COMP6700/2140 Abstract Data Types: Lists and Iteration

Alexei B Khorev and Josh Milthorpe

Research School of Computer Science, ANU

19 April 2017

Abstract Data Types (ADTs)

An *abstract data type* describes data from the point of view of a user, in terms of its behaviour.

A *container* is a very general ADT, serving as a holder of objects. A *list* is an example of a specific container ADT.

An ADT can be described in terms of the semantics of the operations that may be performed over it.

A *data structure* is a concrete implementation of an ADT.

The List ADT

The *list* ADT is a container known mathematically as a *finite sequence* of elements. A list has these fundamental properties:

- duplicates *are* allowed
- order is preserved

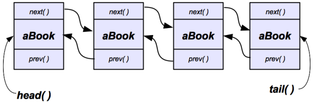
A list may support operations such as:

- *create*: construct an empty list
- *add*: add an element to the list
- *is empty*: test whether the list is empty
- *get element*: get an element at a chosen position in the list

List Implementations

The *List* interface represents an ordered collection which allows a user to access and insert elements at any point in the sequence. Implementations include:

- *ArrayList*: a familiar class with fast access and slow modification:
  - `get(i), set(i, elem),` — $\mathcal{O}(1)$ (constant)
  - `add(i, elem), remove(i)` — $\mathcal{O}(N - i)$ (requires recopying of a part of the list)
- *LinkedList*: a *doubly linked list* which can be traversed *both* forward and backward — each node has two references, to the preceding node and the following node. Slow access and fast modification:
  - `get(i), set(i, elem),` — $\mathcal{O}(i)$ (needs i steps to get there)
  - `add(i, elem), remove(i)` — $\mathcal{O}(1)$ (no recopying necessary)



- *ArrayList* is almost always preferable to *LinkedList* since its operations have better or same performance. One exception — when the number of elements stored inside the list changes frequently at runtime.

Let’s Make a Library!

First, we need a book:

```
public class Book implements Comparable<Book> {
   private String title;
   private boolean fiction;
   public Book(String title, boolean fiction) {
      this.title = title;
      this.fiction = fiction;
   }
   public String toString() {
       return title;
   }
   public boolean isFiction() {
       return fiction;
   }
   /** implements compareTo(Book) of Comparable
    * so the list of books can be sorted
    * @return int result of comparing +1,0,-1 */
   public int compareTo(Book b) {
      return this.title.compareTo(b.toString());
   }
}
```

Book List

... actually, we need more than one book - we need a whole list:

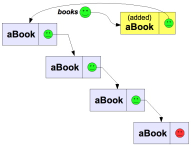
```
public interface BookList  {
 public void addFirst(Book newBook);
 public boolean add(Book newBook);
 public boolean remove(Book book);
 public void insert(Book newBook, int position);
 public Book get(int position);
 public boolean isEmpty();
 public boolean contains(Book book);
 public int size();
 public String toString();
 public Iterator<Book> iterator();
}
```

The methods in BookList.java match methods of theCollection<E> interface. The *BookList* interface is therefore a simplified example of Collection<Book>.

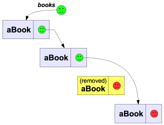
BookList using Linked List

BookList may be implemented as a linked list: three of its methods are illustrated:

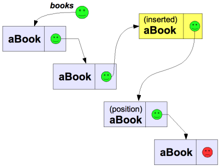
`boolean addFirst(Book b)`



`boolean remove(Book b)`



`void insert(Book b, int p)`



The `iterator()` method returns an implementation of *Iterator* interface given by the class BookListIterator.java.

Iterator

An iterator traverses every element of a collection in some order.

A container object (of type `T`) may provide an access to its internals by **EITHER**

- implementing the method `iterator()` (which is included explicitly, or inherited) — when called, it returns an *Iterator* object, which is your guide to the container internals; the object has a contract to implement `iterator()` if its class implements `java.lang.Iterable<T>` interface. **OR**
- implementing the following three methods:
  - `boolean hasNext()`, returns `true` if there are more elements left;
  - `T next()`, returns the next element;
  - `void remove()`, removes the last element returned by the iterator, (**subtle operation**: requires *safe* removal, optional);

  the object has a contract to implement these three methods if its class implements `java.util.Iterator<T>` interface

Details of implementations are intimately related to the implementation of the container class (our examples of *BookList* implementation include *different* implementations of the iterator: BookListWithArray.java and BookListWithLL.java).

Iterators: Traversal

The access to container's elements is performed via a *traversal*: sequential passage from one element to the next starting with the first one, which is the entry point to the container:

◇ The implementation of *Iterable* allows traversal with a *for-each* loop. However, this approach (see next slide) is not very safe, when one has to filter (remove by a certain criterion) the collection elements, or to traverse multiple collections simultaneously.

◇ The alternative way to traverse a collection involves an explicit *Iterator*, which is the only safe way to modify the collection along the way. (The *Iterator*-based "plain" `for`-loop is also the right way to traverse more than one collections simultaneously).

The `iterator()` method returns an implementation of `Iterable<T>` interface; it guarantees that when the collection is modified during a traversal, the iterator will properly move through the subsequent elements of the collection.

The slides below "Traversing a *Collection* with *Iterator*" demonstrates the two traversals — one not suitable for removal, and second which is removing elements correctly.

Traversal with for-each loop

To traverse collections (and arrays) with less verbose coding, *Java 1.5* introduced the “foreach” loop, a version of `for`-loop (borrowed from *Perl* where `foreach` is the keyword, but not in *Java*!):

```
for( type var : collection ) statement block
```

Two examples:

```
for( String arg : args ) { // args is an String[] type
   System.out.println(arg);
}
for (Book book : books) {     // BookListIsAL: implementation
   System.out.println(book);  // of BookList with ArrayList
}
```

A “for-each” loop can be used to iterate through most of the Java collection classes (arrays, ArrayLists, HashSets, etc) — *anything* that implements *Iterable* interface. The example is in the Library.java, the client program of the BookList.java types.

Once again, **remember**: The collection type which you define yourself must implement *Iterable* interface to be amenable for *for-each* traversal.

The `for-each` loop can also be used on arrays, but it is **not** suitable if there is need to use index during traversal, or traversal is done in the opposite direction.

Traversal with *Iterator*

When we need to modify a collection (e.g. filter out its elements based on some criterion), the safest way is to implement *Iterable* interface and use the *standard* `for`-loop. This can be subtle. The standard *Collection* classes (*ArrayList*, *LinkedList*) include a proper implementation of *Iterator* which guarantees a safe *co-modification* during a traversal:

```
BookListIsAL books = new BookListIsAL();
books.add(new Book("Java Software Solutions", false));
... ...
for (Iterator iter = books.iterator(); iter.hasNext(); ) {
    Book nextBook = iter.next();
    if (!nextBook.isFiction())
        iter.remove();// call "remove" only through Iterator reference!
}
```

The `remove()` method may be called only *once per call to* *`next()`* and throws an exception if this rule is violated. An attempt to achieve the same effect with the would be identical `for-each` loop — and direct call to `remove()` — will result in a run-time exception:

```
for (Book book: books) {     // for-each hides the iterator,
    if (book.isFiction())   // and one cannot call remove()
        books.remove(book);
}
Exception in thread "main" java.util.ConcurrentModificationException
```

Nested Iterators

(From Java SE 8 Technotes) When [one tries] to do nested iteration over two collections, a typical mistake is to call outer-iterator's `next()` too many times and exhausting it (not to mention making logical errors along the way) before its due time:

```
List suits = ...; List ranks = ...;
List sortedDeck = new ArrayList();
// BROKEN - throws NoSuchElementException!
for (Iterator i = suits.iterator(); i.hasNext(); )
    for (Iterator j = ranks.iterator(); j.hasNext(); )
        sortedDeck.add(new Card(i.next(), j.next()));
```

There is an ugly solution and a fair one: (with iterator) add a variable in the scope of the outer loop to hold the suit, or use the foreach-loop:

```
for (Iterator i = suits.iterator();
   i.hasNext(); ) {
   Suit suit = (Suit) i.next();
   for (Iterator j = ranks.iterator();
      j.hasNext(); )
      sortedDeck
        .add(new Card(suit, j.next()));

}
```

```
for (Suit suit : suits)
    for (Rank rank : ranks)
        sortedDeck
            .add(new Card(suit,
                        rank));
```

Removal without Traversal and Iterable String

The default methods "revolution" has brought fruit to the "old" API: `java.util.Collection` interface (`stream()` methods notwithstanding) and `java.lang.CharSequence`:

- `boolean removeIf(Predicate<? super E> filter)` which can remove *in-place* all "undesirable" elements without the client having to iterate:

```
public class RemovingByIf {
   public static void main(String[] args) {
      Random rand = new Random();
      List<Integer> numbers =
      Stream.generate(() -> rand.nextInt(200) - 100) // you can ignore this
            .limit(20)                               // if it looks weird:
            .collect(Collectors.toList());           // we're just creating
      System.out.println(numbers);                   // random ints array
      numbers.removeIf(x -> x < 0);
      System.out.println(numbers);
   }
}
```

- `java.util.stream.IntStream chars()` of the inteface *CharSequence* almost makes strings iterable (*String* implements *CharSequence*)

```
int total = "The Ministry of Silly Walks".chars().reduce(0, (x,y) -> x+y);
```

Further Reading

- Hortsmann *Core Java for the Impatient*, Ch. 7.1–7.2
- Oracle *The Java Tutorials*: The List Interface