# **Lists in the Java Collections API**

* The Java language includes, in its library, an implementation of common data structures.
* This part of the language is popularly known as the **Collections API**.
* The **List ADT** is one of the data structures implemented in the Collections API.

# **Collection Interface**

* The notion of a **collection**, which **stores a collection of identically typed objects**, is abstracted in the Collection interface.
* The Collections API resides in package java.util.
* Figure 3.5 shows the most important parts of this interface (some methods are not shown).

Text, letter

Description automatically generated

## **Collection Methods**

* Many of the methods in the Collection interface do the obvious things that their names suggest.
  + size() returns the number of items in the collection
  + isEmpty() returns true if and only if the size of the collection is zero
  + contains(x) returns true if x is in the collection
  + add(Anytype x) adds item x to the collection, returning true if the operation succeeds and false if it fails for a plausible (nonexceptional) reason (i.e. no duplicates)
  + remove(Anytype x) removes item x to the collection, returning true if the operation succeeds and false if it fails for a plausible (nonexceptional) reason (i.e. not found)

## **Iterators**

* The **Collection** interface extends the **Iterable** interface.
* Classes that implement the Iterable interface can have the enhanced for loop used on them to view all their items.
* For instance, the routine in Figure 3.6 can be used to print all the items in any collection.
* The implementation of this version of print is identical, character-for-character, with a corresponding implementation that could be used if coll had type AnyType[].

Text, letter

Description automatically generated

Text

Description automatically generated

* Collections that implement the **Iterable** interface must provide a method named iterator that returns an object of type Iterator.
* The Iterator is an interface defined in package java.util and is shown in Figure 3.7.

Text

Description automatically generated

* The idea of the Iterator is that via the iterator method, each collection can create, and return to the client, an object that implements the Iterator interface and stores internally its notion of a current position.
* next()
  + Each call to next gives the next item in the collection (that has not yet been seen).
    - The first call to next gives the first item,
    - the second call gives the second item,
    - and so on and so forth.
* hasNext()
  + Each call to hasNext tells you if there is a next item (another item in the collection).
* When the compiler sees an **enhanced for loop** being used on an object that is Iterable, it mechanically replaces the enhanced for loop with calls to
  + The **iterator()** method to obtain an Iterator
  + A while loop calling **hasNext()** to process all items in the collection
  + Calls to **next()** to get the next item in the collection (we know there is a next because of hasNext)
* Thus the previously seen print routine in Figure 3.6 is rewritten by the compiler as shown in Figure 3.8.
* Because of the limited set of methods available in the Iterator interface, it is hard to use the Iterator for anything more than a simple traversal through the Collection.

Text

Description automatically generated

Text

Description automatically generated

* The Iterator interface also contains a method called **remove**.
* remove()
  + This methods lets you remove the last item returned by next (after which you cannot call remove again until after another call to next).
* Although the Collection interface also contains a remove method, there are presumably advantages to using the Iterator’s remove method instead.
* The main advantage of the Iterator’s remove method is that the Collection’s remove method must first find the item to remove.
* Presumably it is much less expensive to remove an item if you know exactly where it is.
* An example that we will see in the next section removes every other item in the collection. This code is easy to write with an iterator, and potentially more efficient than using the Collection’s remove method.
* When using the iterator directly (rather than indirectly via an enhanced for loop) it is important to keep in mind a fundamental rule: **If you make a structural change to the collection being iterated** (i.e., an add, remove, or clear method is applied on the collection), **then the iterator is no longer valid** (and a ConcurrentModificationException is thrown on subsequent attempts to use the iterator).
* This is necessary to avoid ugly situations in which the iterator is prepared to give a certain item as the next item, and then that item is either removed, or perhaps a new item is inserted just prior to the next item.
* This means that you shouldn’t obtain an iterator until immediately prior to the need to use it. However, if the iterator invokes its remove method, then the iterator is still valid. This is a second reason to prefer the iterator’s remove method sometimes.

# **ListIterators**

* Figure 3.13 shows that a **ListIterator** extends the functionality of an **Iterator** for Lists.

Text

Description automatically generated

* previous() and hasPrevious() allow traversal of the list from the back to the front.
* add() places a new item into the list in the current position. The notion of the current position is abstracted by viewing the iterator as being between the item that would be given by a call to next and the item that would be given by a call to previous, an abstraction that is illustrated in Figure 3.14. add is a constant-time operation for a LinkedList but is expensive for an ArrayList. set changes the last value seen by the iterator and is convenient for LinkedLists. As an example, it can be used to subtract 1 from all the even numbers in a List, which would be hard to do on a LinkedList without using the ListIterator’s set method.

A picture containing text

Description automatically generated

# **List Interface**

* The collection that concerns us the most in this section is the list, which is specified by the **List** interface in package **java.util**.
* The **List** interface extends **Collection**, so it contains all the methods in the **Collection** interface, plus a few others.
* Figure 3.9 illustrates the most important of these methods.

Text, letter

Description automatically generated

**List Methods**

* **get(int index)** allows **access** to an item at a specified position in the list, given by its an .
  + index 0 is the front of the list,
  + index size()-1 represents the last item in the list,
  + index size() represents the position where a newly added item can be placed.
* **set(int index)** allows the client to **change** an item at a specified position in the list, given by an index.
  + index 0 is the front of the list,
  + index size()-1 represents the last item in the list,
  + index size() represents the position where a newly added item can be placed.
* **add(int index, AnyType x)** allows the placement of a new item in position index.
  + an add at position 0 adds the item to the front,
  + an add at position size() is adding an item as the new last item.
  + an add to the middle of the list pushes subsequent items one position higher.
* **remove(AnyType x)** removes the first occurrence of the specified element from this list.
  + if the element is it is present (optional operation). If this list does not contain the element, it is unchanged.
* **remove(int index)** removes and returns an item at a specified position from the list.
  + a remove at position 0 removes the item from the front
  + a remove at position size() removes the last item from the list.
  + a remove from the middle of the list shifts any subsequent elements to the left (subtracts one from their indices).
* Finally, the List interface specifies the listIterator method that produces a more complicated iterator than normally expected.