Computer Science 331

Data Structures, Abstract Data Types, and Their Implementations

Mike Jacobson

Department of Computer Science
University of Calgary

Lecture #8

Outline

What This Lecture is About

Significant concepts defined in the first lecture:

- *Data Type:* defined by
  - Data values and their representation
  - Operations defined on the data values and the implementation of these operations
- *Abstract Data Type:* In essence, a "specification of requirements" that is satisfied by a data type
- *Data Structure:* Provides a representation of the data values specified by an ADT

  Together with *algorithms* for an ADT's operations, this provides an *implementation-independent* description of a data type

Goal for Today: Discussion of support for these in Java

Information Hiding

Assumption:

- Everyone in this class has already been introduced to the basic principles of *object-oriented development*...

  ...although this introduction has, sometimes, been quite brief.

- One Very Important Idea: *Information Hiding*
  - Allows various implementation decisions to be made gradually, in a "piecemeal" fashion
  - *All* external access to the information maintained as part of data type must be made using the data type's operations
  - *Consequence:* "The rest of the system" does not need to know how the data type is represented! ...and this data type, and "the rest of the system," can be developed independently

Example Data Type: A Simple Counter

Consider a "simple counter" used to keep track of information about the current time, or progress toward some goal

Data Values:

- `limit`: A positive integer — one more than the maximum value this counter can represent. We will assume (or require) that this value is small enough to be represented using Java's `int` primitive data type — so that

$$0 \leq \texttt{limit} \leq 2,147,483,647$$

- `value`: The current value being represented, i.e., an integer between 0 and $\texttt{limit} - 1$ (inclusive)

Representation:

- One might simply represent these by a pair of variables with names `limit` and `value`, respectively

Example Data Type: A Simple Counter

Operations Might Include...

- Creation: Set `limit` to be a given integer value (throwing an `IllegalArgumentException` instead, if the supplied value is negative or zero) and set `value` to be zero
- Access: A method should be available to report the `limit`
- Access: A method should be available to report the `value`
- Modification: An `advanceValue` method should increment `value`, throwing a `LimitReachedException` if this would cause `value` to be equal to `limit` and setting the value of `value` back to 0 in this case

Once implementations of these methods are supplied, the description of this "data type" would be complete.

Implementation as a Class

A *class* can be provided to implement this data type.

Implementation Details:

- All instance variables (eg. forw `limit` and `value`) should be **private** — only be accessible through the class's *methods*
- Operations that create a new element of this data type that a program will use (that is, create a new *object* in this *class*) should be implemented as *constructor* methods...
- Operations that report information about some element of this data type (ie, about an object in this class) should not modify it as well... and should be implemented as *accessor* methods....
- Operations that *change* — that is, *modify* some object should implemented as *modifier* or "*mutator*" methods....
- ...and, yes: With rare exceptions, each public method in a class should be one of the above three types!.... but *only* one

Class Invariants

At this point, information about our class that is available to the rest of the world includes

- the *names* we have given to the public methods we have provided for use, as well as
- *signatures* for these methods.

In general this fails to include — or clearly convey — info about the acceptable ranges of values, and required relationships, for values represented by private instance variables for our class.

Eg: For our "Counter" example, it is not necessarily clear that...

- `limit` is an integer (whose value will not be changed) that is positive and and can be represented using Java's `int` data type
- `value` is a nonnegative integer that is less than `limit`.

Class Invariants

Definition: A *class invariant* is an assertion about the information that is maintained by each object in the class.

Properties:

- The class invariant must be satisfied whenever the use of a *constructor* method results in the creation of a new object.

  Thus the class invariant should be implied by every constructor's *postcondition(s)*.

Class Invariants

Properties, Continued:

- The class invariant may be assumed to hold immediately before the execution of any other public (accessor or mutator) method begins. It should therefore be part of every such method's *precondition(s)*.

  The class invariant does *not* necessarily hold while the execution of a mutator method is in progress.

  However, the class invariant must hold, once again, when every public method *terminates*. It should therefore be part of every accessor and mutator method's *postcondition(s)*.

Class Invariants

Class invariant for our "simple counter:"

a) `limit` is a positive integer that can be represented exactly using Java's `int` data type

b) `value` is an integer whose value is between 0 and `limit` − 1, inclusive

A `SimpleCounter.java` file implementing this class — and including this class invariant — will be provided for students to examine and use.

Composition

It is possible that the objects in a class *have* an instance (or even multiple instances of) another class as a component(s)

Example: Consider a `TimeOfDay` class whose objects can be used to represent times during a day, using a 24-hour clock, measured in hours, minutes, and seconds.

Each instance of (ie, object in) the `TimeOfDay` class *has* three instances of our "simple counter" class as *components*:

- `seconds:` a simple counter with `limit` equal to 60
- `minutes:` a simple counter with `limit` equal to 60
- `hours:` a simple counter with `limit` equal to 24

Note: In this kind of relationship, the "components" do not have any kind of independent identity, themselves: They are only accessed indirectly, through the larger class's operations

Aggregation

This is another kind of "has a" relationship between objects.

The chief difference between *aggregation* and *composition* is that, when "aggregation" is used, the "component" object *does* have an independent identity — and can be accessed directly by other classes and methods

Example: This relationship is used to define a *linked list* of objects of the same class — each object in the list (except the final one) *has a* next object that follows it.

Note: We will use aggregation quite often in this course, because it is needed to implement recursively defined (and "hierarchical") data types.

Inheritance

Another important relationship between classes is an "is a" relationship: One class *extends* another, "inheriting" all the attributes of the original

Example: the Exception Class Hierarchy

- Classes `Error` and `Exception` both `extend` the class `Throwable` — so that an `Error` object "is a" `Throwable` object, too
- Classes `RunTimeException` and `IOException` both `extend` the `Exception` class — so that a `RunTimeException` object "is an" `Exception` object, too.

  Note: It follows from this that a `RunTimeException` object "is a" `Throwable` object, too.

Note: In CPSC 331 we will *use* libraries of classes that have been developed using inheritance. You will not need to use inheritance to define classes when solving problems in this course (except possibly for a few very limited examples).

Abstract Classes

An *abstract class* is a special kind of class that includes declarations of one or more methods without providing implementations of these methods. These methods are called *abstract methods*.

Such a class cannot have any objects of "its own."

However, other regular (*concrete*) classes can (and generally do) *extend* an abstract class, providing implementations of all the abstract methods whose declarations have been inherited — and these concrete classes *can* have objects.

Interfaces

In Java, an *interface* is . . .

- an extreme case of an "abstract class:" An interface can define *constants* (i.e., "class variables" — declared as both `static` and `final`) and *abstract methods*, but it cannot include any instance variables or implemented methods
- used to represent an abstract data type

CPSC 331 students will be expected to write their own interfaces, and use existing interfaces, to solve problems in this course.

Interfaces: Additional Notes

- Other abstract and concrete classes that "implement" the interface must provide the operations specified by the interface with exactly the same syntax
- 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.

  We will also frequently provide an "interface invariant" (which must be implied by the class invariant of any class that implements this interface).

  However, 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

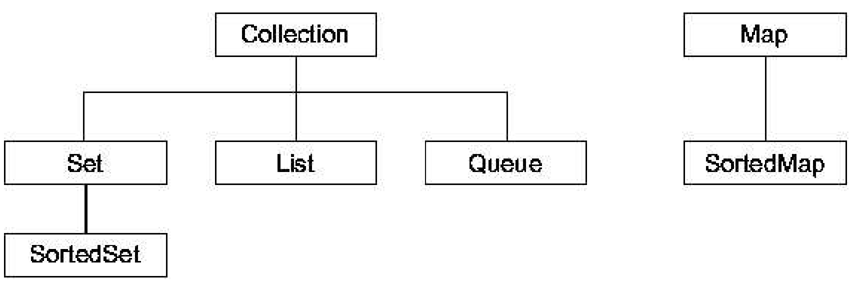
Collections Frameworks

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. The version provided with Java 5 included the following hierarchy of *interfaces*.



Of course, we are now using Java 6 — and various details have changed! Additional information will be provided later on.

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!

Ways To Use Standard Libraries Like the JCF

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

Ways To Use Standard Libraries Like the JCF

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

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

Extending libraries might be discussed *briefly*, but not in detail.

An Array of *What*?

If you want to, you can...

1. Maintain a sorted array of *integers*, or
2. Maintain a sorted array of *reals*, or
3. Maintain a sorted array of *strings*, or...

Essentially the same algorithm can be used to *sort* the array in each case, and essentially the same algorithm can be used to *search* in the sorted array, too.

Java therefore provides (and allows us to develop) *generic types* as well as *generic methods*.

We will need to know more about these in use the Java Collections Framework effectively... and it turns out to be better to use an `ArrayList` instead of a *array* here...

... but that's enough for today!

Reading Assignment

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

- *Sections 1.1 and 1.2*
- *Chapter 2*. While we did not follow the same order of topics, you should see that all the material in this chapter of the text has now been covered.

Lectures will continue with material that is discussed in *Chapter 3* — which introduces both generic programming and the Collections Framework.