Ch 6 – The Collections API

Sections	Pages	Exercises
6.1-6.9	201-245	1, 2, 7

All code in these notes can be found in the dgibson/cs3410/06 folder on the public drive of the Math/CS network.

6.1 Introduction

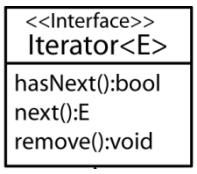
1. "A data structure is a representation of data and the operations allowed on that data." (Weiss). A data structure is a way of organizing data within the confines of a programming language. In the object-oriented sense, it is a class(es) that stores data and provides methods to access it. Data structures provide a public interface and a private implementation. This means that users of the class can only do the things specified in the interface. Typically, the user cannot directly access the stored data, but must use accessors and mutators, specified in the interface, which control access to the data. This is known as encapsulation (information-hiding). There are many data structures built-in to Java (or any programming language) such as an array, hashset, *etc*. We can also design our own data structures to meet custom needs. Frequently, these are encapsulations of existing data structures. For instance, Java's ArrayList encapsulates an array of objects. Thus, we may wrap existing data structures to create new ones. Finally, the choice/design of a data structure influences the performance (complexity) of an algorithm, so we must think about these carefully.

6.2 The Iterator Pattern

1. A (software) *design pattern* "...is a general reusable solution to a commonly occurring problem..." [Wikipedia, Design pattern]. A recurring problem in most software systems is the traversal of the elements in a collection. *How* you do this traversal depends on the underlying data structure used. For instance, if you were storing data in an array, ArrayList, or custom Employees class:

`for( int i=0; i<x.length; i++ ) x[i] = 0;`	Array
`for( int i=0; i<y.size(); i++ ) y.set(i,0);`	ArrayList
`for( int i=0; i<emps.getNumEmployees(); i++ ) {Employee e = emps.getEmployee(i);}`	Custom Employees class

So, a programmer must know a data structure's implementation details to iterate through the collection. The *Iterator Pattern* proposes a generic solution to this problem. It introduces the idea of an *Iterator* interface:



The idea is that all data structures should supply an *iterator* method that returns an *Iterator*. This greatly simplifies what the programmer has to know, just the two important methods, *hasNext* and *next*. For instance in Java, we can use the Iterator associated with the ArrayList class:

```
ArrayList<Employee> emps = new ArrayList<Employee>();
emps.add(...);
...
Iterator iter = emps.iterator();

while( iter.hasNext() )
{
   System.out.print( iter.next() );
}
```

All *Collection*s in Java implement the *Iterable* interface, which specifies just one method, *iterator():Iterator*. In other words, any *Collection* must have an *iterator* method that returns an *Iterator*. Any class that implements *Iterable* can also be traversed using the *for-each* loop:

```
ArrayList<Employee> emps = new ArrayList<Employee>();

for( Employee e : emps ) System.out.print( e );
```

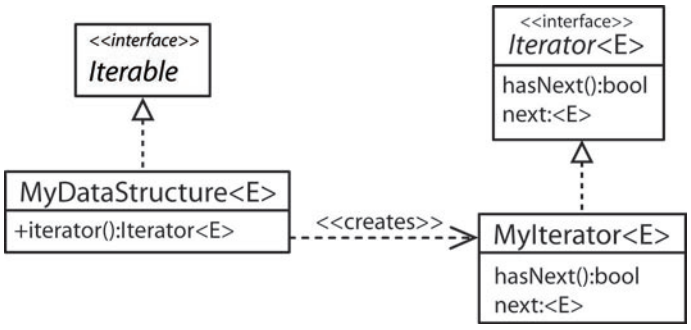
The *for each* loop is simply a short-cut for using the iterator. The compiler essentially converts the *for each* loop to an equivalent using the iterator.

2. An important design principle is, *Program to an interface, not an implementation*. When we write a standard loop to access the elements in an array, we write code like this:

```
for( int i=0; i<ary.length; i++ ) System.out.println( ary[i] );
```

Here, *i* is called the iterator object because it controls how the elements in the array are traversed. Notice that this is a very common problem: the traversal of elements in a collection. However, the solution above is coded directly to the implementation of the array. If the data structure above (array) changes, then we will have to change the iteration. When we use the *Iterator Pattern*, we program against an interface. Thus, the underlying data structure change without affecting the code.

3. So, what does it mean for a class to provide an *iterator* method? First, we notice that the *iterator* method requires the return of an *Iterator*, which is an interface. Thus, there is a *concrete iterator* that implements the *Iterator* interface. This concrete iterator will manage the iteration process. In other words, it will remember where you are in the iteration and how to obtain the next item. The important part is that the programmer (client) never sees the concrete iterator directly, it only sees it through the *Iterator* interface.



Thus, the client writes code like this:

```
Iterator iter = myDataStructue.iterator();

while( iter.hasNext() ) System.out.println( iter.next() );
```

4. We can use Java Iterators with generics:

```
Iterator<Blob> iter = dataStructue.iterator();

while( iter.hasNext() ) System.out.println( iter.next().blobMethod() );
```

5. Consider this example from the text. Notice that *MyContainer* is the data structure and it contains an array of Objects. The concrete iterator is *MyContainerIterator* which is created by passing a reference to the data structure.

```
1  package weiss.ds;
2
3  public class MyContainer
4  {
5      Object [ ] items;
6      int size;
7
8      public Iterator iterator( )
9        { return new MyContainerIterator( this ); }
10
11     // Other methods not shown.
12 }
```

figure 6.5

The `MyContainer` class, design 2

```
1  package weiss.ds;
2
3  public interface Iterator
4  {
5      boolean hasNext( );
6      Object next( );
7  }
```

figure 6.6

The `Iterator` interface, design 2

The concrete iterator simply contains a variable, *current* to keep track of the current element. Each time *next()* is requested, the variable is incremented.

figure 6.7

Implementation of the `MyContainerIterator`, design 2

```
1  // An iterator class that steps through a MyContainer.
2
3  package weiss.ds;
4
5  class MyContainerIterator implements Iterator
6  {
7      private int current = 0;
8      private MyContainer container;
9
10     MyContainerIterator( MyContainer c )
11       { container = c; }
12
13     public boolean hasNext( )
14       { return current < container.size; }
15
16     public Object next( )
17       { return container.items[ current++ ]; }
18 }
```

figure 6.8

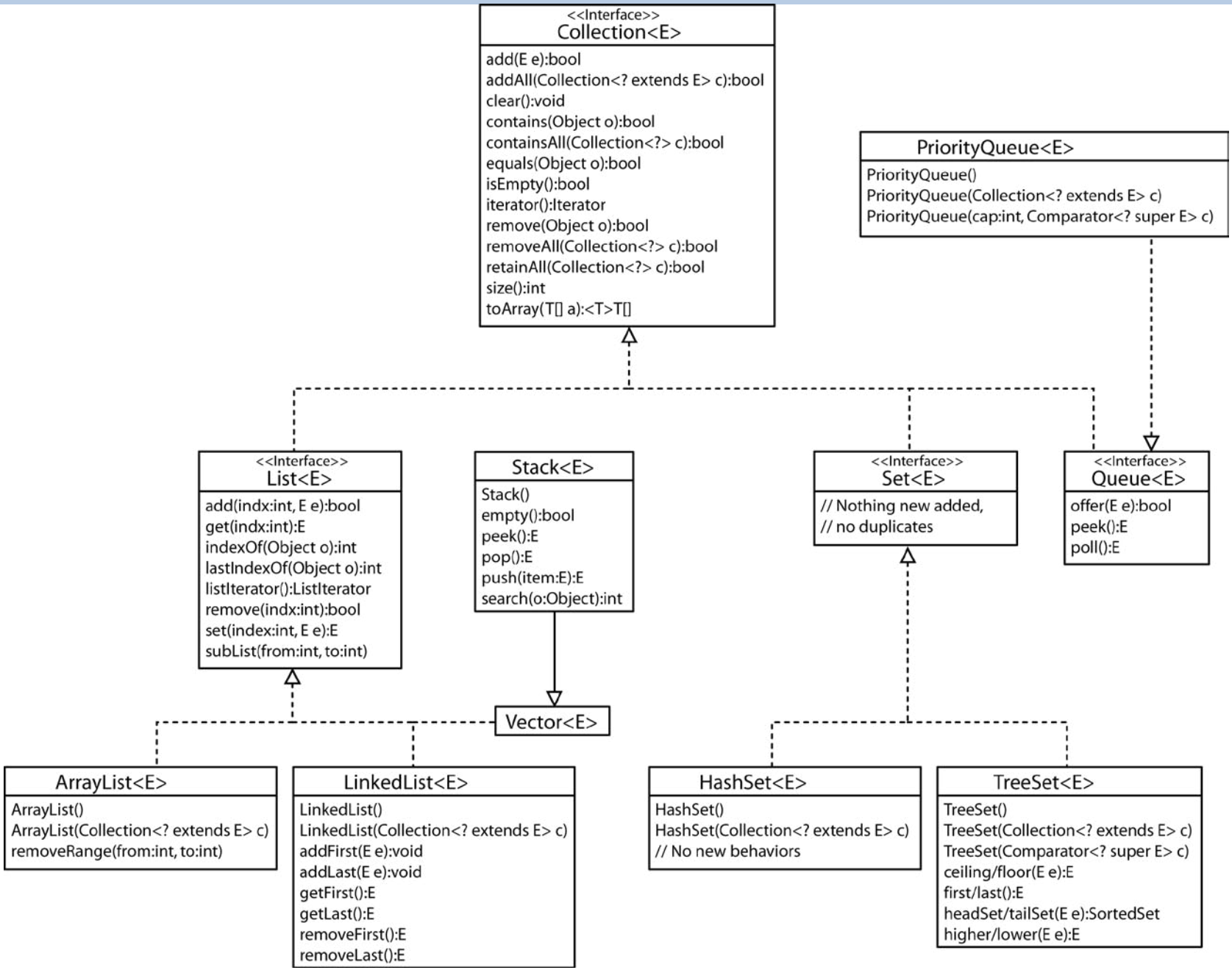
A `main` method, to illustrate iterator design 2

```
1      public static void main( String [ ] args )
2      {
3          MyContainer v = new MyContainer( );
4
5          v.add( "3" );
6          v.add( "2" );
7
8          System.out.println( "Container contents: " );
9          Iterator itr = v.iterator( );
10         while( itr.hasNext( ) )
11             System.out.println( itr.next( ) );
12     }
```

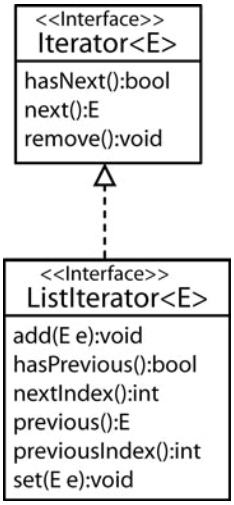
Homework 6.1

1. Consider figure 6.5. Replace: Object[] items with: List items = new LinkedList(). Now, rewrite the iterator defined in figure 6.7. In other words, write a simple container class that stores item internally as a linked list and implements the Iteratable interface.

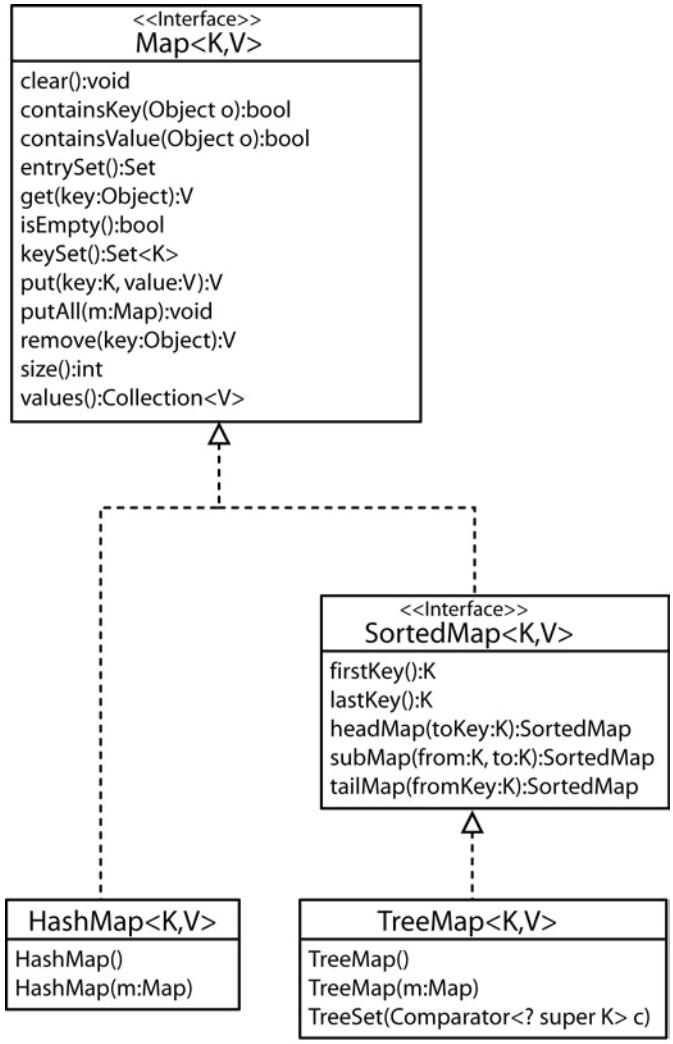
6.3 Collections API



Iterator Interfaces



Map Interfaces



Example – Generic *toArray* method

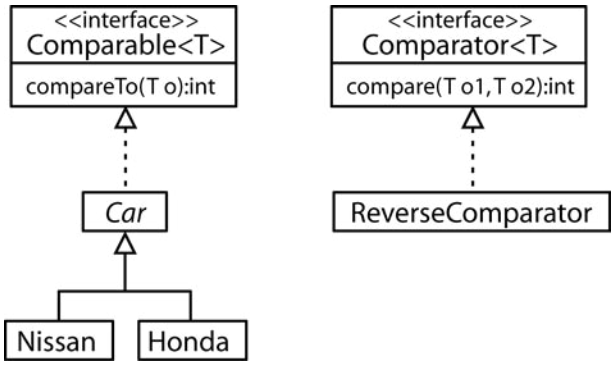
```
Honda h1 = new Honda(33);  Honda h2 = new Honda(44);

ArrayList<Honda> hondas = new ArrayList<Honda>();

hondas.add(h1);  hondas.add(h2);

Honda[] hAry = new Honda[1];

hAry = hondas.<Honda>toArray( hAry );

for( Honda h : hAry ) System.out.println( h );
```

Homework 6.2

1. Write the code required to define an immutable ArrayList class? Assume that the constructor accepts a Collection.

6.4 Generic Algorithms

1. We can use the Comparable Interface or the Comparator Interface to order objects.




```
abstract class Car implements Comparable<Car>
{
   public int speed;
   public Car( int i ) { speed = i; }

   public int compareTo( Car c )
   {
      return this.speed - c.speed;
   }

   public String toString()
   {
     return "car speed=" + speed;
   }
}

class Honda extends Car
{
   public Honda( int i ) { super(i); }
}

class Nissan extends Car
{
   public Nissan( int i ) { super(i); }
}


class ReverseComparator implements Comparator<Car>
{
   public int compare( Car c1, Car c2 )
   {
      return -(c1.compareTo(c2));
   }
}
```

And we can sort the objects based on "natural ordering" (*e.g.* defined by the object's implementation of Comparable) or by using an explicit Comparator (*e.g. ReverseComparator*).

```
Honda h1 = new Honda(33);
Nissan n1 = new Nissan(45);
Honda h2 = new Honda(22);

ArrayList<Car> cars = new ArrayList<Car>();

cars.add(h1); cars.add(n1); cars.add(h2);

// Based on natural ordering (e.g. Comparable)
Collections.sort( cars );

for( Car c : cars ) System.out.println( c );

// Based on custom Comparator
Collections.sort( cars, new ReverseComparator() );

for( Car c : cars ) System.out.println( c );
```

Note that the ReverseComparator uses the fact that Cars are Comparable (*e.g.* it uses the compareTo method). A Comparator is more general and doesn't have to use compareTo; it can order objects any way it wants to (there are technical considerations that are important). For instance, it could define Red cars as coming before Blue cars, *etc*.

Homework 6.3

1. Write a Comparator that can be used to sort cars on price (lowest to highest) and gas mileage (highest to lowest).
2. Write a Comparator that takes argument that determines whether Person objects should be sorted on age, either ascending or descending.

6.4 Generic Algorithms (continued)

2. In the example below, the author defines a *reverseOrder* static method in the Collections class that returns a Comparator that provides a reverse ordering. It utilizes the concept of a *nested class*. A nested class is declared inside another class and must be static; however, it can be instantiated. As shown here, it is a way to *hide* a class (private declaration, though public, protected, and package are allowed).

```
1  package weiss.util;
2
3  /**
4   * Instanceless class contains static methods that operate on collections.
5   */
6  public class Collections
7  {
8      private Collections( )
9      {
10     }
11
12     /*
13      * Returns a comparator that imposes the reverse of the
14      * default ordering on a collection of objects that
15      * implement the Comparable interface.
16      * @return the comparator.
17      */
18     public static <AnyType> Comparator<AnyType> reverseOrder( )
19     {
20         return new ReverseComparator<AnyType>( );
21     }
22
23     private static class ReverseComparator<AnyType> implements Comparator<AnyType>
24     {
25         public int compare( AnyType lhs, AnyType rhs )
26         {
27             return - ((Comparable)lhs).compareTo( rhs );
28         }
29     }
30
31     static class DefaultComparator<AnyType extends Comparable<? super AnyType>>
32                  implements Comparator<AnyType>
33     {
34         public int compare( AnyType lhs, AnyType rhs )
35         {
36             return lhs.compareTo( rhs );
37         }
38     }
```

figure 6.13

The `Collections` class (part 1): private constructor and `reverseOrder`

Homework 6.4

1. Write a snippet of code that uses the ReverseComparator in figure 6.13 above to order an ArrayList of Blob objects. You can assume that Blob implements Comparable.

6.4 Generic Algorithms (continued)

3. Example – Use of Comparable and a generic method:

```
public static <T extends Comparable<? super T>> T best( T obj1, T obj2 )
{
   return ( obj1.compareTo(obj2) > 0 ) ? obj1 : obj2;
}
```

To use:

```
Honda h1 = new Honda(33);
Honda h2 = new Honda(44);

Honda h = best( h1, h2 );

System.out.println( h );

Submarine s1 = new Submarine(33);
Submarine s2 = new Submarine(55);

Submarine s = best( s1, s2 );

System.out.println( s );
```

Where:

```
abstract class Car implements
                   Comparable<Car>
{
   public int speed;
   public Car( int i ) { speed = i; }

   public int compareTo( Car c )
   {
      return this.speed - c.speed;
   }
}

class Honda extends Car
{
   public Honda( int i ) { super(i); }
}
```

```
class Nissan extends Car
{
   public Nissan( int i ) { super(i);
}
}

class Submarine implements
                  Comparable<Submarine>
{
   public int depth;
   public Submarine( int i ) {
          depth = i; }

   public int compareTo( Submarine c )
   {
      return -(this.depth - c.depth);
   }

}
```

Homework 6.5

1. Explain the difference in these two declarations:

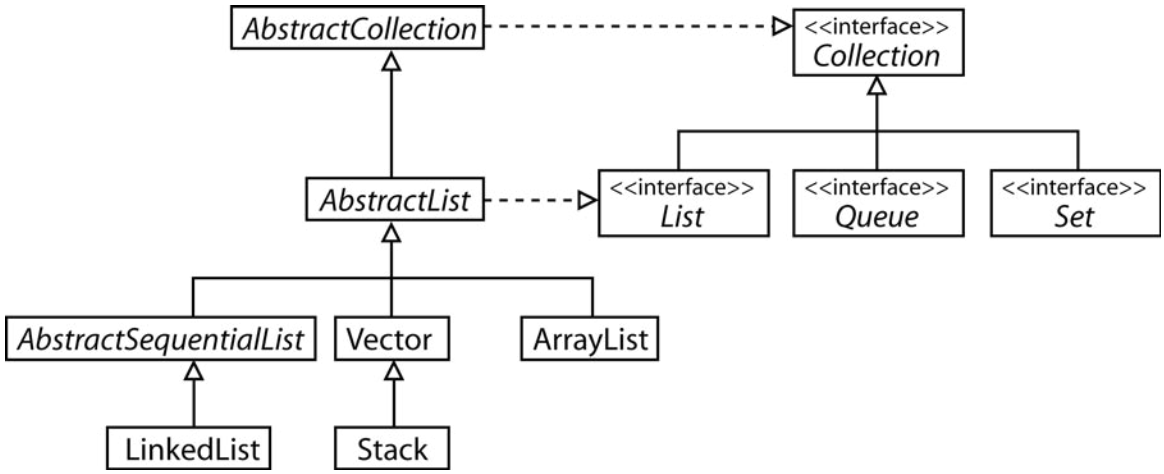
```
public static <T extends Comparable<? super T>> T best( T obj1, T obj2 )
```

and

```
public static <T extends Comparable<T>> T best( T obj1, T obj2 )
```

6.5 The List Interface

The four major implementations of the List interface are the ArrayList, LinkedList, Vector, and Stack.

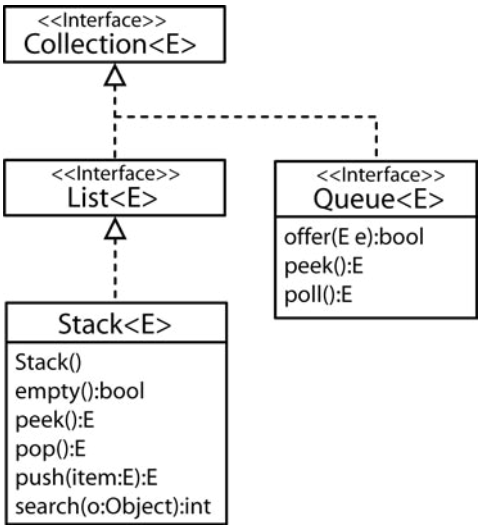


Homework 6.6

1. Explain the implementation of ArrayList.
2. What is the difference between an Iterator and a ListIterator?
3. How do you create a ListIterator that will traverse the elements in reverse order?
4. Suppose you have an LinkedList, *blobs* that contains Blob objects. Write a snippet of code that utilizes a single iterator to traverse the elements in forward order and then in reverse order.
5. Justify the complexity of the following operations on an ArrayList, *al*
   add(E e)
   add(int i, E e)
   remove(int i)
   remove(al.size()-1)
   contains(Object o)
   get(int i)
6. Explain the implementation of LinkedList
7. Justify the complexity of the following operations on an LinkedList
   add(E e)
   add(int i, E e)
   remove(int i)
   remove(al.size()-1)
   contains(Object o)
   get(int i)

6.6 Stacks and Queues

A Stack is a subclass of Vector which implements the List interface. A Stack is a data structure that in principle restricts access to the most recently added item. The Stack interface defines the methods: push (insert onto top of stack), pop (removes and returns the top item), peek (returns the top item, but does not remove it). These operations are constant time.



Another basic data structure is the Queue which is an interface in Java. In some sense, a Queue is the opposite of a Stack. A Queue allows access to the *least* recent item, *e.g.* the item that has been there the longest. For instance, jobs are submitted to a Printer Queue. When the printer becomes available, it takes the job that has been waiting the longest. The basic Queue operations are: offer/enqueue (adds to the end of the queue), poll/dequeue (removes and returns the item at the front/head of the queue), and peek (returns the item at the front/head of the queue).

Strangely, there is no simple Queue implementation, as described, that orders items as they were added. (There is a PriorityQueue implementation which we will cover in Section 6.9.) Thus, to make a simple queue, you use the LinkedList class using *addLast* instead of *offer*, *removeFirst* instead of *poll*, and *getFirst* instead of *peek*.

Homework 6.7

1. Explain how the basic Stack operations can be implemented in constant time.
2. Explain how the basic Queue operations can be implemented in constant time.

6.7 Sets

A Set is a collection that has no duplicate elements. There are two major implementations: HashSet (unordered) and TreeSet (ordered). The Set interface does not introduce any new methods.

6.7.2 The HashSet Class

1. A HashSet is an unordered collection of items and all operations are $O(1)$ as we will see in Chapter 20. It is important to remember that there is no guarantee on the order returned by the HashSet iterator, except that all items will be visited.

2. The Object class provides two important methods: equals and hashCode. Equality as defined in the Object class simply states that two objects are equal if their respective references point to the same object. For example, if *equals* has not be overridden in the Person class, then:

```
Person p1 = new Person( "Bob", 12 );
Person p2 = new Person( "Bob", 12 );

System.out.println( hs.add( p1 ) ); // true
System.out.println( hs.add( p2 ) ); // true
```

Further, if two objects are equal, then they must have the same hashCode (an integer value). As defined in the Object class, hashCode usually converts the memory address to an integer value. Anyway we look at it, *p1* and *p2* above are two different objects and thus are not considered duplicates. If we tried to add *p1* again, it would fail:

```
System.out.println( hs.add( p1 ) ); // false
```

3. However, many times we may want to use the idea of *logical equality* where two objects are considered *logically* the same if they have some of the same attribute values. For instance, we may want two Person objects to be considered the same whenever the *name* and *id* fields have the same value. To provide this notion of logical equality, we must override the *equals* and *hashCode* methods. We will learn more about hash codes later in the semester. For now, we can think of them as being a hint about where an object is located. For an excellent, in-depth explanation, see:

   http://java.sun.com/developer/Books/effectivejava/Chapter3.pdf

4. Logical Equality Example 1 – In this case we define a Student01 class and override equals to say that two Student01's are equal if their *name* fields have the same value. This doesn't lead to the desired result because hashCode was not overridden.

```
// equals overridden, but not hashCode
class Student01
{
    String name; int id;

    public Student01( String n, int i ) {
        name = n;
        id = i; }

    public boolean equals( Object rhs )
    {
        if( rhs == null || getClass( ) != rhs.getClass( ) ) return false;

        Student01 other = (Student01) rhs;
        return name.equals( other.name );
    }
}
```

Test Code:

```
Student01 s1 = new Student01( "Bob", 0 );
Student01 s2 = new Student01( "Bob", 2 );

Set<Student01>  hs = new HashSet<Student01>();

System.out.println( hs.add( s1 ) ); // true
System.out.println( hs.add( s2 ) ); // true
```

5. Logical Equality Example 2 – In this case we define a Student02 class and override equals as before as well as override hashCode. Now, two objects with the same *name* will not be allowed in the hashset because they will have the same hashCode.

```
// equals and hashCode overridden
class Student02
{
   ...
   public int hashCode()
   {
      return name.hashCode();
   }
}
```

Test Code:

```
      Student02 s1 = new Student02( "Bob", 0 );
      Student02 s2 = new Student02( "Bob", 2 );

      Set<Student02>  hs = new HashSet<Student02>();

      System.out.println( hs.add( s1 ) ); // true;
      System.out.println( hs.add( s2 ) ); // false;
```

6. Logical Equality Example 3 – In this case we define a Student03 class and override equals so that two students are equal if they have the same name and id. As well, we override hashCode.

```
// equals and hashCode overridden
class Student03
{
    ...

    public boolean equals( Object rhs )
    {
        if( rhs == null || getClass( ) != rhs.getClass( ) )
            return false;

        Student03 other = (Student03) rhs;
        return name.equals( other.name ) && id==other.id;
    }

    public int hashCode()
    {
       // Although this works, there are better things we can do here.
       return name.hashCode()+id;
    }
 }
```

Test Code:

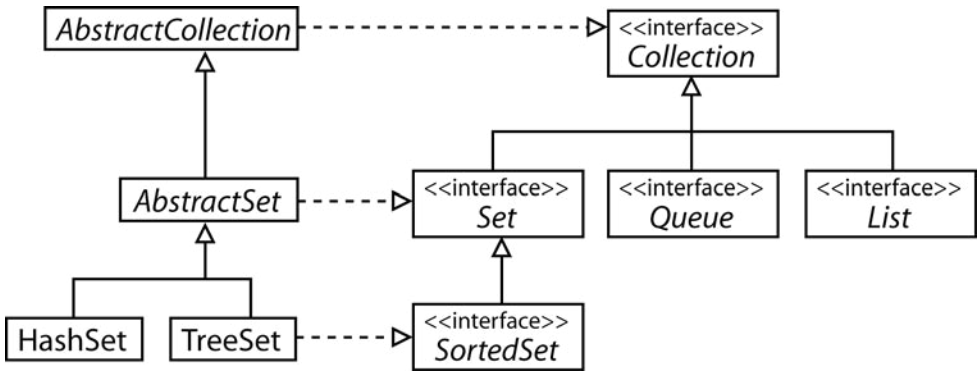
```
Student03 s1 = new Student03( "Bob", 0 );
Student03 s2 = new Student03( "Joe", 1 );
Student03 s3 = new Student03( "Bob", 2 );
Student03 s4 = new Student03( "Bob", 0 );

Set<Student03>  hs = new HashSet<Student03>();

System.out.println( hs.add( s1 ) ); // true
System.out.println( hs.add( s2 ) ); // true
System.out.println( hs.add( s3 ) ); // true
System.out.println( hs.add( s4 ) ); // false
```

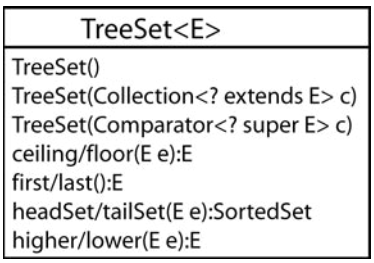
Homework 6.8

1. You need a HashSet that will store Person objects, where a duplicate is defined as one with the same *lastName, firstName,* and *id.* Write the Person class and a snippet of test code.
2. Why must equals and hashCode be overridden when using HashSet with objects?
3. What does logical equality mean and how is it enforced in the context of HashSet?

6.7.1 TreeSet

1. A TreeSet is ordered by using natural order or by specifying a comparator. As we will see in Chapter 19, contains, add, and remove can be implemented in $O(\log n)$ by using a binary search tree and its variants. The iterator for the TreeSet presents the elements in the order defined by the comparator. In addition, TreeSet has some additional methods. Note that with a TreeSet it is not necessary to override equals and hashCode. A TreeSet only uses the compareTo method or an explicit Comparator. (However, this is inconsistent with the definition of the Map interface, but it doesn't cause a problem with the order).





2. A TreeSet does not directly support a get operation. However, you can use the *ceiling* (or *floor*) method. To see this, consider a Student class with this implementation of compareTo (equals and hashCode are not needed):

```
public int compareTo( Student01 s )
{
      return name.compareTo(s.name);
}
```

First, note that logical equality is enforced through the compareTo method:

```
Student01 s1 = new Student01( "Kay", 4 );
Student01 s2 = new Student01( "Dan", 6 );
Student01 s3 = new Student01( "Bob", 9 );
Student01 s4 = new Student01( "Bob", 2 );

TreeSet<Student01>  ts = new TreeSet<Student01>();

System.out.println( ts.add( s1 ) ); // true
System.out.println( ts.add( s2 ) ); // true
System.out.println( ts.add( s3 ) ); // true
System.out.println( ts.add( s4 ) ); // false
```

Finally, we can "search" for "Bob". Note that in *s5* the *id* is unimportant as it is not used in the compareTo method.

```
Student01 s5 = new Student01( "Dan", 33 );
s5 = ts.ceiling( s5 );
System.out.println( s5 ); //  Dan 6
```

As shown below, the ceililng function returns the object that is greater than or equal to the input parameter:

```
Student01 s6 = new Student01( "Fred", 33 );

Student01 s7 = ts.ceiling( s6 );

if( s7.compareTo(s6) != 0 )
   System.out.println( "Not found, next higher: " + s7 );
   // Not found, next higher: Kay 4
else
    System.out.println( "Found: " + s7 );
```

Homework 6.9

1. Write all necessary code to declare and instantiate a TreeSet of Strings sorted smallest to largest.
2. Write all necessary code to declare and instantiate a TreeSet of Strings sorted largest to smallest.
3. Write all necessary code to declare and instantiate a TreeSet of Person objects ordered on the *name* field using:
   a. implementing the Comparable interface
   b. a custom Comparator
4. Write code to convert a LinkedList of Person objects to a TreeSet whose Person objects are ordered by the *name* field. Assume that you have a custom Comparator, *nameComparator* that defines the ordering.

6.8 Maps

1. A *Map* stores *key/value pairs*. The idea is that you are storing *value* objects in this data structure and each *value* has a unique *key* associated with it. For instance, we may have a map from social security numbers (key) to employees (value). Each employee has a unique social security number. The syntax for the Map interface in Java, is:

```
Map<KeyClass,ValueClass> map;
```

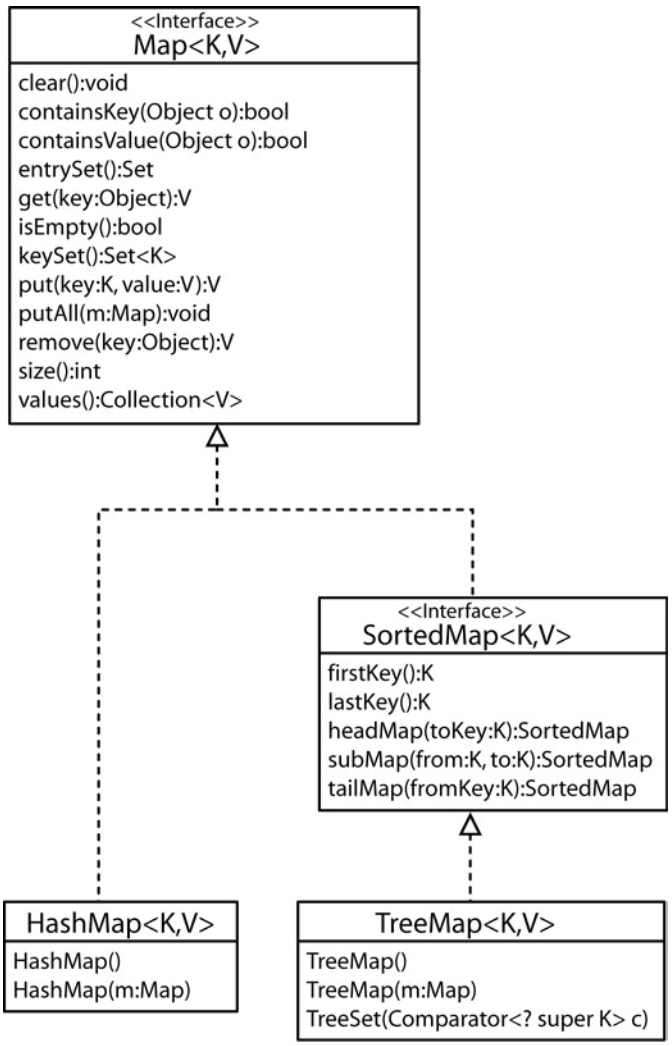
and for the example above:

```
Map<Integer,Employee> map;
```

2. Maps are not Collections, the Map interface is separate. The Map interface provides methods put, get, and remove. With any Map, you can retrieve a value with a key using the *get(key):value* method. The *keySet* method returns a Set of keys and *values* returns a Collection of the values. There are two implementations of the Map interface, HashMap and TreeMap.

   With HashMap, the keys are unordered and must be unique. As stated with the HashSet, if the keys are not types that have a natural ordering (*e.g.* Integer, Double, String, *etc.*), then equals and hashCode should be overridden.

   A TreeMap has keys that are ordered. Thus, a TreeMap requires an explicit Comparator, keys that are Comparable, or a natural ordering. TreeMap provides a number of useful methods for obtaining subsets of the map.

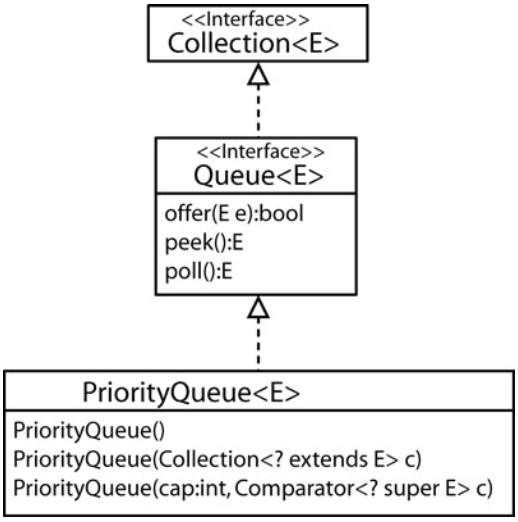


Homework 6.10

1. Suppose you have a Person class with a *name* and *id* public fields. Define a TreeMap to store Person objects using the *id* as the key.
2. Define an ArrayList of TreeMaps where the TreeMap is defined as in problem 1.
3. Suppose you have a Company class and a Employee class. Define a TreeMap that will store HashSets of Employee objects using a Company as the key.

6.9 Priority Queues

A PriorityQueue is useful in many real situations. A regular Queue is First-In, First-Out (FIFO). A PriorityQueue is different in that it orders items on their priority. For instance, with a printer queue, a print job may be added to the end of the queue (offer); however, if the print job is particularly important we might want to print that job first. The concept of *priority* is specified through a Comparator, Comparable, or natural ordering. Although a bit counter-intuitive, the PriorityQueue is ordered from smallest to largest so that the items that come off the queue first have smallest priority. Poll and peek return the item with smallest priority. Note, a PriorityQueue can have duplicate elements. Note from Sun/Oracle on Java 1.5: "Implementation note: this implementation provides O(log(n)) time for the enqueing and dequeing methods (offer, poll, remove() and add); linear time for the remove(Object) and contains(Object) methods; and constant time for the retrieval methods (peek, element, and size)."



Homework 6.11

1. Describe several situations (software context) where a queue would be a useful model component.
2. Describe how a TreeSet could be used as a priority queue. What limitations would this TreeSet have?
3. Suppose you have a PQ of Job objects, with priority, *dueDate* and a dueDateComparator. Suppose now that you want to create a new PQ of the same Job objects, with priority, *orderCost* and an *orderCostComparator*. Write the code to do this.
4. Determine the complexity of this method:

```
public static void addStuds( Student[20] arStuds,
                             Treemap<Integer,Student> tmStuds )
{
   for( Student s : arStuds ) tmStuds.put( s.SSN, s );

}
```

5. How would you create a custom PriorityNQueue class that allowed a fixed number of elements to be returned from a *peek* or *poll*. This value should be able to be set when the PriorityNQueue is created. What risks are associated with this data structure. Can you think of an inheritance based solution? Can you think of a composition based solution? What are the benefits and disadvantages to each?

   Implementing Queue interface:

   1. The PNQ could extend the PQ class and override peek, etc; or, it could use composition, PNQ has a PQ. The class of the objects in PQ could have a pointer to next item in PQ. That seems like a lot of extra work and seems to defeat the purpose of a PQ.

   2. The generic type for the contained/extended PQ could be an array of PQ. Thus, when you offer, you would provide the object inside an array.

   Breaking Queue interface:

   1. Just write it from scratch, perhaps using a binary heap, keeping track of top of heap and next to top.