Animal> animalList) {
2      animalList.add(new Animal());
3  }
```

[9] You may now wonder what that word "`Foo`" means and whether it is yet another Java keyword. The answer is that the word "Foo", along with a few others such as "Bar", "Baz", ... tends to be used in programming-related documentation and literature in general, to indicate "it is just an example name for a variable/method/type/...". This is a bit similar to the use of names like "Joe Bloggs" or "John/Jane Doe" in English if an example of a person's name is given, but the concrete identity is unimportant.
More on this subject: `https://en.wikipedia.org/wiki/Metasyntactic_variable`

Finally, if you write a method that returns an object with a generic type, say a list of animals, try to avoid wildcards in that generic type: getting a `List<Animal>` is more useful to other programmers than getting a `List<? extends Animal>` or a `List<? super Animal>`.

While our examples for wildcards used the `List<E>` interface, of course the concepts carry over to other generic interfaces and classes, such as `Set<E>`, `Queue<E>`, `Map<K,V>`, `LinkedHashMap<K,V>`, our own `Box<T>`, ...

You will learn more about the use and the limitations of generic types in the *Software Design and Programming* module of the MSc Computer Science at Birkbeck. For now, the main takeaway is that wildcards in generic types of formal parameters can make methods more flexible. This is why many of the methods in the Java libraries, specifically in the helper class `Collections` in package `java.util`, will use types with wildcards for their arguments: the authors wanted these methods to be as flexible as possible.