
# Iterator

# 11

Java knows different collections, for example the classes ArrayList and LinkedList. The data is stored differently internally in these classes. However, it is of interest for the client to be able to iterate over the data without knowing the internal structure. In this chapter, you will look at collections in detail and recreate the ArrayList and LinkedList classes. You will also create an interface that allows you to iterate over these classes.

## 11.1 Two Ways to Store Data

In the following sections, I will show you two ways to create collections.

### 11.1.1 Storing Data in an Array

The first collection you encountered at some point was most likely the array. In an array, you store any number of elements of a particular data type. With the declaration `int[] numbers = new int[5]`, you create an array that can store five int numbers. You access the individual memory areas very efficiently. An array has the disadvantage that it cannot be enlarged. This is impractical if it turns out that more data is to be stored than it was originally intended.

The basis for our first collection will be an array. The elements in this array will be of the general type Object. Initially, five objects are to be referenced. To be able to store different data types in a type-safe way, create the class generically. In the sample project Collections_1 you will find the following code:



O. Musch, *Design Patterns with Java*, https://doi.org/10.1007/978-3-658-39829-3_11

```
public class MyArray<E> {
    private int counter = 0;
    private Object[] elements = new Object[5];
}
```

To insert a new element, define the `add()` method. It is passed an argument of generic type. This element is stored at the next free position in the array. The `counter` data field stores this position. If five elements are already stored and a sixth is to be added, the database must be expanded. Since an array cannot be enlarged, the only way left is to redefine the array. In general: If the next free position is equal to the number of elements, the size of the array must be increased by a certain value. In the example, the array size is to be increased by another five elements.

```
public void add(E e) {
    if (counter == elements.length) {
        var tempArray = new Object[counter + 5];
        System.arraycopy(elements, 0, tempArray, 0, counter);
        elements = tempArray;
    }
    elements[counter] = e;
    counter++;
}
```

The client may want to inquire how many items are stored in the collection. For this, it is sufficient that you return the position of the next free item.

```
public int size() {
    return counter;
}
```

The collection only fulfills its purpose when the individual elements can be returned. To do this, you create the `get()` method, which expects an index as a parameter that describes the location of the element you are looking for in the database. Before the return, the stored value is cast to the generic type.

```
public E get(int index) {
    return (E) elements[index];
}
```

If you want to delete an element, the counter must first be decremented. The element is then deleted by moving the subsequent elements forward one place at a time. However, in order not to throw an “index out of bound” exception, checks on the range between `0` and `counter` are still required. And to avoid leaving the removed object at the end of the field in memory, it must be overwritten with `null`.

```
public void remove(int index) {
    if ((index <= counter) && (counter > 0)
                           && (index >= 0)) {
        if (index != counter)
            System.arraycopy(elements, index + 1,
                                       elements, index,
        elements.length - 1 - index);
        elements[counter--] = null;
    }
}
```

By the way, the collection you have just developed corresponds in its methodology to the ArrayList of the class library. It is optimal if you need to access individual elements via their index.

### 11.1.2 Storing Data in a Chain

You take a completely different approach if you store the individual elements not in an array, but in a chain. Each element knows its successor. It would also be conceivable that an element also knows a predecessor; I won't go into this possibility further – the project would only become unnecessarily extensive without changing the underlying principle.

Strings and any other objects you want to store in your list, I call elements. They are not stored in the collection, but in the instance of an inner class I call node. The `Node` class has two data fields that store the element you want to store and the subsequent node object. The collection can then restrict itself to referencing the first object (in the `header` data field).

```
public class MyList<E> {
    private int counter = 0;
    private Node<E> header = null;
    private class Node<E> {
        private final E element;
        private Node<E> nextNode;
        Node(E element, Node<E> next) {
            this.element = element;
            this.nextNode = next;
        }
    }
}
```

Data is inserted into the collection by creating a new node object. This object is referenced by the `header` variable and displaces the object previously stored as `header`. The `nextNode` field of the new header object references the previous `header`. And

finally, the counter must be incremented. When you query the size of the collection, the counter is returned.

```
@SuppressWarnings("unchecked")
public void add(E element) {
    header = new Node(element, header);
    counter++;
}

public int size() {
    return counter;
}
```

To delete an element from the collection, go through the collection with a while loop and check whether the referenced element is the same as the element you are looking for. If so, pass the reference of the subsequent node object to the predecessor of the node object that references the element you are looking for. Then decrement the counter. The local variable `previous` references the predecessor of the node object whose element is currently being checked.

```
public boolean remove(E element) {
    Node<E> previous = null;
    var tempNode = header;
    while (tempNode != null) {
        if (equals(element, tempNode.ELEMENT)) {
            if (previous == null)
                header = tempNode.nextNode;
            else
                previous.nextNode = tempNode.nextNode;
            zaehler--;
        return true;
        }
        previous = tempNode;
        tempNode = previous.nextNode;
    }
    return false;
}
```

The `get()` method is intended to solve the same task as the get method of the `MyArray` class. However, because the database is not index-based, you cannot directly query the xth element in the collection. You must go through the entire collection until you find the xth element.

```
public E get(int index) {
    if (index < 0 || index >= counter)
        throw new NoSuchElementException(index + " Size "
                                         + counter);
    var tempNode = header;
    for (var i = 0; i < index; i++)
        tempNode = tempNode.nextNode;
    return tempNode.ELEMENT;
}
```

By the way, the collection you just developed is similar in methodology to the LinkedList of the class library.

## 11.2 The Task of an Iterator

When you create a collection, you will certainly want to iterate over all elements. A first approach could be the procedure of the test methods (of the respective main method), which you can find for both classes in the sample project Collections_1. Please analyze them and run them. In both test methods, iterate over the data collection with a for loop.

```
for (var i = 0; i < myList.size(); i++)
    System.out.println(myList.get(i));
```

You access each element of the collection with the `get()` method. With the `MyArray` class, this makes perfect sense. However, the performance of the `MyList` class falls far short of its capabilities when accessing its elements in an index-based manner. So it makes sense to outsource the algorithm of how to iterate over the collection to its own class, the iterator. You can think of the iterator as a bookmark that is pushed through a book page by page. The iterator knows the specific features of a collection and makes the best use of them.

You can design an iterator as an internal iterator or as an external iterator. Internal means that you pass the action of iterating to the iterator, which iterates "independently" over all objects. When you program an external iterator, you have it return the next item at a time and query whether there are any other items you can request; thus, it is the client's job to drive the iterator. You get the most flexibility with an external iterator. In what follows, I will only deal with external iterators. You will find an internal iterator in the composite pattern.

## 11.3 The Interface Iterator in Java

The class library knows the interface `Iterator`, which is an interface for all conceivable iterators. With `hasNext()` you let yourself return whether there are still further elements in the data collection. The `next()` method returns the next element in the collection. If the client wants to access an element that doesn't exist, throw a `NoSuchElementException`. And finally, `remove()` deletes the current element from the underlying data collection. According to the specification, the `remove()` method does not need to be implemented, it is allowed to throw an `UnsupportedOperation` Exception.

### 11.3.1 The Iterator of the Class MyArray

The simplest form of an iterator is returned by the `iterator()` method in the `MyArray` class. The iterator internally stores the position at which the bookmark is set. The `next()` method returns the next element in each case, and the `hasNext()` method returns `true` if there are more elements. Take a look at the sample project Collections_2 and there the class `MyArray`.

```
public class MyArray<E> {
    // … abridged
    public Iterator<E> iterator() {
        return new Iterator<E>() {
            private int position = -1;

            @Override
            public boolean hasNext() {
                return (position < size())
                       && elements[position + 1] != null;
            }

            @Override
            public E next() {
                position++;
                if (position >= size()
                        || elements[position] == null)
                    throw new NoSuchElementException("No more data");
                @SuppressWarnings("unchecked")
                var value = (E) elements[position];
                return value;
            }
```

```
            @Override
            public void remove() {
                throw new
                        UnsupportedOperationException();
            }
        };
    }
    // … abridged
}
```

The next section shows how to use the iterator.

#### 11.3.1.1 Test of the Iterator

You first create a collection of type `MyArray` in the main method and store some strings in it.

```
var myArray = new MyArray<>();
myArray.add("String 1");
// … abridged
myArray.add("String 6");
```

After that, you have an iterator returned and query data in a while loop until there is no more data in the collection. To provoke the exception, you specifically retrieve one more element than is stored.

```
var iterator = myArray.iterator();
while (iterator.hasNext()) {
    String temp = iterator.next();
    System.out.println(temp);
}
// throws an exception
System.out.println(iterator.next());
```

Each string is now output on the console. Afterwards the exception is thrown.

#### 11.3.1.2 Benefits and Variations of the Iterator

Iteration now becomes much easier, and since the client code doesn't change, the lists are interchangeable. You first let it give you an iterator and call the method `next()` as long as the method `hasNext()` returns a `true`. The examples are kept very simple, the method `iterator()` returns the instance of an anonymous class as the iterator. This iterator steps through the database one element at a time from front to back. This simplification should not obscure the fact that you, as a programmer, are free to define the iterator as you need. For example, the iterator could go through the array from back to front.

Alternatively, you could have an iterator that first copies and/or sorts the database before returning the individual elements. Also, if you build your own complex structures (tree structures, …), you can design a "standard" iterator for it according to your needs. Of course, you don't have to define the iterator as an anonymous class – you could also design a class outside the collection's namespace.

### 11.3.2 The Iterator of the Class MyList

The class `MyList` - also in the project Collections_2 – is internally structured differently. The individual elements are not stored in an array, but linked. What consequence does this have for the iterator? In contrast to `MyArray` no counter is stored as bookmark, but the current element. Initially the current element is set to the header of the list.

If you want to test whether a collection contains more elements, you must ask whether the current element of the iterator is non-null, in which case the `hasNext()` method may return `true`. The `next()` method returns the contents of the current element and advances the pointer to the next element one place.

```
public Iterator<E> iterator() {
    return new Iterator<E>() {
        private Node current = header;
        @Override
        public boolean hasNext() {
            return (current != null);
        }

        @Override
        public E next() {
            if (current == null)
               throw new NoSuchElementException("...");
            @SuppressWarnings("unchecked")
            var value = (E) current.element;
            current = current.nextNode;
            return value;
        }

        @Override
        public void remove() {
            throw new UnsupportedOperationException();
        }
    };
}
```

In the next section we will also test this iterator.

#### 11.3.2.1 Test of the Iterator

To test the iterator, you create an instance of the `MyList` class and store various strings in it. You let it return its iterator and iterate over the complete collection. Of course, I also included one query too many in this test to provoke the exception to be thrown.

```
var myList = new MyList<>();
myList.add("String 1");
// … abridged
myList.add("String 6");
var iterator = myList.iterator();
while (iterator.hasNext())
    System.out.println(iterator.next());
System.out.println(iterator.next());
```

Again, at first all strings are printed to the console; then an exception is thrown.

#### 11.3.2.2 Benefits of the Iterator

The special features of the two collection classes are now taken into account without the client even noticing. You remember that with the Observer pattern, you saw that iterators are prone to problems resulting from concurrency. Since you can create multiple iterators, it's not at all unlikely that an element will be deleted, added, or replaced during iteration. You are then quickly faced with the situation that an iterator wants to access an element that no longer exists or that it cannot yet access a new element. The collection classes of the class library throw a `ConcurrentModificationException` in such cases.

## 11.4 The Iterable Interface

A while loop is related to the for loop. Instead of the while loop.

```
while (iterator.hasNext()) {
    // … Action
}
```

you could also use a for loop:

```
for (var iterator = myList.iterator();
                        iterator.hasNext(); ; ) {
    // ... Action
}
```

Since Java 5, there is the for-each loop. Iterating becomes much easier:

```
for (var tempString : list)
    System.out.println(tempString);
```

How can you prepare your `MyList` and `MyArray` collections so that you can use them in a for-each loop? All you need to do is implement the `Iterable` interface. This interface dictates the `iterator()` method, whose return value is the collection's iterator. You can see this in the Collections_3 sample project – here using the `MyArray` class as an example:

```
public class MyArray<E> implements Iterable<E> {
    public Iterator<E> iterator() {
        // ... as before
    }
    // … abridged
}
```

Now you can use the MyArray class in an extended for loop:

```
var myArray = new MyArray<>();
myArray.add("String 1");
// … abridged
myArray.add("String 6");

for (var tempString : myArray)
    System.out.println(tempString);
```

This works accordingly for the MyList. Please have a look at it again independently.

Search the API documentation for the interface `Iterable`. A subinterface of this is the `Collection` interface. This interface is implemented by all common collections such as `List` and `Set` and by many others. Therefore, you can trust that all collections define the `iterator()` method, which returns an iterator object.

With a `map`, the issue is a bit more complicated. As an example, consider a `HashMap` that consists of three collections: a `KeySet` for the keys, a `Collection` for the values, and an `EntrySet` for the connection between the two collections. No one can know in advance whether the user will want to iterate over the keys or the values; therefore, the `HashMap` cannot design a "default" iterator.

The for-each loop is a very useful language construct for the programmer. However, although a lot of work is done for the programmer, the compiler has relatively little work to do with it. It rewrites the for-each loop into a while loop before compiling it (code rewriting) and lets you give it the iterator. If you iterate over an array with the for-each loop, the loop is rewritten into a conventional for-next loop and then compiled.

> You can see from this example how patterns have found their way into the class library. Not only the name, but also the realization of the pattern corresponds to the description of the GoF.

## 11.5 Iterator – The UML Diagram

From the example project Collections_3/MyList you can see the UML diagram in Fig. 11.1.

## 11.6 Summary

Go through the chapter again in key words:

- Collections can be structured internally in very different ways.
- Iterators are bookmarks that are moved from one element to the next.
- They allow you to iterate over the collection without knowledge of the internal structure.
- Internal iterators are responsible for the progress of the iterator itself.
- For external iterators, the client is responsible for advancing.
- An iterator, when following the Java specification, has three methods:
- `remove()` deletes an element from the collection – the method does not need to be overridden,
- `hasNext()` returns `true` if the collection has more elements,
- `next()` returns the next element in the collection; if there is no element, a `NoSuchElementException` is thrown.
- The `ListIterator` is an iterator specialized on lists and has some more methods that allow iteration in both directions.
- The iterator is returned by the `iterator()` method.
- The programmer can adapt the definition of a custom iterator to the structure of his collection as needed.

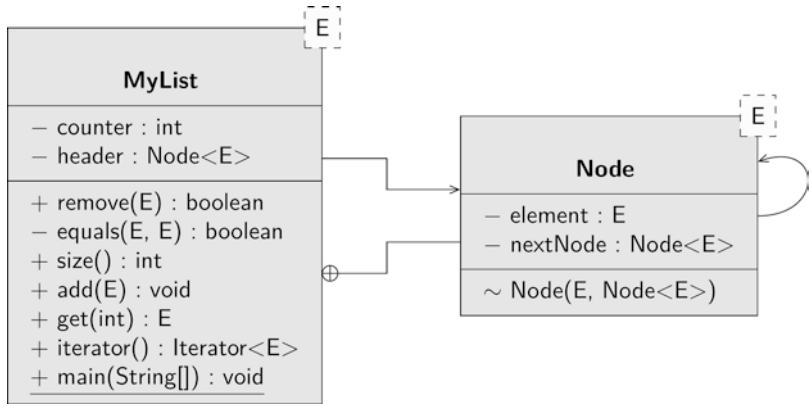


**Fig. 11.1** UML diagram of the Iterator Pattern (sample project Collections_3/MyList)

- There can be multiple iterator classes and multiple iterator instances.
- While at least one iterator is active, the database must not be changed.
- If a collection implements the `Iterable` interface, you can iterate over it with the for-each loop.

## 11.7 Description of Purpose

The Gang of Four describes the purpose of the "Iterator" pattern as follows:

> Provides a way to sequentially access the elements of a composite object without exposing the underlying representation.