Computer Science 331

Abstract Data Types, Interfaces, and the Java Collections Framework

Mike Jacobson

Department of Computer Science
University of Calgary

Lecture #8

Outline

1. Abstract Data Types and Interfaces
   - Abstract Data Types
   - Interfaces
2. Java Collections Framework
   - Introduction to the Java Collections Framework
   - Notes on the Use of Standard Libraries
   - More About This Course and The Textbook
3. Reading Assignment

Abstract Data Types

Recall that a **data type** is defined by

- (a set of possible) *data values* and their *representations*
- *operations* defined on the data values and the *implementations* of these operations as executable statements

A *specification of requirements* for a data type is given by an **abstract data type (ADT)**

An *implementation* of a data type is given by a **data structure**

Specifying an Abstract Data Type

A specification of an ADT includes the following **components**:

- **ADT Name:** The name of the ADT
- **ADT Description:** Brief description (ideally written in simple English) of the ADT's characteristics and purpose
- **ADT Invariants:** Conditions that must be satisfied
  - *immediately after* all ADT *constructors* have terminated
  - *immediately before* all *other* ADT operations begin execution and *immediately after* these operations have terminated

  Note that these conditions are *not* necessarily satisfied *during* the execution of ADT operations
  These are also called *class invariants* in OOP literature

Specifying an Abstract Data Type (cont.)

Additional ADT Components:

- **ADT Attributes:** Pieces of information that must be available in order for the ADT to work properly (and are maintained *by* instances of the data type specified by the ADT)
- **ADT Operations:** Specifications of procedures that define the *behaviour* of the ADT and its *interface* with the rest of the system

These are more formal than described in your textbook. In this course, we will primarily consider Name, Description, and Operations.

Example: List

A `List` is a collection of data that supports the following operations:

- return the size of the list
- return *i*th element in the list
- determine whether a data item is in the list
- etc... See Ch.4 of the textbook for more details.

Some data structures that can be used to implement the List ADT are:

- static array (data items allocated together in memory, accessed by indexing)
- dynamic array (resizes itself as neccessary) — see `ArrayList` class
- linked list (exactly one entry per item, chained together via references) — see `LinkedList` class

Interfaces

In Java, an **interface** is . . .

- an extreme case of an “abstract class:” An interface can define **constants** (i.e., “class variables”) and **abstract methods**, but it *cannot* include any instance variables or implemented methods
- used to represent an abstract data type

A class *implements* an interface:

- use the `implements` clause with a class to show that your class provides an interface’s methods (checked by the compiler)
- used to represent a particular data structure

Section 1.3 includes a simple example of an interface. Later chapters include considerably more complicated (and useful) examples

More About Interfaces

Other abstract and concrete classes that “implement” the interface must provide the operations specified by the interface with *exactly* the same syntax

Note: It is customary, and useful, to include comments that specify the “semantics” of the operations (giving their requirements in more detail) as part of an implementation — but these details are *not* checked by Java!

It is possible for a class to implement more than one interface; this is Java’s (only) support for multiple inheritance

Example: Using the `List` Interface

Suppose we have two classes that implement `List`:

```
public class LinkedList<T> implements List<T> {...}
public class ArrayList<T> implements List<T> {...}
```

We can declare references of type `List<T>`

- separates implementation of `List` from its ADT definition (implementation is almost completely transparent)

Example (instance of `List` using a linked list implementation):

```
List<Integer> L = new LinkedList<Integer>();
Integer x = new Integer(5);
L.add(x);
```

Generics in Java

Recent versions of Java permit data structures of a *generic* type. For example:

- `List<T>` denotes a list whose elements are all of some reference type `T` (i.e. only classes, no primitive types)
- the statement `List<Integer> L` declares `L` to be a reference to a list of `Integers`s.

Generics facilitate code re-use (eg. don't need separate implementations for lists of strings and lists of integers).

More information in the text (eg. Section 4.1)

- We will use generics in this course by necessity, especially when working with the Java Collections package, but will try to keep this to a minimum.

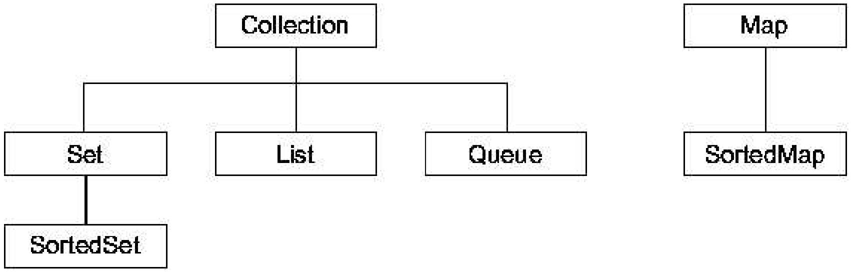
Collection Frameworks

A **collections framework** is a software architecture consisting of the following

- A hierarchy of **interfaces** that define various kinds of collections and specify how they are related
- A set of **abstract classes** that provide *partial* implementations of the interfaces and serve as the foundation for constructing **concrete classes**
- A set of **concrete classes** based on different underlying data structures that offer different runtime characteristics
- A set of **algorithms** that work with these

Java Collections Framework

The **Java Collections Framework** provides implementations for a number of common collections, including lists, maps, sets and vectors. It currently includes the following hierarchy of *interfaces*



Collection

Set

List

Queue

SortedSet

Map

SortedMap

Additional information about this can be found in Section 4.8 of the textbook. Considerably more information is available online.

Ways To Use Standard Libraries Like the JCF

One Approach: Build Everything From Scratch . . .

- In other words, don’t use the libraries at all!
- **Advantage:** You don’t have to depend on someone else’s implementation of something that you use
- **Disadvantage:** Development is more time-consuming, expensive, and, potentially, error-prone
- **Analogy:** Building a house by fabricating *everything* that you need instead of purchasing standard materials off-the-shelf
- Older data structures textbooks focus almost entirely on this approach, because useful “standard libraries” were not available when they were written!

A Second Approach

Use Libraries in a Limited Way

- In particular, understand what the libraries provide and make use of this in a straightforward way . . .
- . . . *without* trying to provide additional interfaces, abstract classes, and concrete classes that **extend** the library
- **Advantage:** The current project is likely completed more efficiently and reliably than using the first approach, *provided* that the library is already well-suited to it
- **Another Advantage:** Design and coding is (still) reasonably straightforward
- **Disadvantage:** You lose the ability to customize and extend the library in a way that simplifies development of your own *future* projects

A Third Approach

Use and Extend These Libraries

- Build components that will likely be of use in future projects
- Implement and test these in a way that facilitates reuse. . . for example, planning for the likelihood that inheritance hierarchies will be extended in ways you do not know about
- **Potential Advantage:** The library will gradually become more suitable for *your* application area
- **Potential Advantage:** Future projects will be completed more efficiently and reliably than would otherwise be possible
- **Disadvantage:** Implementation and testing (of the components to be added to the library) can be *considerably* more complicated than would otherwise be the case!

Expectations for This Course

You will be able to “build from scratch,” and you will occasionally be asked to do so on assignments and tests, because

- this is a very effective way to learn *about* the data structures that are being discussed, and
- programming tasks that are involved with this will reappear (in more complex forms) in the future, anyway!

You will be able to make (limited) use of standard libraries without necessarily being able to extend them, because

- You should get into the habit of *using* these libraries instead of “re-inventing the wheel” as soon as possible
- You will discover (very quickly) that you simply *do not have time* to solve the problems and design the software that you need to if you try to build everything from scratch

Reading Assignment

Please read the following. You may ask questions about this material in tutorials.

- *Read Chapter 1* if you have not already done so! Virtually everything here will, eventually, be needed in this course.
- *Read Chapter 4* In addition to the descriptions of the List ADT and various implementations, it is worthwhile to the know the information about the “Collection” interface (Section 4.8), and about iterators (Section 4.5)

Note: *Lectures*, after a discussion of the basic list data types, will continue with a discussion of **stacks**, which are discussed in Chapter 5. However, you need to know a little bit about the “List” interface in order to make sense of the material in this chapter of the book.