COMP6700/2140 Implementation of ADT

Alexei B Khorev and Josh Milthorpe

Research School of Computer Science, ANU

May 2017

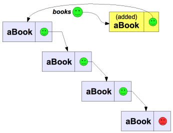
Topics

1. Implementing an ADT
2. `List` ADT: `array` based Implementation
3. `List` ADT: Linked List based Implementation:
    - `Node` with an element
    - `Link` of nodes ("chain", linked list)
4. Implementing via Inheritance from standard API: Adapters

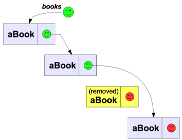
List ADT Operations

Recap from the A1 lecture (the Book class has two "expected" fields: `String title` and `boolean isChildren`), the linked list based implementation of *some* operations:

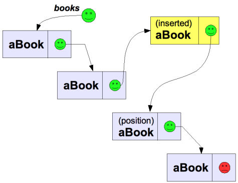
`boolean add(Book b)`



`boolean remove(Book b)`



`void insert(Book b, int p)`

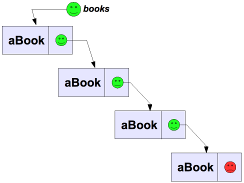


The `iterator()` method returns an implementation of *Iterator* interface given by the class BookListIterator.java. More on an iterator is in the next A-lecture, A2.

Implementing with own Data Structures

The example of BookList.java defines an interface for the *Library program* for filing a collection of *Book* objects, processing and printing them. Its two implementations use different DS for *internal* storing collection elements, but their usage is exactly the same (polymorphism).

1. The BookListWithLL.java implementation uses a *linked list*. It is an example of a *dynamic* DS (size changes during runtime). Each *node* contains a `Book` object and a *link* to the next node. Types like this are *self-referential* — they contain a reference to a same type object. There is no limit on the length. Links can be broken and reconnected: The cost is in traversing the list to find the place `i` where to break/reconnect, $\mathcal{O}(i)$.



1. The second implementation BookListWithArray.java uses an `array` DS, which can provide random access to a its elements, $\mathcal{O}(1)$. The implementation of `add` and `insert` requires copying a part of the underlying array DS, $\mathcal{O}(n-i)$.

Juche: Generic In-house Implementation

Forget *Book*s, we can do it with a type parameter, and we can define our algorithms also generically (independent of a concrete implementation):

- ArrayIterator.java — generic array based implementation of *Iterator* interface (could be defined as inner class in *ListWithArray*)
- BadOperation.java — an exception class to signify an illegal operation
- Book.java — a concrete type to be used in the client *Library*
- Iterator.java — one's own iterator (to make The State Department bilious)
- Library.java — a client class which can choose either array based, or linked list based implementation of *MyList* as a container of *Book* objects
- ListIterator.java — generic linked list based implementation of *Iterator* interface
- ListWithArray.java — an array based implementation of *MyList*
- ListWithLL.java — a linked list based implementation of *MyList*
- MyList.java — a generic interface of a list container type
- MyUtilities.java — a utility class for *generic* methods on a container type whose interface is defined in *MyList* (think of it as an "in-house" version of `java.util.Arrays` or `java.util.Collections`); it has only two methods, `min(list,lo,ho)` and `sort(source, target)` to work with *MyList* objects (*ie*, their implementation uses **only** *MyList* interface methods); the `sort`-method implementation uses the *Selection Sort* algorithm
- Node.java — a generic class used in linked list based implementations

Note: *juche* is the North Korean principle of self-reliance, an ultimate form of a state independence.

Implementations of Collection interface via Inheritance

We have implemented the *BookList* interface from scratch. It is more practical to use data structures provided by the *Java Collection Framework* — for a *List* type ADT (**not** the interface `java.util.List`!), one can, however, use `java.util.ArrayList` and `java.util.LinkedList` which use exaclty the same DS, array and linked list, correspondingly.

When a standard library offers an effective implementation of a collection type, it is more prudent to use it for implementing your own interface. We discuss two ways to use *ArrayList* collection class for implementation of our *BookList* interface. Two approaches are possible.

One approach is very convenient and allows a minimum of work because it uses

Inheritance

Make the class to implement *BookList* and *inherit ArrayList*, thus making the implementation class of the *ArrayList* type. In software design this is called *"Is-A"* relationship (between the classes) — by the virtue of inheritance BookListIsALjava **is** *ArrayList*. The trade-offs:

- **(+)** ***Less*** work if the interfaces of *BookList* and *ArrayList* are similar. Provides access to protected members of the parent class. Allows to override the parent methods.
- **(-)** Too ***rigid construction*** which does not allow future extension of the *BookList* type ("no can do multiple inheritance"). Inheritance represents a strong coupling between classes, and should be avoided when there are alternatives. Inheritance constrains the object type, which can be undesirable for the client. The derived class can become fragile if the parent class includes undocumented "self-use".

Implementations of Collection interface via Composition

Another approach, however, is often a better choice — it's based on

Composition

Compose the class which implements *BookList* with a field of the *ArrayList* object. In software design this is called *"Has-A"* relationship — by the virtue of *composition* (aka *containment*), BookListHasAL.java *has* an instance of the *ArrayList* class among its fields. The trade-offs:

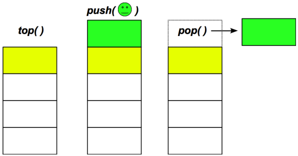
- **(+)** Flexibility without affecting the client code (one can change into "having" a different class of the same type, or even a different type).
- **(-)** Suffers from the *SELF problem*: the composed (aka *wrapper*) class is not suited for use in *callback frameworks*, where objects pass self-references to other objects for later invocations. Because the wrapped object does not know its wrapper, it passes the reference to itself (`this`), and callbacks do not find the wrapper.

Stacks

Queues and stacks are very popular interfaces. Both are widely used in systems level programming (for managing memory and processes) as well as in other applications.

A `Stack<E>` is a "last in first out" (LIFO) collection type which can be implemented by adding (with a `push` method) and extracting (with a `pop` method) from the *head of a list*. The interface:

- `int size();`
- `boolean isEmpty();`
- `int search(E el);`// similar to `indexOf()`
- `E top(E el);` // JCF calls it `peek()`
- `void push(E newEl);`
- `void pop();`
- `String toString();`

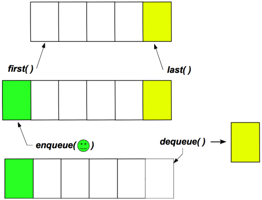


Stack **class** in Java Collections Framework has slightly different interface. It extends *Vector* class to implement with operations which allow *Vector* to be treated as a stack. Java regards *Stack* (as well as *Vector*) as a legacy class (on the way out), and suggests to use *ArrayList* for implementing the *Stack* interface in applications. (*What could be the reason for such policy?*)

Queues

A `Queue<E>` is a "first in first out" (FIFO) type which can be implemented by adding objects (with an `enqueue` method) *to the head* of a list and by extracting them (with a `dequeue` method) *from its tail*. The interface:

- `int size();`
- `boolean isEmpty();`
- `E first(E element);`
- `E last(E element);`
- `void enqueue(E newEl);`
- `void dequeue();`
- `String toString();`

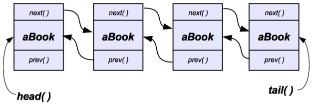


In JCF, *Queue* interface has slightly different names for the above operations, plus additional operations which allow to implement this interface into a stack, as well as to have a specific ordering (defined by the constructor with *Comparator* parameter), for instance, in the *PriorityQueue* class.

Lists

The *List* interface represents an ordered collection with controls *where* a new element is inserted, what element the user can access and search (any, given the index). Implementations classes:

- *ArrayList*, a familiar class with the following performance of the main operations:
  - `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*. This is a *doubly linked list*, or *deque*, can be traversed *both* forward and backward — an element (deque's node) has two references, to the preceding node and the following node. The operation performance is complimentary to the *ArrayList*:
  - `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 usage involves frequent change in the number of elements stored inside the list during the run time.

Set Interface and Its Proper Implementation

Sets a collections which resemble lists, but have two crucial differences:

- The can only contain a unique element (no duplicates allowed) or none at all
- The have no order like lists

The attempt to use a list-like implementation for a set interface results in very inefficient `add` method (which cannot be avoided by using whichever implementation), because its behaviour involves a check on whether the added element is (is not) already present, thus requiring call to `contains`, which is bound to have $\mathcal{O}(N)$ performance.

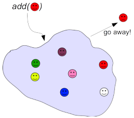
Therefore, a proper implementation of a set interface requires first establishing a more sophisticated structure which would make the "look-up" operation faster (scale slower than $\mathcal{O}(N)$, or not scale at all, $\mathcal{O}(1)$).

The trick which allows to achieve this is a new data structure called a *hash table*, a jewel of the computer science.

Set and *SortedSet*

The extensions of *Collection* type which disallow duplicates (identical elements): a repeated invocation of `add(elem)` with the same element `elem` (or such `elem1`, that `elem.equals(elem1)` returns `true`) returns `false`, and the collection remains unchanged. *Set* types, therefore, are sets in the mathematical sense (some computer scientists call them "bags"). The elements of a set are unordered. The subtype *SortedSet* (extension of the *Set* interface) represents a collection, whose elements can be *compared* — they implement *Comparable* interface. The ordering allows to introduce additional methods:

- `comparator()` — returns the comparator used with this sorted set (`null` in case of *natural ordering*)
- `first()`, `last()` — smallest and largest elements
- `subset(eMin,eMax)`, `headSet(eMax)`, `tailSet(eMin)` — subsets of elements in given ranges

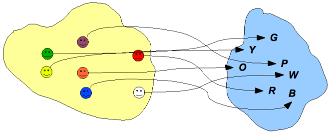


Set is implemented by the *HashSet* class with a hash table as the DS. Content modification (`add`, `set`) and testing (`contains`) are $\mathcal{O}(1)$ operations. *SortedSet* is implemented by the *TreeSet* class with a binary tree as the DS. If the implementation can maintain the *balanced tree*, the search and modify operations are $\mathcal{O}(\log_2 N)$. Example — SetTest.java.

Map and *SortedMap*

Unlike all previous types from JCF, the *Map* family of types does not extends the *Collection* interface: their contract is different — to represent not a collection of elements, but a *correspondence between two collections*. So, a map contains **key/value pairs**, with *no duplicate keys* (the keys form a set) and *at most one value* for each key. *Map* is a model of a mathematical abstraction called *function*. The interface Map<K,V> operations are:

- `V put(K key, V value)`
- `V get(K key)` and `V remove(K key)`
- `boolean containsKey(K key)`
- `boolean containsValue(V value)`
- `public Set keySet()` — returns a **view** of the set of keys
- `public Collection values()` — returns a **view** of the collection of stored values

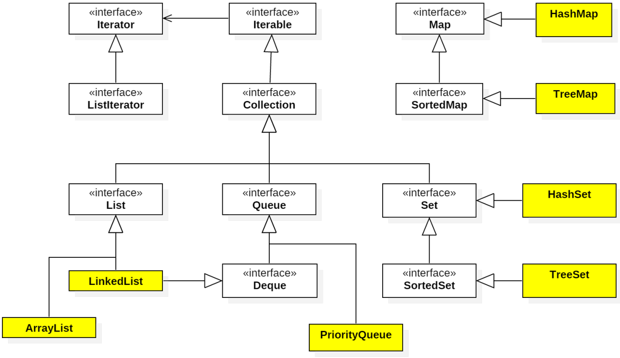


Map of Smiley to Character

The *SortedMap* extension requires the set of keys be sorted. Methods like `firstKey()` and `lastKey()` are added. The *Map* interface has two major implementation classes (similar to *Set*) — *HashMap* which uses a hash table DS for the implementation (with similar $\mathcal{O}(1)$ performance for put/get operations), and *TreeMap* which implements *SortedMap* in the similar to *TreeSet* way (with $\mathcal{O}(\log_2 N)$ efficiency). Example — MapTest.java.

Collections: summary of types

The interfaces of the Collections Framework and *some* of their implementations:



Note The above figure doesn't show that *LinkedList* also implements *Deque* which, in turn, extends *Queue* (this changes to JFC have been made after the above picture was created).

Implementations

Maximum Performance Data Structures used in the implementation

		Implementations		
Interfaces	**Hash table**	**Resizable array**	**Tree**	**Linked list**
`Set`	`HashSet`	Makes no sense	`TreeSet`	Makes no sense
`List`	*?*	`ArrayList`	*?*	`LinkedList`
`Deque`	*?*	`ArrayDeque`	*?*	`LinkedList`
`Map`	`HashMap`	Makes no sense	`TreeMap`	Makes no sense

Only a few data structures are fundamental; ultimately they represent the layout of memory and access to it in the computer architecture of the *von Neumann machine*:

1. resizable arrays — random access linear data structure, used in BookListWithArray.java
2. linked list — sequential access linear data structure, used in BookListWithLL.java
3. binary tree — recursive data structure with two (or more) self-references
4. hash table — hybrid data structure

Algorithms: Standard operations

Generic algorithms, which operate on the collection classes are provided as static methods from *Arrays* and *Collections* (not to be confused with *Collection* interface) classes. The *Collections* methods include (method return values and parameters are bounded types, actually; consult the API)

- `public static <T> min(Collection c)` — returns the smallest element in the collection (another overloaded version also which accepts a *Comparator* object)
- `public static <T> max(Collection coll)`
- `public static Comparator reverseOrder()` — returns a `Comparator` which reverses the natural ordering of the collection
- `public static void reverse(List list)` —— reverses the order of a list
- `public static void shuffle(List list)` — randomly shuffles a list
- `public static void fill(List list, E el)` — replaces each element of a list with `el`
- `public static void copy(List destination, List source)`
- `public static List nCopies(int n, E el)` — returns an immutable list that contains `n` copies of `el`
- `public static void sort(List list)` and `sort(List list, Comparator c)`
- `public static int binarySearch(List list, K key)` — the list must be sorted already for this method to work

Algorithms: examples

Various methods to manipulate arrays (such as sorting and searching) are contained in *Arrays* class. When combined with *Collections* class methods, one can achieve powerful results with just few lines of code. The following program (from *Java Tutorial*) prints out its arguments in alphabetical — natural for *String*s — order (another example is Anagram.java):

```
public class Sort {
   public static void main(String[] args) {
      List<String> list = Arrays.asList(args);
      Collections.sort(list);
      System.out.println(list);
   }
}
```

Two examples — a list with 100 elements each initialised to an Integer of value `-1` and the method `Array.asList()` (used above) returns a *list view of the array*:

```
Integer init = new Integer(-1);
List values = new ArrayList(Collections.nCopies(100,init));
Card[] cardDeck = new Card[52];
List cardList = Arrays.asList(cardDeck);
```

Because `cardList` is only a view of an underlying array, you can set the values of elements but you cannot add or remove them. The list view is useful if you want to pass a simple array to methods of the *Collections* class.

Reiterate

Major advantages of using the JCF (repeating the *Java Tutorial*)

- **Reduces programming effort** by providing useful data structures and algorithms so you don't have to write them yourself
- **Increases performance** by providing high-performance implementations of useful data structures and algorithms. Because the various implementations of each interface are interchangeable, programs can be easily tuned by switching implementations
- **Provides interoperability between unrelated APIs** by establishing a common language to pass collections back and forth
- **Reduces the effort required to learn APIs** by eliminating the need to learn multiple ad hoc collection APIs
- **Reduces the effort required to design and implement APIs** by eliminating the need to produce *ad hoc* collections APIs
- **Fosters software reuse** by providing a standard interface for collections and algorithms to manipulate them

No need to invent a wheel every time, the best ("well rounded") wheels have been already invented! Use *them*.

Copies and Views

Collections can grow very large and consume large memory.

- Same collection object can be used by different clients with different needs in terms of how data represented by the collection are used (read, modified).
  - Some clients must be prevented from modifying the data
  - Some clients must ensure that the data is protected from concurrent access/modification
  - Some clients may require different interface to the data (*eg*, when irrelevant data are filtered out to simplify the use of remaining ones)
- Using different variables to stand for the same collection – **aliases** — does not provide an acceptable solution (aliasing achieve nothing)
- **Copying** the data for different clients may be too expensive and undesirable (which copy must be regarded as primary when it comes to data preservation?)
- Instead, different **views** of the same collection object offer the right solution
  - Views fake the fact that underlying data are shared
  - Views provide different interface to the data and prescribe different protocol to how the data is used. *Metaphor:* interface to *Google* and similar "global Internet data companies" for normal clients, you!, and security services, like "FBI" and "CIA", when they need to get the data on you. The data is one and the same, but normal clients are not given full access to it, and their ability/efficiency to search it is impaired, and their ability to modify it is very restricted; not so for the spooks! (Julian Assange)

Views are normally created by calling a static *factory method* with the underlying collection object passed as a parameter. Technically speaking: "A view is an object of a class that implements one of the interface types in the JFC, and that permits restricted access to a data structure (CH)." A view object is a shallow copy of the original *Collection* object (*ie*, a modifiable view can be used to modify the original).

Collections Views

Some operations declared in *Collection* interface are marked as *optional* (`add()`, `clear()`, `remove()` and some bulk operations). This is because for certain implementations of the interface, these operations are deemed "destructive" — they attempt to modify the collection, and, if this is not consistent with the collection contract, they are *not* supported and their invocation throws *UnsupportedOperationException*.

```
boolean remove(E o) { throw UnsupportedOperationException; }
```

Also, "…some *Collection* implementations have restrictions on the elements that they may contain, like prohibiting null elements, and some have restrictions on the types of their elements. Attempting to add an non-eligible element throws an unchecked exception, typically `NullPointerException` or `ClassCastException`. Attempting to query the presence of an ineligible element may throw an exception, or it may simply return false", depending on implementation (*Java API docs*).

The reason for including optional operations in the first place is to increase re-usability and flexibility of the *Collection* types.

The mechanism of *views* increases the number of concrete JFC classes by an integer factor (number of views). The different views — *normal* (unsynchronised), *synchronised*, *unmodifiable* and *checked* wrappers — that can be obtained from a *Collection* (or *Map*) type as three new sorts of collections (or maps) except that they are supported by an underlying (`Map`) data structure. Wrapper implementations delegate all their real work to a specified collection but add extra functionality on top of what this collection offers. This is another example of the *Decorator* pattern.

Collections Wrappers

The wrapper views are obtained via static *factory methods* (methods which return newly created objects) defined in *Collections* class of `java.util` package. The views and their advantages are:

- the *synchronised* wrappers which add automatic synchronisation (thread-safety) to an arbitrary collection:

  ```
  Collections.synchronizedCollection(Collection c)
  Collections.synchronizedList(List l)
  ... similar for Set, SortedSet, Map, SortedMap
  Map map = Collections.synchronizedMap(new HashMap());
  //Vector class is synchronised by default, it's slower then ArrayList
  ```

- the *unmodifiable* wrappers which can be obtained via `unmodifiableCollection()` and similar calls; all modifying methods in this view throw *UnsupportedOperationException*
- the *checked* wrappers are needed when working with a legacy code, which uses the *raw* (non-generic) collection types; they are obtained via `checkedCollection()` and similar calls; checked view of the original raw collection will make a runtime check to enforce the type safety lost in raw types
- the *concurrent* collections are not standard wrappers, but are types defined in a separate package `java.util.concurrent`; they provide implementations which are not only thread-safe for synchronised access, but are specially designed to support multi-threaded use, like ensuring that a collection becomes non-empty before a request to access it is carried out.

Where to look for this topic in the textbook?

- Oracle's Java Tutorial Chapter on (Collection) Implementations