The Java Collections Framework

- Data Structures
- Performance
- Collection Types

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Introduction to Collections

- Collections store object references and provide methods to add, remove and find objects
- Java provides several different types of collections so you can choose the best collection type for your application
- JSE Version 5 adds **generic** collections and Queue interfaces and classes
- JSE Version 8 adds **streams**

Since the very beginning, Java has provided collection classes, but with the advent of J2SE 1.2, Java added the Collection Framework, which greatly expanded the support for collections.

Collections are more flexible than arrays, but generally do not have the raw performance of arrays.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Characteristics of a Collection

- How fast is random access to elements?
- How fast are inserts and deletes?
- Are duplicates allowed?
- Does the collection sort elements as they are added?
- Does the collection store simple elements or key/value pairs?

Java supplies many different types of collections, and sometimes it's a bit daunting trying to figure out which one to use. Here are a set of questions you can use to help you match a collection type to your application.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Collection Performance Terminology

Term	Speed	Description	Notation
Quadratic time	slow	Time proportional to the square of the number of elements	O(n**2)
Linear time	faster	Time proportional to the number of elements	O(n)
Logarithmic time	even faster	Time proportional to the log of the number of elements	O(log n)
Constant time	fastest	Time doesn't depend on the number of elements	O(l)
Amortized constant time	approaching constant time	Time varies widely, costly operations enable subsequent faster operations	N/A

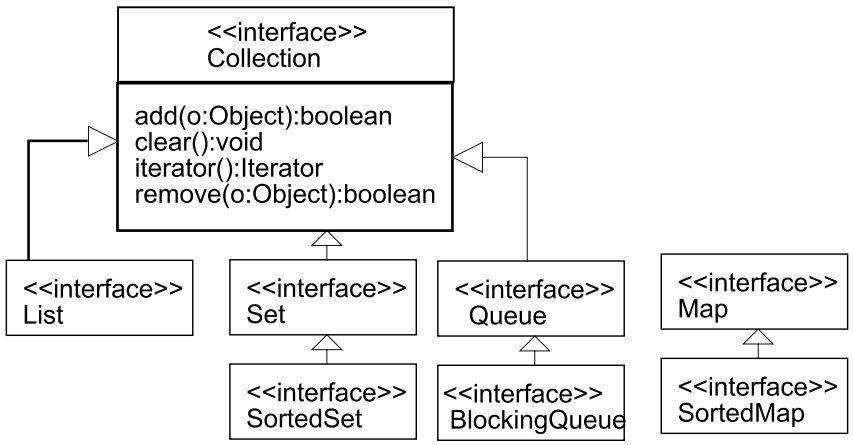
When you are evaluating the performance of operations on a particular collection type, you can use the terminology shown here. Note that "constant" time is the fastest, since it doesn't depend on the number of elements in the collection (i.e. works the same for large collections).

Amortized constant time is an average, but more geared toward common usage of the collection. An example of a operation that takes amortized constant time is appending elements to an array-based collection. Such collections normally pre-allocate space, so initially, appends are very fast. But when the allocation is used up, the collection must re-allocate a larger array and then copy elements from the initial array to the new array -- this is expensive. But since the collection allocates the new array with plenty of extra space, subsequent appends are again fast (until the free space is gone). So if you take a typical case where such a collection grows over time, the average time for appends approaches constant time.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Basic Collection Types

- The *Collection* interface defines the basic behavior for most collections
- **Lists** are sequential collections that allow duplicates
- **Sets** reject duplicates and are optionally sorted
- **Maps** are key/value collections where the key must be unique
- **Queues** have *head* and *tail* elements and are often used as a FIFO



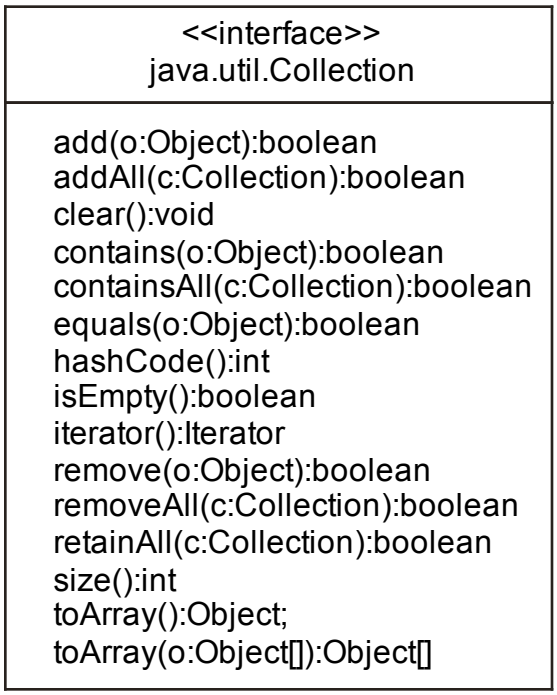
These are all "interfaces", which are like classes but cannot be instantiated. Interfaces define a set of methods that actual classes can implement -- in other words, these are not actual collections, just "prototypes" for the actual collections we will see in a moment.

The Map interface is not a subtype of Collection since maps work somewhat differently than other collections.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The Collection Interface

- The **Collection** interface defines the basic behavior of most collection classes



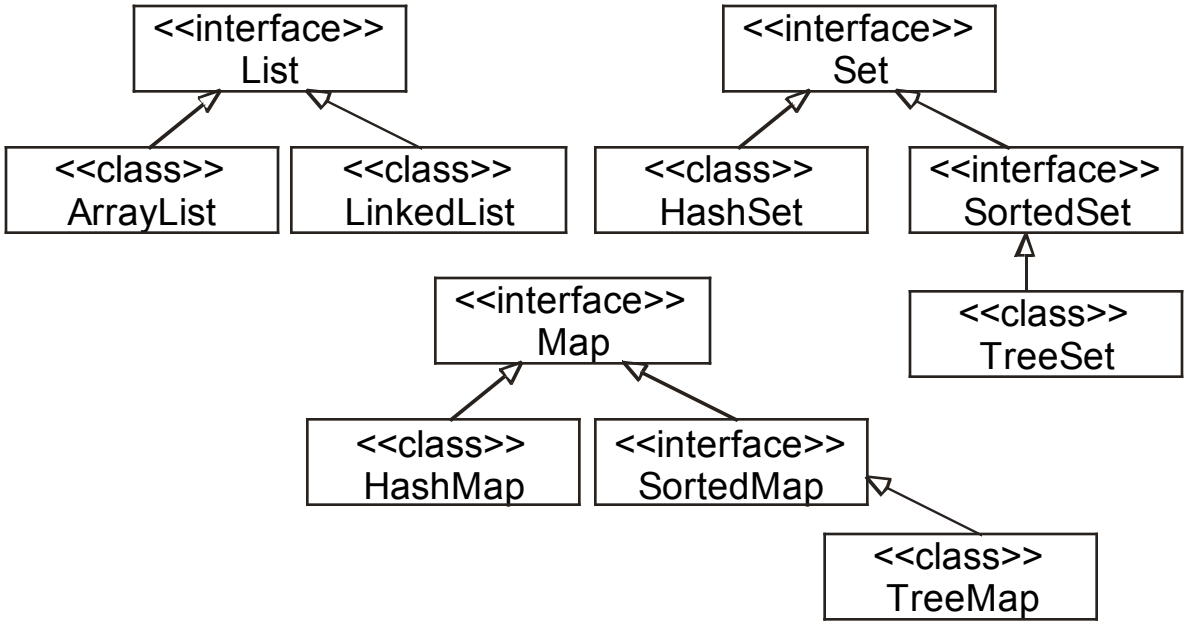
Please note that not all collections actually implement all of these methods: add(), addAll(), clear(), remove(), removeAll() and retainAll() methods are optional (these operations modify the collection, and the framework allows for read-only collections). If a collection doesn't support the operation, it can throw the UnsupportedOperationException, though this is not required.

The size() method returns the count of elements currently stored in the collection.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Concrete Collection Classes

- Concrete collection classes use well-known data structures to store elements

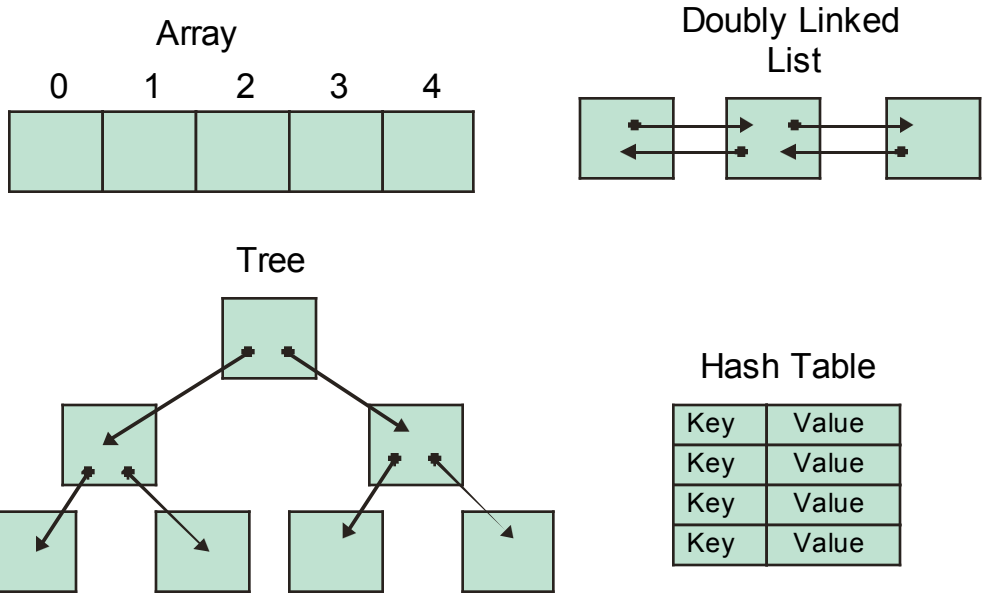


This figure shows a few of the "concrete" collection classes that Java provides; there are others. Note that each concrete class uses a different type of underlying data structure to implement the interface -- for example, an ArrayList uses an array, while a LinkedList uses a doubly-linked list.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Data Structures Overview

- Each type of data structure has different characteristics for: random access performance, insertion performance, sorted or sequential elements



2 - 8

Array-based collections have fast random access, but it's relatively slow to insert items in the middle, since the collection must move any existing elements back.

Linked lists have fast inserts, but relatively slow random access, since the collection must, on average, navigate through half of the collection to get to a given element (Java collections use doubly-linked lists).

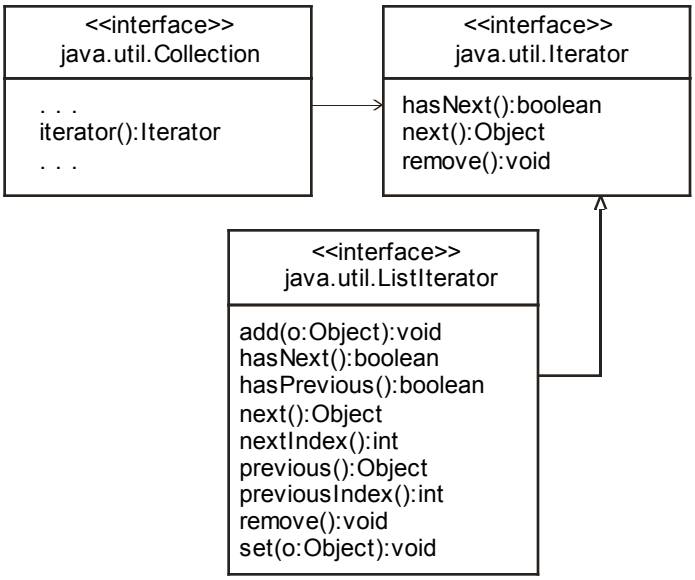
Trees provide automatic sorting and relatively fast inserts and random access.

Hash tables use a key and a value -- you can look up values using the key relatively quickly.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using Iterators

- Most collection types let you retrieve an *iterator* object that understands the collection's underlying data structure
- The iterator provides methods to let you traverse a collection in the most efficient way



Iterator classes implement the Iterator or ListIterator interfaces and provide methods to at minimum, navigate through a collection in the most efficient manner. Iterators replace the Enumerators that were originally part of Java. Note that the remove() operation is optional for iterators to actually implement.

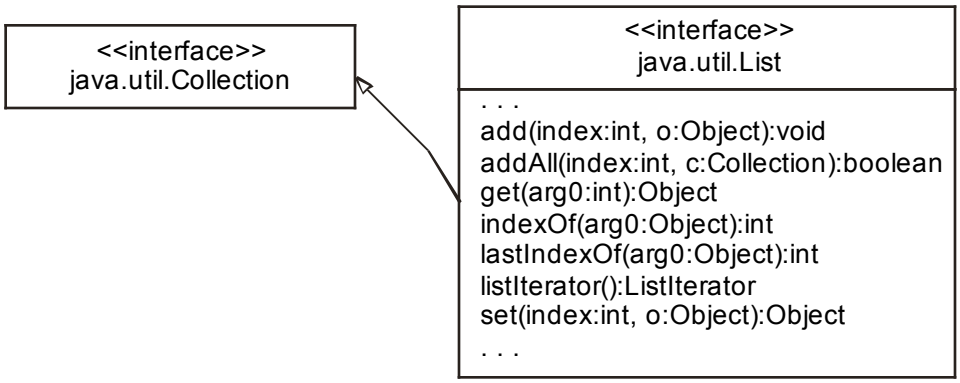
You should use iterators whenever possible even though some collections (e.g. Lists) provide other ways to traverse the collection (e.g. List provides the get() method to retrieve an element at a particular index). Iterators, in general, will provide the quickest way to traverse from one element to the next, since they know the internal structure of the collection. For example, an iterator provided by a LinkedList collection knows how to traverse the doubly-linked list.

Most collections supply so-called "fail-fast" iterators. That means that if you retrieve an iterator and then change the associated collection (e.g. delete an element), the iterator is now invalid, but will throw an exception on next use of the iterator. This behavior is not guaranteed, however, so your code should not depend on it.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

List Collections

- The **List** interface augments **Collection** to specify the basic behavior of List collections
- There are two basic data structures to implement List collections: arrays and linked lists
- All List collections are sequential, that is, they maintain the order of elements (unless you explicitly sort them)
- List collections allow duplicate elements



The main feature of Lists is that they are sequential collections in which each element has an index. The collection maintains the index of existing elements as you append elements (using the add() operation without specifying an index). If you use add() and specify an index, the collection "pushes" back existing elements to insert the new element at the specified index. Note that this operation may be costly in terms of performance, depending on the data structure used to implement the List.

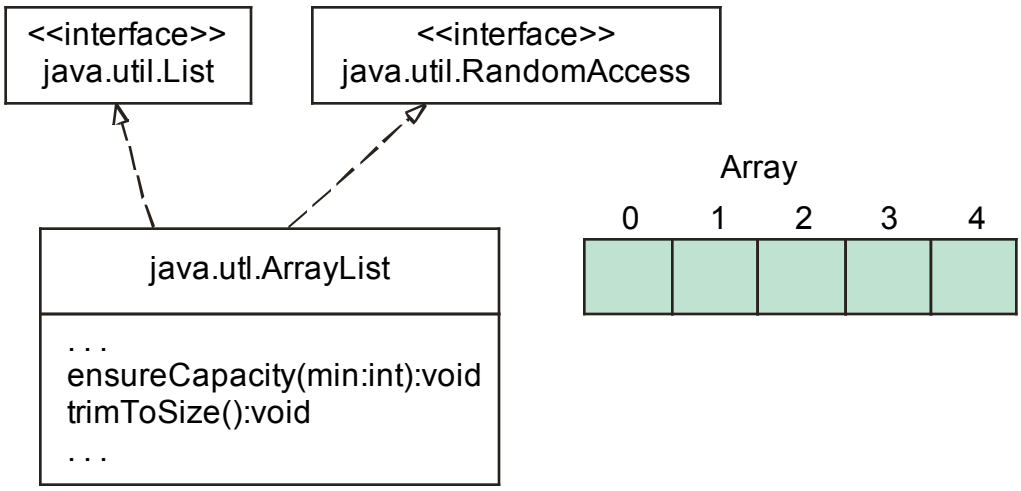
Note also that the performance of the get() method depends greatly on the data structure used to implement the collection. Some data structures (e.g. arrays) provide constant-time random access, while others (e.g. LinkedList) have linear-time random access.

In any case, it's best not to use get() as a general-purpose way to traverse the elements in a List -- iterators provide better performance in that role.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

ArrayList: Array-Based List Collection

- The **ArrayList** collection has constant-time random access
- With an **ArrayList**, it's faster (amortized constant time) to append elements, while inserts anywhere else run in linear time



As you might guess from its name, the ArrayList collection internally uses an array to implement the List interface. As such, it provides much of the performance characteristics of normal arrays (e.g. fast random access) with the ability to resize itself as you add elements.

ArrayLists have an initial capacity, which you can specify when you instantiate the ArrayList (if you use the no-argument constructor, the default capacity is 10). The ArrayList provides the trimToSize() and ensureCapacity() methods to let you indirectly manipulate the internal array.

The RandomAccess interface is a "marker" interface with no methods. It exists so that algorithms that accept a generic List reference can determine if the collection supports constant-time random access.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using the ArrayList Collection

```
 1    import java.util.*;
 2
 3    public class TestArrayList
 4    {
 5        public static void main(String[] args)
 6        {
 7            List<Integer> al =
 8               new ArrayList<Integer>(25);
 9
10            for(int i = 0; i<25; i++)
11            {
12                al.add(new Integer(i));
13            }
14
15            for(Integer anInt : a1)
16                System.out.println(anInt);
17        }
18   }
```

In Java SE Version 5 (aka Java 1.5), you can use these generic collections for which you specify a type, in this case Integer.

If you use generic collections, you can then use the "enhanced for loop", which has a simple syntax. Note that the enhanced for loop uses an Iterator behind the scenes.

Note that in JSE version 7, you can simplify the creation of the collection using so-called diamond syntax:

```
List<Integer> al = new ArrayList<>(25);
```

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Old School ArrayList Collection

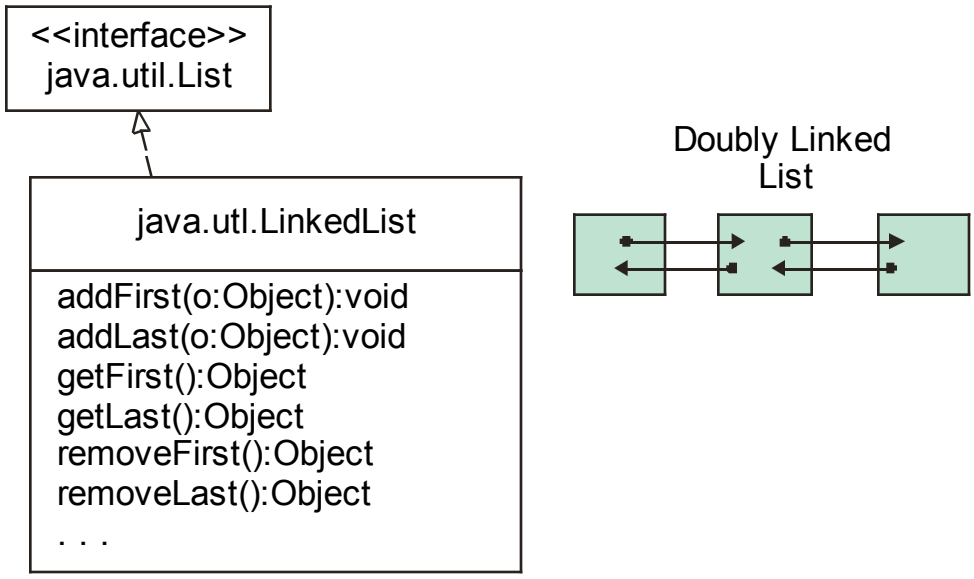
```
1    import java.util.*;
2
3    public class TestArrayList
4    {
5        public static void main(String[] args)
6        {
7            List al = new ArrayList(25);
8
9            for(int i = 0; i<25; i++)
10           {
11               al.add(new Integer(i));
12           }
13
14           Iterator iter = al.iterator();
15           while(iter.hasNext())
16               System.out.println(iter.next());
17       }
18   }
```

This code demonstrates creating an ArrayList without using generics. This was the standard approach before JSE version 5.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

LinkedList: Linked List-Based Collection

- With a **LinkedList** collection, random access takes linear/2 time
- **LinkedList** collections provide constant-time inserts anywhere in the collection **



The LinkedList class internally uses a doubly-linked list to implement the List interface. That means that elements cannot be directly addressed -- the collection must iterate to access an element at any given index. Since the implementation is doubly-linked, random accesses run in linear-time/2, since the collection will start traversing from the beginning or end of the list, whichever is closer.

The LinkedList class provides additional methods with intuitive names to add/delete elements and the beginning and end of the List. These methods are more or less aliases or wrappers around existing methods from the List interface.

** Note that performance of inserts in the middle is not as fast as it might seem -- unlike with an ArrayList, the collection doesn't need to rearrange existing elements (just update references), but the traversal time penalty mentioned above is incurred.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

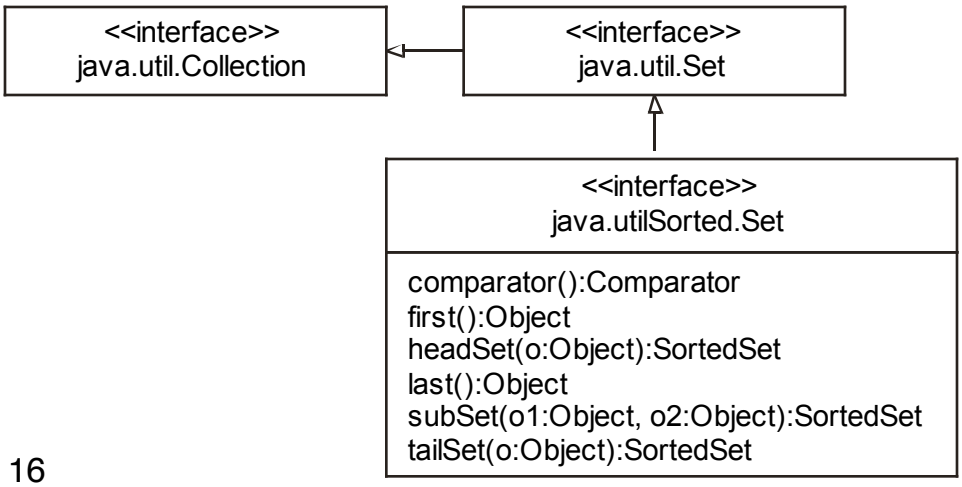
Using the LinkedList Collection

```
 1    import java.util.*;
 2
 3    public class TestLinkedList
 4    {
 5        public static void main(String[] args)
 6        {
 7            List<Integer> ll =
 8               new LinkedList<Integer>();
 9
10            for(int i = 0; i<25; i++)
11            {
12                ll.add(new Integer(i));
13            }
14
15            for (Integer anInt : ll)
16                System.out.println(anInt);
17        }
18   }
```


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Set Collections

- The **Set** interface augments **Collection** to specify the basic behavior of Set collections
- The **SortedSet** interface augments **Set** to specify the basic behavior for Sets that sort elements as they are added
- **Sets** reject duplicate elements



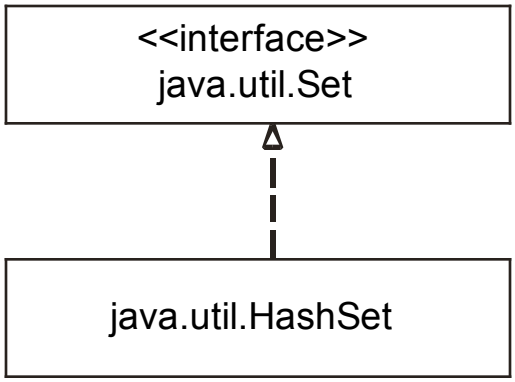
The Set interface adds no new functionality to the Collection interface, but SortedSet provides methods that let you work with positions of elements. SortedSets also support using objects that implement the Comparator interface -- Comparators help SortedSets sort objects as they are added to the collection (alternatively, the objects themselves can implement the Comparable interface).

There are two basic data structures used to implement Sets and SortedSets: hashtables and trees. We will see examples of both coming up.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

HashSet: Hash Table-based Set Collection

- A **HashSet** provides constant time performance for *add*, *remove* and *contains* operations
- **HashSet** collections use a hashtable data structure behind the scenes to actually implement the **Set** interface
- **HashSet** collections do not guarantee any ordering of elements



The HashSet class adds no new methods to those defined in Set (which themselves are the same as in Collection).

To implement the Set interface, HashSets internally use a hashtable data structure. Hashtables provide good performance for most operations in generally usage, but can have very bad performance if the objects stored in the collection do not have reasonable implementations of the hashcode() method.

When you create a HashSet, you can specify a "capacity" and a "load factor". The collection uses these values to size the internal hash table. You can read about how they affect performance in the JavaDoc, but for most programs, the default values provided by the no-argument constructor work OK.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using a HashSet

```
 1    import java.util.*;
 2
 3    public class TestHashSet
 4    {
 5        public static void main(String[] args)
 6        {
 7            String[] stuff =
 8              {"The", "quick", "brown", "fox", "The"};
 9            Set<String> set =
 10               new HashSet<String>();
 11
 12           for (int i=0; i<stuff.length; i++)
 13               set.add(stuff[i]);
 14
 15           for (String s : set)
 16               System.out.println(s);
 17       }
 18   }
```

This program instantiates a HashSet and then adds five String objects. Then the program uses an iterator to traverse the collection.

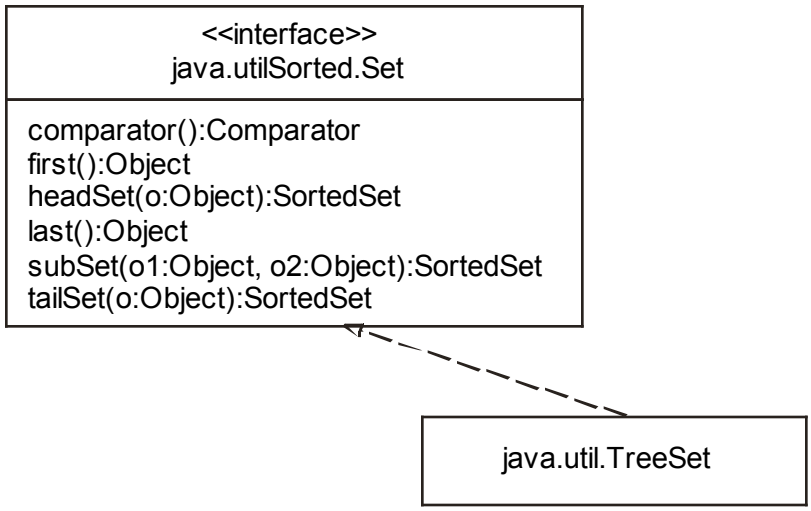
The output from this program is:

fox quick brown The But you should not depend on any particular ordering.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

TreeSet: Tree-based SortedSet Collection

- **TreeSet** collections use a tree data structure behind the scenes to actually implement the *SortedSet* interface
- **TreeSets** reject duplicates and automatically sort elements as they are added (via the *Comparable* interface)



To implement the Set interface, TreeSets use a "red-black" tree data structure, which provides logarithmic-time performance for most operations (compared to the faster constant-time performance for HashSet). However, TreeSets perform automatic sorting of elements using either the element's "natural order" (provided by the Comparable interface) or by a Comparator object that you supply when you construct the TreeSet.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using a TreeSet

```
 1    import java.util.*;
 2
 3    public class TestTreeSet
 4    {
 5        public static void main(String[] args)
 6        {
 7            String[] stuff =
 8              {"The", "quick", "brown", "fox"};
 9            Set<String> set =
10                new TreeSet<String>();
11
12            for (int i=0; i<stuff.length; i++)
13                set.add(stuff[i]);
14
15            for (String s : set)
16                System.out.println(s);
17        }
18   }
```

The output of this program is:

The

brown

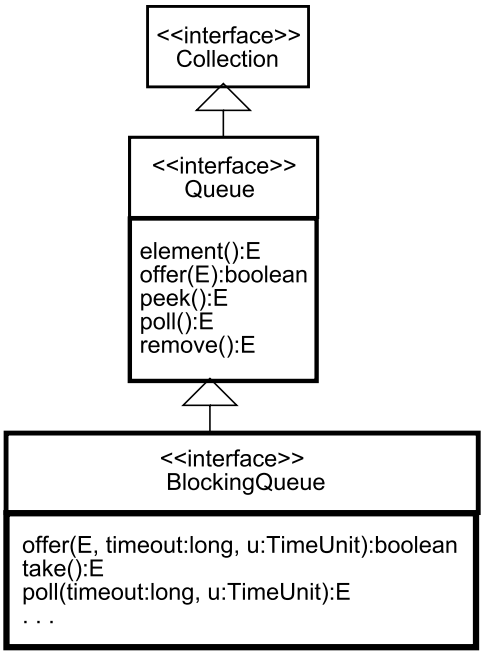
fox

quick

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Queue Collections

- The **Queue** interface augments **Collection** to specify the basic behavior of Queues
- The **BlockingQueue** interface augments **Queue** to specify the basic behavior for Queues that block on reads from an empty queue and writes to a full queue
- Commonly, we use **Queues** as first-in-first-out (FIFOs)



Queues have "head" and "tail" elements. Methods like "add()" (inherited from Collection) and "offer()" insert an element at the head of the queue, while methods like "remove()" and "take()" extract the tail element.

Blocking means to wait. In other words, if you try to add an element to a blocking queue that's full, the "offer()" call will block, or wait until there's room in the queue or until a timeout is reached.

While we often use queues for FIFOs, that's not always the case. Java also supports a "priority" queue in which the elements from head to tail are ordered by relative priority. The queue determines an element's priority using the Comparable or Comparator interfaces (covered later).

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

LinkedList: Linked List Queue Collection

- The **LinkedList** collection implements both the **Queue** and **List** interfaces, so it can play both roles

```
1    LinkedList<String> l = new LinkedList<String>();
2
3    l.add("String 1");
4    l.add("String 2");
5
6    System.out.println(l.remove());
7    System.out.println(l.remove());
8    System.out.println(l.remove());
```

```
String 1
String 2
Exception in thread "main" java.util.NoSuchElementException
```

The LinkedList class has multiple personalities: it can act either as a simple List or as a Queue.

Here we add two elements to the linked list and then attempt to retrieve them using the "remove()" method from the Queue interface. The remove() method retrieves the element at the head of the queue and pulls it from the queue.

When used as a Queue, LinkedList acts as a FIFO -- "String 1" comes out first since it was added first.

Note also that the remove() method throws an exception if you try to remove an element from an empty queue.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

ArrayBlockingQueue: Array-Based Blocking Queue

- The **ArrayBlockingQueue** collection implements the **BlockingQueue** interface
- Blocking queues hold, or "block" the caller on attempts to add to a full queue or read from an empty queue

TestBlockingQueue.java

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

```
1    package collections;
2
3    import java.util.concurrent.ArrayBlockingQueue;
4    import java.util.concurrent.TimeUnit;
5
6    public class TestBlockingQueue
7    {
8        static final int count = 3;
9        ArrayBlockingQueue<String> queue
10           = new ArrayBlockingQueue<String>(count);
11
12       public class Consumer extends Thread
13       {
14           @Override
15           public void run()
16           {
17               while(true)
18               {
19                   try
20                   {
21                       System.out.println("Consumer waiting for request");
22                       String s = queue.take();
23                       System.out.println("Consumer got a request: " + s);
24                   }
25                   catch (InterruptedException e)
26                   {
27                       e.printStackTrace();
28                   }
29               }
30           }
31       }
32       public static void main(String[] args)
33           throws Exception
34       {
35           TestBlockingQueue tbq = new TestBlockingQueue();
36           Consumer consumer = tbq.new Consumer();
37
38           for(int i = 1; i <= count+1; i++)
39           {
40               String s = "String " + i;
41               boolean added = tbq.queue.offer(
42                       s, 3000, TimeUnit.MILLISECONDS);
43               if (added == false)
44                   System.out.println(
45                       "Add timed out for String " + i);
```

```
46            }
47            consumer.start();
48        }
49    }
```

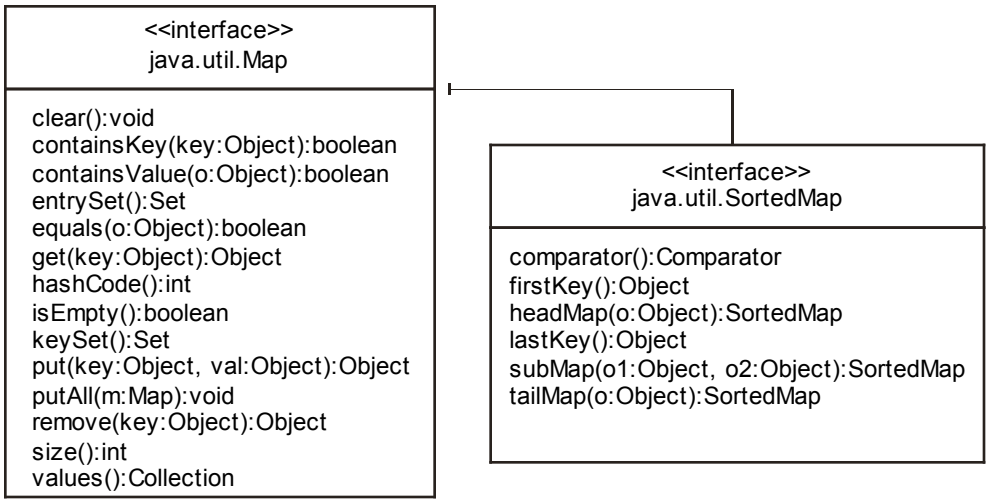
ArrayBlockingQueue Example Output

```
Add timed out for String 4
Consumer waiting for request
Consumer got a request: String 1
Consumer waiting for request
Consumer got a request: String 2
Consumer waiting for request
Consumer got a request: String 3
Consumer waiting for request
```


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Map Collections

- The **Map** interface provides methods so that you can store key/value pairs of objects where the key is unique
- The **SortedMap** interface augments **Map** to provide basic behavior for Maps that sort objects as they are added



A Map is like a phone book -- each value (person's name) is associated with a unique key (phone number). The basic operations for a Map are put() which stores a key/value pair and get(), which retrieves a value given a key.

All Map-based collections require unique keys -- if you put() a new key/value pair where the key already exists in the Map, the Map will overwrite the old value with the new value.

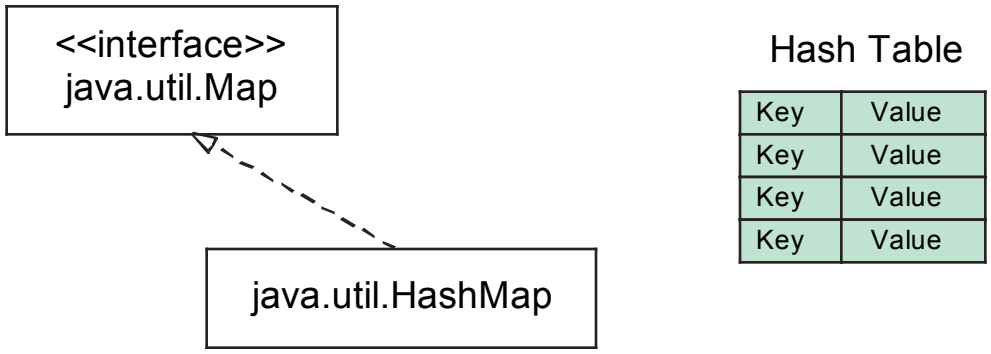
The main utility of a Map is fast searches -- unlike with Lists or Sets, retrieving a value, given a key should take place in near constant time.

SortedMaps have the additional property of sorting elements as they are added.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

HashMap: Hash Table-based Map

- A **HashMap** provides constant time performance for *get* and *put* operations **
- **HashMaps** do not guarantee any ordering of elements



HashMaps internally use a hashtable data structure to implement the Map interface. The HashMap class adds no additional methods over and above that specified by the Map interface (except for constructors).

When you create a HashMap, you can specify a "capacity" and a "load factor". The collection uses these values to size the internal hash table. You can read about how they affect performance in the JavaDoc, but for most programs, the default values provided by the no-argument constructor work OK.

** Performance depends greatly on how well the objects in the collection distribute within the hashtable, which is dependent on how well the objects' hashcode() method returns a distributed, non-colliding range of values. Constant-time performance assumes a reasonable hashcode() implementation.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using a HashMap

```
1    import java.util.*;
2
3    public class TestHashMap
4    {
5        public static void main(String[] args)
6        {
7            String[] domains =
8              {"google.com", "ibm.com",
9               "descriptor.com"};
10           String[] ips =
11             {"216.239.57.99", "129.42.17.99",
12              "209.18.79.131"};
13
14           Map<String, String> map =
15               new HashMap<String, String>();
16
17           for (int i=0; i<domains.length; i++)
18               map.put(domains[i], ips[i]);
```


This program demonstrates storing key/value pairs in a HashMap, where the keys and values are both String objects (note that they don't need to be Strings). We first instantiate the HashMap and then put() the key/value pairs in the Map.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using a HashMap, cont'd

```
19            Set<String> keys = map.keySet();
20
21            for(String aKey: keys)
22                System.out.println(map.get(aKey));
23        }
24    }
```


Continuing the listing from the last page, we then retrieve a Set collection that contains the keys. We can then iterate through the set of keys, retrieving the value for each key.

The output from the program is:

209.18.79.131

129.42.17.99

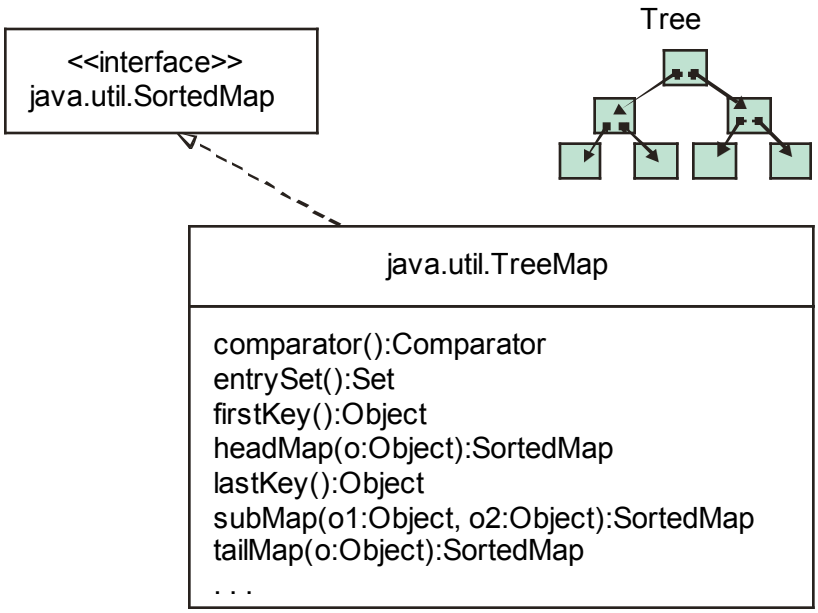
216.239.57.99

Note that we cannot assume any ordering of key/value pairs within a HashMap.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

TreeMap: Tree-based Map Collection

- A **TreeMap** provides constant time performance for *get* and *put* operations
- **TreeMaps** sort elements as they are added



TreeMaps use a red-black tree data structure to implement the SortedMap interface. Red-black trees guarantee constant-time performance for get() and put() even as the sort newly put()-ed elements.

The TreeMap sorts objects as they are added, using either the object's own Comparable interface implementation or via a Comparator object that you pass to the TreeMap's constructor.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using a TreeMap

```
1    import java.util.*;
2
3    public class TestTreeMap
4    {
5        public static void main(String[] args)
6        {
7            String[] domains =
8              {"google.com", "ibm.com",
9               "descriptor.com"};
10           String[] ips =
11             {"216.239.57.99", "129.42.17.99",
12              "209.18.79.131"};
13
14           Map<String,String> map =
15              new TreeMap<String,String>();
16
17           for (int i=0; i<domains.length; i++)
18               map.put(domains[i], ips[i]);
```

This program demonstrates storing key/value pairs in a TreeMap, where the keys and values are both String objects (note that they don't need to be Strings). We first instantiate the TreeMap and then put() the key/value pairs in the Map.

Note that the TreeMap will sort the key/value pairs in ascending key order.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using a TreeMap, cont'd

```
19          Set<String> keys = map.keySet();
20
21          for(String aKey : keys)
22          {
23              System.out.println(
24                  map.get(aKey));
25          }
26      }
27  }
```


Continuing the listing from the last page, we then retrieve a Set collection that contains the keys. We can then iterate through the set of keys, retrieving the value for each key.

The output from the program is:

209.18.79.131

216.239.57.99

129.42.17.99

This is the ascending order of the keys: "descriptor.com", "google.com", "ibm.com".

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Legacy Collections: Vector and Hashtable

- The interfaces and concrete collections covered here were added to Java in JSE 1.2
- Prior to JSE 1.2, Java defined the **Vector** and **Hashtable** -- these are still supported
- Vector is roughly equivalent to a thread-safe ArrayList and Hashtable is similar to a thread-safe HashMap
- Note that starting in JSE 5, you can use Vector and Hashtable as generic collections

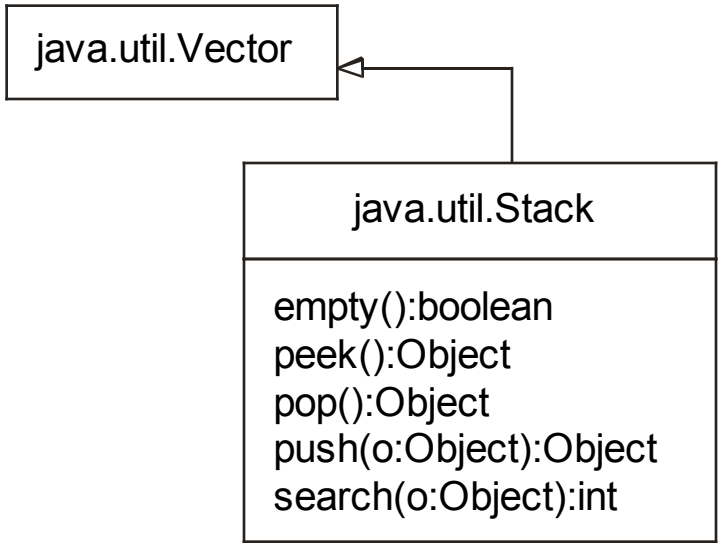
---

You may see older code that uses Vector and Hashtable -- and they are still fully supported. However, they don't really fit into the Java Collections Framework since they were created before the Framework, so new code should generally use the ArrayList and HashMap collections instead.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Stacks

- The **Stack** class is a legacy subclass of Vector that provides classic stack operations, for example, *push* and *pop*



If you prefer to use a standard part of the Java Collections Framework (or don't need thread safety), you can instead use a LinkedList collection, which provides methods to add and remove elements from the front and end of the list.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The Collections Class: Algorithms

- The **Collections** class provides several operations such as searching and sorting

java.util.Collections
binarySearch(l:List, o:Object):int copy(l1:List, l2:List):void fill(l:List, o:Object):void max(c:Collection):Object min(c:Collection):Object replaceAll(l:List, o1:Object, o2:Object):boolean reverse(l:List):void shuffle(l:List):void sort(l:List):void sort(l:List, co:Comparator):void synchronizedCollection(c:Collection):Collection synchronizedList(l:List):List synchronizedMap(m:Map):Map synchronizedSet(s:Set):Set synchronizedSortedMap(s:SortedMap):SortedMap synchronizedSortedSet(s:SortedSet):SortedSet . . .

This page shows a subset of the methods available on the Collections class. Refer to the JavaDoc for a complete list.

Many of these methods are static methods, so you don't need to instantiate a Collections object before using them.

Note: The name of this class is Collections, plural. Don't confuse it with the Collection interface (singular) that we covered earlier.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Objects Compatible with Collections

- When you write a class that will reside in a collection, you should provide features so the collection can interrogate the object

```
1    import java.util.HashSet;
2    import java.util.Iterator;
3    import java.util.TreeSet;
4
5    public class Student implements Comparable<Student>
6    {
7        private int studentID;
8        private String name;
9        private double gpa;
```


Here we start a listing of a Student class that works in pretty much any collection type. Note that the class implements the Comparable interface, which means that we can control the "natural order" if objects of this type are stored in a sorted collection. We will see the implementation of the compareTo() method from that interface in a moment.

The Student class defines three fields: studentID, name and grade-point average. We could conceivably sort on any of those, but in this example we will sort on the GPA.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Compatible Object, cont'd

```
10      public int compareTo(Student s)
11      {
12          int retVal = 0;
13
14          // sort in ascending GPA order
15          if (this.gpa < s.gpa)
16              retVal = -1;
17          else if (this.gpa > s.gpa)
18              retVal = 1;
19          System.out.println("compareTo returned: " +
20                  retVal);
21          return retVal;
22      }
```


Continuing the listing, here we see our implementation of the compareTo() method, which sorting collections call as objects of this type are added. Essentially, the collection asks the new object to compare itself to each of the existing objects. The compareTo() method returns three possible types of results: (1) A negative integer if this object is "less than" the other object, (2) zero if the objects are the same and (3) a positive integer if this object is "greater than" the other object.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Compatible Object, cont'd

```
29      public boolean equals(Object obj)
30      {
31          boolean retVal = false;
32
33          if (obj == this)
34          {
35              retVal = true;
36          }
37          else if ((obj instanceof Student))
38          {
39          Student other = (Student)obj;
40          if ( this.gpa == other.gpa )
41              retVal = true;
42          }
43          System.out.println("equals returned: " +
44              retVal);
45          return retVal;
46      }
```


Continuing the listing, here we show the implementation of equals(), which allows the outside world to see if this object is semantically the same as another. Implementing this method is not strictly required (at least as far as collections are concerned) since the compareTo() method provides that functionality. However, it's a good practice to also implement equals() and in fact it's almost required since our class implements the hashcode() method as shown coming up (implementing hashcode() and equals() as a pair is a best practice for Java classes).

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Compatible Object, cont'd

```
47      public int hashCode()
48      {
49          int result = 17;
50          long l = Double.doubleToLongBits(gpa);
51          int i = (int)(l ^(l >>>32));
52          result = 37*result + i;
53          System.out.println("id: " + studentID +
54              " hashcode returned: " + result);
55          return result;
56      }
57
58      public String toString()
59      {
60          StringBuffer sb = new StringBuffer();
61          sb.append ("ID: " + studentID);
62          sb.append (" Name: " + name);
63          sb.append (" GPA: " + gpa);
64          return sb.toString();
65      }
```


Continuing the listing, here we show implementing the hashcode() method, which is required if objects of this class will be store in any collection that internally uses a hashtable (e.g. HashSet). A good implementation of hashcode() should return values based on the object's fields that are used in equality testing-- the hashcode should be unique as possible and should be distributed across a wide range of values. The "recipe" shown here for calculating a hashcode based on the GPA is adapted from "Effective Java Programming" by Joshua Bloch.

We then implement the toString() method to return a snapshot of the object's data, complete with labels. This is not required for collections, but it never hurts.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Compatible Object, cont'd

```
66        public Student()
67        {
68        }
69
70        public Student(int studentID,
71            String name, double gpa)
72        {
73            this.studentID = studentID;
74            this.name = name;
75            this.gpa = gpa;
76        }
```


Continuing the listing, here we show the object's constructors.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Compatible Object, cont'd

```
77      public static void main(String[] args)
78      {
79          Set<Student> ts = new TreeSet<Student>();
80          System.out.println(
81              "Adding Students to a TreeSet");
82          ts.add(new Student(12,  "Sam", 3.44));
83          ts.add(new Student(3,   "Sue", 3.55));
84          ts.add(new Student(456, "Tim", 2.49));
85
86          System.out.println("Iterating a TreeSet");
87
88          for(Student s : ts )
89          {
90              System.out.println(s);
91          }
92          System.out.println();
```


Continuing the listing, here we show the beginning of a main program that tests the Student class by storing Student objects in a couple of different kinds of collections.

We first instantiate a TreeSet -- remember that TreeSets reject duplicates and sort objects as they are added. We then add three Students. Since we used TreeSet's no-argument constructor, the Student objects are sorted by their "natural order", which means using the compareTo() method, which sorts in ascending GPA order.

Then we retrieve an iterator and display the objects, hopefully in sorted order. We will see the results of running this program in a moment.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Compatible Object, cont'd

```
93            Set<Student> hs = new HashSet<Student>();
94            System.out.println(
95                 "Adding Students to a HashSet");
96            hs.add(new Student(12,  "Sam", 3.44));
97            hs.add(new Student(3,   "Sue", 3.55));
98            hs.add(new Student(456, "Tim", 2.49));
99
100           System.out.println("Iterating a HashSet");
101           for(Student s : hs)
102           {
103               System.out.println(s);
104           }
105       }
106   }
```


Continuing the listing, here we show instantiating another collection, this time a HashSet. HashSets reject duplicates, but do not impose any sorted ordering.

We then add three Student objects and then iterate as before. The results are on the next page.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Compatible Object, cont'd

Output of Program:

```
Adding Students to a TreeSet
compareTo returned: 1
compareTo returned: -1
Iterating a TreeSet
ID: 456 Name: Tim GPA: 2.49
ID: 12 Name: Sam GPA: 3.44
ID: 3 Name: Sue GPA: 3.55

Adding Students to a HashSet
id: 12 hashcode returned: -128290544
id: 3 hashcode returned: 644481653
id: 456 hashcode returned: 1589361886
Iterating a HashSet
ID: 456 Name: Tim GPA: 2.49
ID: 3 Name: Sue GPA: 3.55
ID: 12 Name: Sam GPA: 3.44
```


Here we show the results of running the program. Note that as we add objects to the TreeSet, the collection calls the compareTo() method so it can sort the new entry in relation to the existing entries. Then note that when we iterate, the objects are sorted in ascending GPA order, which is what you would expect, given the Student's implementation of compareTo().

The results from using a HashSet are different -- since that collection doesn't sort, it doesn't call the compareTo() method as objects are added. Instead, since it internally uses a hashtable, the collection calls each newly added object's hashcode() method, which hopefully returns a fairly unique, widely distributed value. Then when we iterate, note that the objects are not sorted in any particular order.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Collections and Thread Safety

- Legacy collections (e.g. Vector) are thread-safe by default; for others, you can "wrap" them with synchronized versions

```
1    List<String> list1 =
2     Collections.synchronizedList(new ArrayList<String>());
3
4    Set<String>  set1 =
5     Collections.synchronizedSet(new HashSet<String>());
6
7    Map<String, String> map1 =
8     Collections.synchronizedMap(
9        new HashMap<String, String>());
```


The Collections class is in the java.util package.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Collections and Thread Safety, cont'd

- Iterating a collection in a multithreaded environment can result in odd results or the **ConcurrentModificationException** if another thread alters the collection during iteration
- A common strategy is to clone the collection in a synchronized block before iteration

```
1    List<String> list1 = Collections.synchronizedList(
2      new ArrayList<String>());
3    . . .
4    List<String> copy = null;
5
6    synchronized(list1)  {
7        copy = new ArrayList<String>(list1);
8    }
9
10   for(String s : copy)
11     System.out.println(s);
```

Most Java collection iterators are "fail-fast" - the collection includes code that tries to determine if there's been a modification during iteration. For example, according to the JavaDoc for Vector:

```
Note that the fail-fast behavior of an iterator cannot be guaranteed as it is,
generally speaking, impossible to make any hard guarantees in the presence
of unsynchronized concurrent modification. Fail-fast iterators throw
ConcurrentModificationException on a best-effort basis. Therefore, it would
be wrong to write a program that depended on this exception for its correctness:
the fail-fast behavior of iterators should be used only to detect bugs.
```

Note that cloning collections may cause performance issues if they are large collections.

Also note that when we synchronize, we specify the source list as the monitor object - that ensures that we are locking on the same object that the collection itself uses in methods such as add() and delete().

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Java 5 Concurrent Collections

- In the case where you plan to often read collections from multiple threads, but don't often modify the collection, cloning or synchronizing is expensive
- Java 5 adds **concurrent collections** for that purpose

```
1    Map<String, String> map1 =
2      new ConcurrentHashMap<String, String>();
3
4    . . .
5
6    Set<String> keys = map1.keySet();
7    for(String key : keys)
8     System.out.println("key: " + key + " val: " +
9        map1.get(key));
```


Though ConcurrentHashMap is thread-safe, it doesn't use an locking - instead it supports concurrent access for an arbitrary number of readers and a configurable number of updaters (via a constructor parameter).

The iterator returned here does not perform any locking and thus is very fast. However, the iteration reflects the contents of the map when the iterator was created and may not include concurrent updates, e.g put() or remove(). Note that this iterator is thus considered to be "weakly consistent" and never throws ConcurrentModificationException.

Java 5 also provides CopyOnWriteArrayList and CopyOnWriteArraySet which are similar in concept to ConcurrentHashMap in that they perform well if you have a lot of reading threads and few modifications.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Review Questions

- Describe the different data structures that are used to implement collections
- What is the difference between a TreeSet and a Hashset?

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Chapter Summary

In this chapter, you learned:

- About the different types of collections in the Java Collection Framework: Lists, Maps and Sets
- About the data structures used to implement those interfaces and the various performance characteristics of the implementations

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum