COMP6700/2140 Abstract Data Types

Alexei B Khorev and Josh Milthorpe

Research School of Computer Science, ANU

May 2017

Topics

1. (Abstract) Data Types and Data Structures
2. Interfaces **are** ADTs
3. Interfaces of *JFC* (recap of the container hierarchy)
4. Implementation Classes
5. Generic Algorithms

Data types and data structures

To group multiple elements into a single unit, known as a *collection or container*, to store and retrieve the elements, and manipulate this group as a whole through a small set of operations, defined in its interface, is a common computational problem. The most basic examples of such collection types are *arrays*. Classes like *Vector*, and *Hashtable* are separate (*raw*) collection types (they predate *JCF*). With generics, collection types are *vastly* expanded and organised into the *Java Collections Framework*, endowed with the architecture for representing and manipulating its constituents. JCF tries to emulate the famous `C++` *Standard Template Library*. The framework resides in `java.util` package. It has the following components:

- **Interfaces**: These are abstract **data types** that *represent* interfaces to the framework's collections. Interfaces allow collections to be manipulated independently of the details of their representation. Interfaces generally form a hierarchy.
- **Implementations**: These are the concrete implementations of the collection interfaces. In essence, they are reusable **data structures** (DS).
- **Algorithms**: These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. An algorithm is *generic* if it can be used on many different implementations of the appropriate collection interface. The use of bounded wild cards allows to constraint possible type values to ensure support of necessary operations. Java can implement generic algorithms because it can define parameterised types (it's a **third benefit** of having them in the language; remember the first two?). Generic algorithms provide **reusable functionality**.

Quotes from the Masters

- *“Bad programmers worry about the code. Good programmers worry about data structures and their relationships.”* [Linus Torvalds]
- *“Smart data structures and dumb code works a lot better than the other way around.”* [Eric S. Raymond]

Collection Metaphors

Kent Beck (a well known software expert, the co-author of *JUnit*, and one of the *Extreme Programming*'s pioneers) remarks, in his recent book "Implementation Patterns", that collections should be regarded as the first-class language construct (alongside with variables), because they *are* multi-valued variables, and because their cardinality (**"many"**, meaning two and more), is, in fact, complimentary to other two fundamental mathematical cardinalities: **zero** (no field) and **one** (a usual field variable). The collection concept combines in itself three metaphors: it's an *object*, which can hold *multiple values* at the same time, and has *properties of a set*.

Collections which you can encounter in a program can be classified according to their interface:

- Array — simplest, most rigid, fixed size, and fastest
- Iterable —– most basic interface to use a collection for iteration and nothing else
- Collection (proper) — allows adding, removing and testing for elements
- List — collection whose elements are ordered and accessible by their location in the collection
- Set — a collection with no duplicates
- SortedSet — an ordered collection which contains no duplicates
- Map — a collection where elements are stored and retrieved not by an index, but via a key

Collections

To put it simply — *Collection* objects store other objects (*eg, ArrayList*). We shall:

1. re-enforce the OO idea of a separation between the *interface* to a collection and the *implementation* (DS) of that collection
2. introduce (or review, in the case of array) new

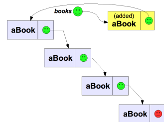
Collection Data Types	**Data Structures**
lists	arrays
queues	linked lists
stacks	hash tables
maps (dictionaries)	binary trees

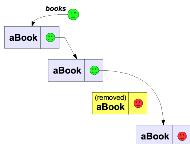
3. describe how collections are defined in *Java Collections Framework*

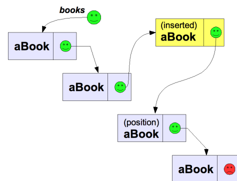
Collection interface

Before considering other implementations of the BookList.java interface, let us describe its *basic* operations. These basic operations (alongside with the *bulk operations*) form the Collection<E> interface. The *BookList* interface is therefore a particular example of Collection<Book>. The linked list implementation of three of its methods are illustrated:

int size()	boolean isEmpty()	boolean contains(Book b)
String toString()	Book get(int p)	Iterator<Book> iterator()
boolean add(Book b)	boolean remove(Book b)	void insert(Book b, int p)





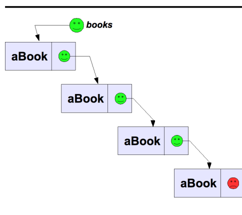


All implementations of iterator() returns the implementation of *Iterator* given by the class BookListIterator.java.

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).

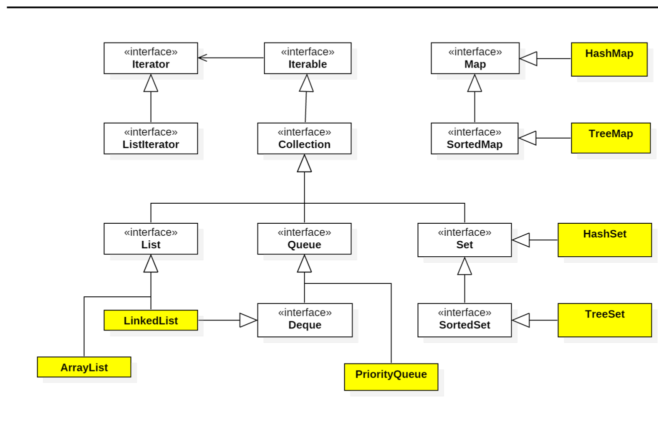
1. The BookListWithLL.java implementation uses a *linked list*. It is an example of a *dynamic* DS (size changes during runtime). Each *node* contains a `Book` object and a *link* to the next node. Types like this are *self-referential* — they contain a reference to a same type object. There is no limit 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, $\mathcal{O}(i)$.
2. The second implementation BookListWithArray.java uses an `array` DS, which can provide random access to a its elements, $\mathcal{O}(1)$. The implementation is simpler.



We have implemented the *BookList* interface from scratch. It is more practical to use data structures provided by the JFC, as we discuss later.

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).

Where to look for this topic in the textbook?

- Hortsmann's Core Java for the Impatient, Ch. 7.1–7.4
- Oracle's Java Tutorial First Three Sections in Collections Chapter
- The *Collections* Chapter (like others *trails* from the *Tutorial*) is available as an *epub* eBook