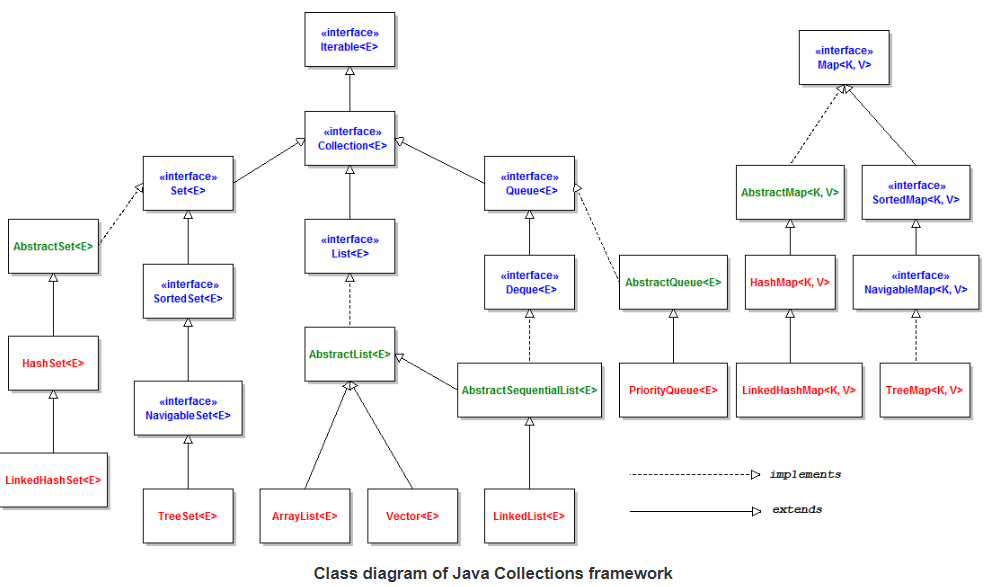
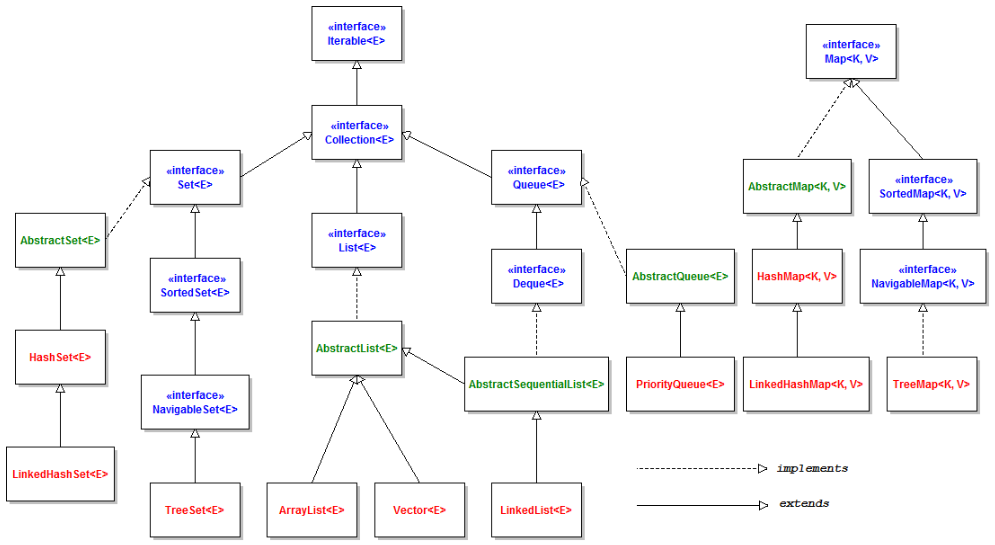
COP 2251 – Java Programming II – Lists, Stacks, Queues - Chapter 20

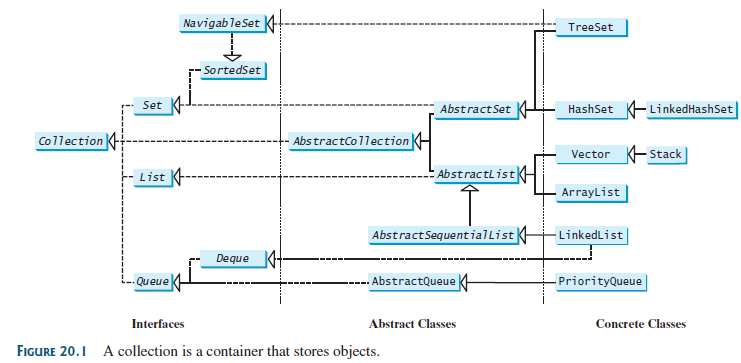
* Collections are objects that hold and organize other objects, usually of the same type.
* The objects in a collection are called the elements of the collection.
* A collection cannot have primitive data types as elements, only reference types (objects).
* Wrapper classes like Integer and Double are used to store numbers in a collection.
* The **java.util** package contains the “Collections Framework” (**CF**) to manage collections.
* In Java 1.5, the interfaces and classes of the Collections Framework were updated for generic data types.
* The Java CF consists of:
  + Interfaces that can be implemented to create collections.
  + Implementations (concrete classes) of these interfaces.
  + Static Methods that perform common tasks on collections.
* There are 3 main kinds of collections:

1. **Sets**: collections of elements with no duplicates.
2. **Lists**: ordered collections of elements with duplicates possible.
3. **Maps**: collections of elements, each consisting of a key and value.



LEGEND: Interfaces Abstract Classes Concrete Classes





**The Collection Interface**

* As Figure 20.1 shows, this interface is the superclass of the CF.
* It contains core methods common to all collections.
* Study the Collection methods in the UML diagram in Figure 20.2 on page 778.
* NOTE: Rather than implementing the Collection interface, coders usually choose the best concrete class from the Java CF for the new application.

Try **TestCollection.java**.

**The Iterator Interface**

* This interface is used to pass through (iterate over) the elements of a collection.
* The Iterator interface has three methods. See again UML in Figure 20.2.
* Think of an iterator as a “pointer” between collection elements.
* The **hasNext()** method will return true if the collection contains another element.
* The imaginary pointer “hops over” and returns elements with **next()** method calls.
* The **remove()** method will remove the last element returned by the iterator.

Run **TestIterator.java**.

* Note the **iterator()** method that returns an instance of interface **Iterator**.
* Java 10 introduced **var** for declaring a variable. See page 781.
  + The compiler deduces its type from the context.
  + Don’t use it for primitves.

**The forEach Method** (page 782)

* This (new in Java 8) method makes it easy to process every element in a collection.
* Code it with a lambda as shown in Liang’s next example.

Run **TestForEach.java**

**The List Interface**

* A list is an ordered collection in which each element has an index (starting from 0).
* Duplicate elements and null values are allowed. All elements are of the same data type.
* The List interface extends Collection and includes its methods.
* List also defines its own methods. See the UML diagram in Figure 20.3 on page 783.

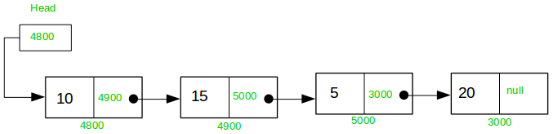
**The ListIterator Interface**

* The Iterator interface also has a **ListIterator** sub-interface for iterating forward or backward through a **list** type of collection.
* ListIterator adds **previous( )** and **hasPrevious( )** for traversing in either direction.
* See the methods of ListIterator in the UML diagram in Figure 20.4 on page 783.
* As an interface, List cannot be instantiated but it has concrete subclasses.
* Two commonly used concrete classes are ArrayList and LinkedList.

**ArrayList**

* This is a resizable array implementation (arrays cannot be resized) of the List interface.
* JDK 1.5 revised ArrayList to include generics. See Figure 20.5.
* ArrayList is very similar to **Vector** but Vector is thread-safe. ArrayList is not.
* It also allows capacity increments as Vectors do.
* ArrayList is best when you need to access an element by index, unless you need to insert or delete elements at the beginning or end of the list, when LinkedList is more efficient.
* ArrayList’s get(i) method is not efficient for traversing an entire list. Use an iterator.
* A foreach loop is efficient because it uses an implicit iterator. See boxes on page 786.

Try **ArrayListOne.java** (not in the book). Note the usage of ListIterator and forEach.  
LinkedList



* A linked list is an ordered collection created with the **LinkedList** class.
* Each element (except head) consists of data plus the address of the next element.
* A linked list can grow or shrink in size.
* Like an ArrayList, LinkedList allows insertions in the middle of the list.
* Duplicates are allowed as with all lists.
* Use the methods of the List interface to work with a linked list. See UML in Figure 20.6 on page 785.
* There are also doubly-linked lists as shown here.
* Linked lists can create other data structures like stacks, queues, trees, and dequeues.
* An iterator is more efficient for traversing a LinkedList than a loop with the get() method.
* Iterator doesn’t have an add() method, but **ListIterator does**.
* This add( ) method will add a new element **before** the iterator’s current position.

Try **LinkListOne.java** and **TestArrayAndLinkedList.java**.

**The Comparator Interface**

* The **Comparable** interface (Section 13.6) has a **compareTo()** method for comparing objects of the **same** class.
* The **Comparator** interface can compare objects that don’t implement Comparable or objects of two **different** classes!
* Comparator has the **compare()** method for this. See page 787.

Examine **GeometricObjectComparator.java.**

Then try **TestComparator.java**.

Try **SortStringByLength.java**

* It uses inner class MyComparator to compare Strings by length.

Try **SortStringIgnoreCase.java**

* This example does an alphabetical, case-insensitive sort of Strings with a nifty lambda.
* Note that the alternative syntax, a method reference using **::,** can be used (page 789)
* Interface Comparator has a static comparing() method that can execute a lambda expression or a method reference, as an alternative to making an inner class. See page 790. Then, try **SortStringByLength2.java** (not in book).
* The comparing method can also be used to make a Comparator that compares object properties. See again page 790.
* Class Comparator also has a thenComparing() method for secondary, or deeper, sorts. See page 791.
* Chaining the reverse() method to comparing() can reverse a sort. See page 791.

Examine class **Human.java** and then try **HumanSortByNameAndAge.java. Note** that the two peeps named M Jones are sorted by age and so are the two peeps named K Smith.

Static Methods for Lists and Collections

* Class **Collections** contains a large number of static methods for manipulating lists.
* See Figure 20.7 on page 792 for a UML diagram of these methods.
* Examine the numerous examples using these methods on pages 792-794.

Case Study: Bouncing Balls

Try **MultipleBounceBall.java**. It would be a good screen saver.

**Class Vector**

* This legacy class was updated for generics. It’s like an ArrayList except that its methods are synchronized to be thread-safe.
* If synchronization is unnecessary, ArrayList is more efficient.
* See UML for Vector in Figure 20.10 on page 799.
* Use Vectors when you need to avoid the corruption danger caused by multiple threads.
* Vectors can expand by a capacity that can be specified in the constructor.
* The line below creates a vector with an initial capacity of 100 Double elements that will expand by 10 elements when more capacity is required:

**Vector<Double> v1 = new Vector<Double>(100,10);**

Try **VectorOne.java**. This “extra” is not in the book.

**Class Stack**

* The Stack class is a last-in first-out (**LIFO**) stack of objects. It extends class Vector.
* When a stack is first created, it contains no objects.
* See the UML diagram in Figure 20.11 for the methods of the Stack class.

**peek()** looks at the top element of the stack without removing it.

**pop()** removes and returns the top element.

**push()** adds an element to the top of the stack.

Try **BookStack.java**. This “extra” is not in the book.

**The Queue and Deque Interfaces**

* A queue (4 consecutive vowels!!) is a first-in first-out (**FIFO**) collection.
* You have suffered in a queue at fast food eateries, banks, telephone support, etc.
* See Figure 20.12 on page 801 for the unique methods of Queue.
* As shown in Figure 20.13, class **LinkedList** is a concrete implementation of the Queue interface, so you can use it create queues in your programs.
* A **deque** (or dequeue) is a double-ended queue, with additional methods enabling inserts and deletes at both ends.
* The interface **Deque** is a sub-interface of Queue. See Figure 20.13.

Try **TestQueue.java**.

**Class PriorityQueue**

* This concrete class inherits from the Queue interface. See Figure 20.14.
* PriorityQueue uses natural ordering (alpha for strings) to sort its elements.

Try **PriorityQueueDemo.java**.

* First loop sorts in natural order (alpha) and second loop sorts in reverse alpha.

Case Study: Evaluating Expressions

* This interesting study uses two Stack instances to evaluate expressions.

Try **EvaluateExpression.java**. Use command line arguments in Eclipse to run it.