6.0 Introduction

A collection is an object that represents a group of objects (a group of strings, group of beans). Collections are also called as Containers since they can contain group of mixed type data. Normally the data in a collection is related to each other even if it is of mixed data types.

A collection object (collections are also classes so need to create its object to use it) can contain or store only single type of data (ex: strings) or different/mixed types of data elements. Collections can contain objects that are of type *Object*, which means that any object that is a sub-type of the *Object* class can be stored in a collection object. It is also possible to store objects of other collections in a collection.

Collection Framework related interfaces and classes are defined under the *java.util* package.

6.0.1 Storing data in Collections

The collections can contain only objects, therefore primitive data elements like int, double are converted into their class representations (Integer, Double) before they are stored into the collection. The compiler achieves this with autoboxing (converting primitive data types into their class representations and vice-versa) and eliminates the need to explicitly convert primitive data elements into their corresponding reference data types.

Thus you can add an int value into a collection and it will be converted into Integer due to autoboxing and stored in the collection. Similarly when you retrieve the element it will converted back into int and returned.

```
List<Integer> l = new ArrayList<Integer>();
l.add(5);
l.add(new Integer(7));
```

However when declaring a collection reference to store primitive data, you can only use the corresponding wrapper class as the type parameter.

```
List<Integer> l = new ArrayList<Integer>();  // correct
List<int> l = new ArrayList<int>(); // compile-time error
```

As stated earlier it is possible to store either single type of data or mixed type of data in a collection (collections can contain anything that is of sub-type *Object*). By using Java generics it is possible to designate a collection to either store only one type of data or mixed type of data.

```
List<String> al = new ArrayList<String>(); // can contain only String type data
```

6.0.3 Why Collections?

The primary advantages of collections framework can be listed as:

- Reduces programming effort by providing useful ready-made data structures(ex: ArrayList) and algorithms(ex: sorting, searching) so that the developers do not have to re-invent the wheel.
- Increases performance by providing high-performance implementations of useful data structures and algorithms. Data structures like HashMap, ArrayList are already performance optimized.

- Provides interoperability between unrelated APIs by establishing a common language to pass collections back and forth. Collections are understood by different API since collection framework is part of the core java API.
- Reduces the effort required to learn APIs by eliminating the need to learn multiple ad hoc collection APIs.
- Reduces the effort required to design and implement APIs by eliminating the need to produce ad hoc collections APIs. There is no need to write custom collection type data structures and the associated utility methods to manipulate them since most commonly required methods are already available in the collections framework.
- Promotes software reuse by providing a standard interface for collections and algorithms to manipulate them. The already available utility methods (in Collections class) can be used to perform some standard operations like searching, sorting etc. without having to write such methods again.

6.0.4 Collections vs Arrays

Both Arrays and Collections can be used to store multiple data elements or objects. Arrays are simple data structures that can store only one type of data. They are of fixed length and store data linearly. Which means that arrays cannot grow dynamically and their size is fixed (predefined).

```
// declares an array of size 4 which can store a maximum of 4 strings
String[] str= new String[4] ;

// declares an array of size 4 and initializes with 4 string literals
String[] str = {“anand”, “ramana”,”gowtham”, “subbarao”} ;
```

The elements of an array can be obtained using the index of the element (ex: array[n]). When adding or deleting elements from an array additional handling needs to be done by the programmer to fill the voids created due to removing elements from array, since arrays cannot handle such tasks on their own.

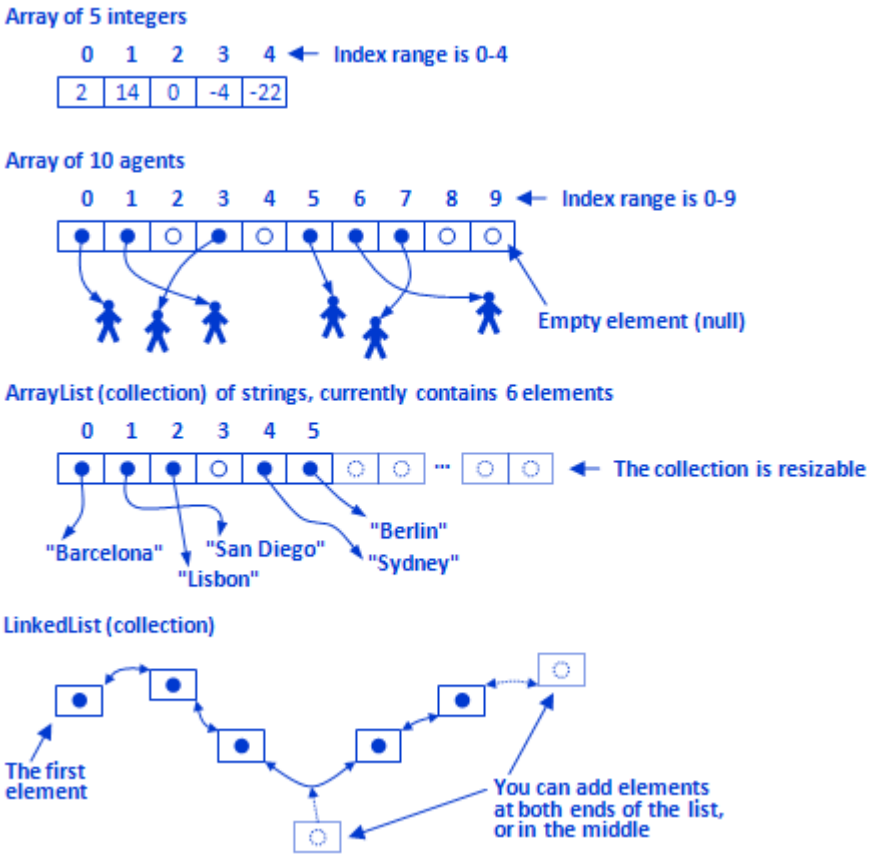
Collections unlike arrays can store single or multiple types of data in the same collection. Collections grow dynamically meaning, as you keep adding data elements to a collection the size of the collection keeps increasing. When declaring a collection object it is not required to mention its size. Therefore it is not required to know the size of the data beforehand while using collections to store such data, which is very convenient when you do not know the size of the data. This makes programming much easier. On the other hand array size must be predefined which makes it nearly impossible to work with under situations where the exact size of data is not known at the time of writing the program.

```
// declares an ArrayList object
ArrayList al = new ArrayList() ;
```

When manipulating a collection it is not required to do any additional processing like re-arranging the data elements, as such operations are automatically handled by the collections, reducing the burden on the programmer.

In-general collections are easier to work with, with many useful algorithms and utility methods already available for working with collections in the java.util.Collections class. Similarly utility methods are defined for array objects in the Arrays class. Both the Collections and Arrays classes are part of the Collections Framework.

Illustration of how Arrays and Collections grow/wane when adding data.



6.1 Collections Framework

Let us discuss the Java Collections framework. Collections framework contains the following:

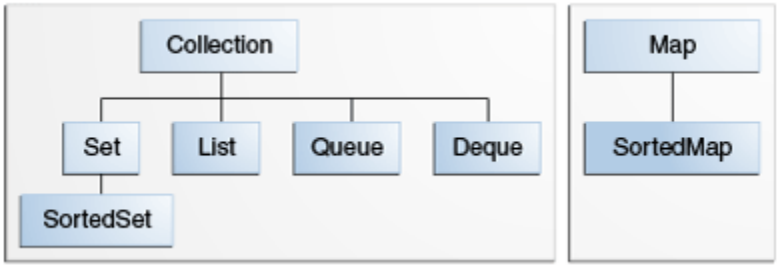
- Interfaces: These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy. (Ex: Collection, List, Set, Map)
- Implementations: These are the concrete implementations of the collection interfaces . In essence, they are reusable data structures. (ArrayList, HashSet, HashMap)
- Algorithms: These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be polymorphic: that is, the same method can be used on

many different implementations of the appropriate collection interface. In essence, algorithms are reusable functionality.

Apart from the Java Collections Framework, the best-known examples of collections frameworks are the C++ Standard Template Library (STL) and Smalltalk's collection hierarchy.

6.1.1 Interfaces

The *core collection interfaces* encapsulate different types of collections, which are shown in the figure below. Core collection interfaces are the foundation of the Java Collections Framework. As you can see in the following figure, the core collection interfaces form a hierarchy.



Note that the hierarchy consists of two distinct trees — a Map is not a true Collection. It does not derive from the core collections interface 'Collection'.

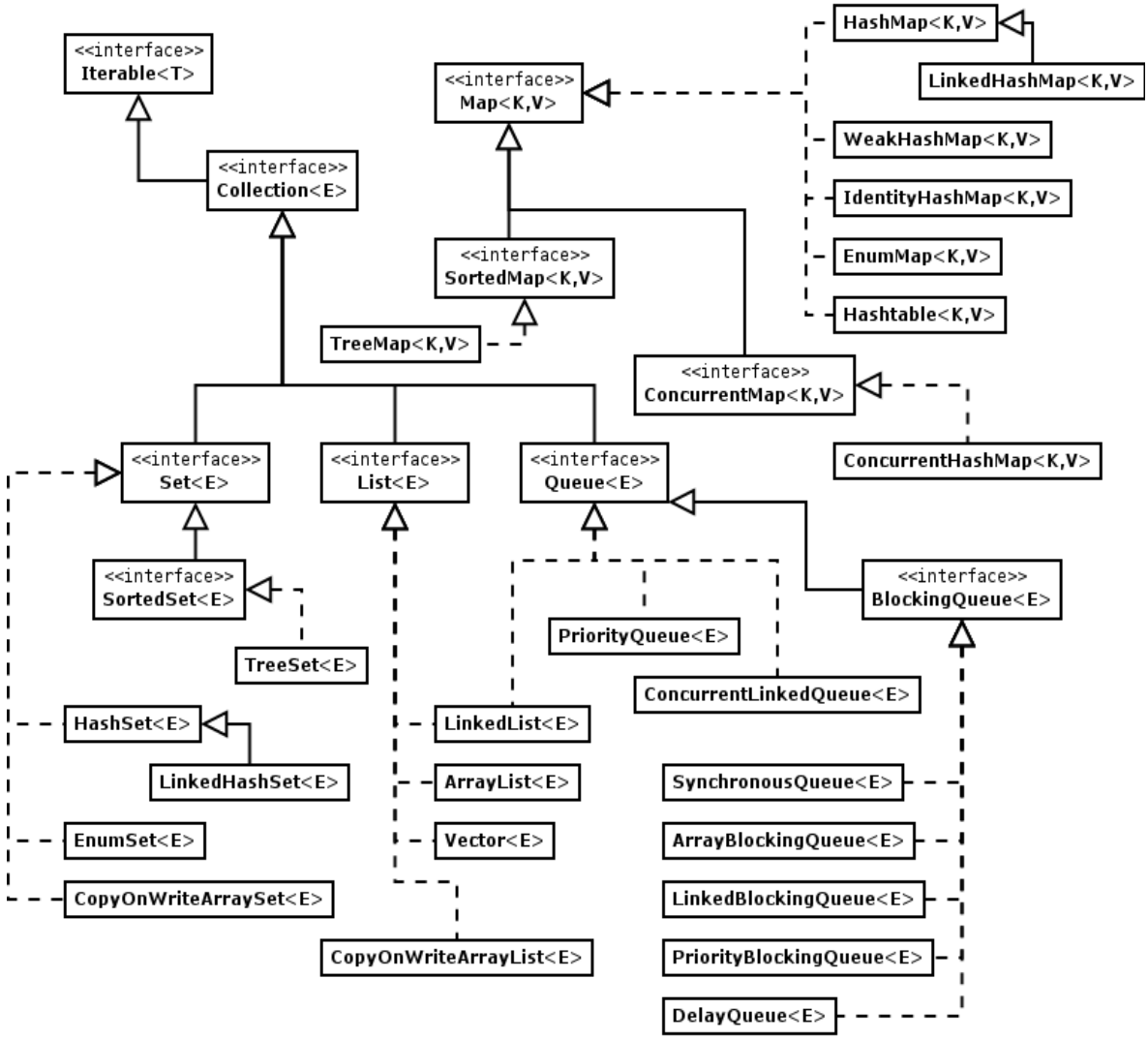
Collections are Generic

Note that all the core collection interfaces are generic. For example, this is the declaration of the Collection interface.

public interface Collection<E>...

The <E> syntax tells you that the interface is generic. Meaning at the time of declaring a collection instance you can and must specify the type(s) of object(s) it can (or cannot) contain. Specifying the type allows the compiler to verify (at compile-time) that the type of object you add into the collection is correct, thus reducing errors at runtime. Generics were introduced starting Java 1.5.

The illustration below shows the Collections framework and its members in more detail.



The following list describes the core collection interfaces:

- Collection — the root of the collection hierarchy. A collection represents a group of objects known as its *elements*. The Collection interface is the root interface all collections implement and is used to pass collections around and to manipulate them when maximum generality is desired (You should instantiate an object and assign to its super-type class/interface reference variable if you do not know what sub-type you may be instantiating later in the code so that you dont have to worry about conversion or casting exceptions). Some types of collections allow duplicate elements, and others do not. Some are ordered and others are unordered. The Java platform doesn't provide any direct implementations of this interface but provides implementations of more specific sub-interfaces, such as Set and List (which extend Collection interface).

- Set — a collection that cannot contain duplicate elements. This interface models the mathematical set abstraction and is used to represent sets, such as the cards comprising a poker hand, the courses making up a student's schedule, or the processes running on a machine.
- List — an ordered collection (sometimes called a *sequence*). Lists can contain duplicate elements. The user of a List generally has precise control over where in the list each element is inserted and can access elements by their integer index (position). If you've used Vector, you're familiar with the general flavor of List.
- Queue — a collection used to hold multiple elements prior to processing. Besides basic Collection operations, a Queue provides additional insertion, extraction, and inspection operations.

- Map — an object that maps keys to values. A Map cannot contain duplicate keys; each key can map to at most one value.

The last two core collection interfaces are merely sorted versions of Set and Map:

- SortedSet — a Set that maintains its elements in ascending order. Several additional operations are provided to take advantage of the ordering. Sorted sets are used for naturally ordered sets, such as word lists and membership rolls.
- SortedMap — a Map that maintains its mappings in ascending key order. This is the Map analog of SortedSet. Sorted maps are used for naturally ordered collections of key/value pairs, such as dictionaries and telephone directories.

A Collection represents a group of objects known as its elements. The Collection interface is used to pass around collections of objects where maximum generality is desired. For example, by convention all general-purpose collection implementations have a constructor that takes a Collection argument. This constructor, known as a *conversion constructor*, initializes the new collection to contain all of the elements in the specified collection, whatever the given collection's subinterface or implementation type. In other words, it allows you to *convert* the collection's type.

Suppose, for example, that you have a Collection<String> c, which may be a List, a Set, or another kind of Collection. This idiom creates a new ArrayList (an implementation of the List interface), initially containing all the elements in c.

```
List<String> list = new ArrayList<String>(c);
Or — if you are using JDK 7 or later — you can use the diamond operator:
List<String> list = new ArrayList<>(c);
```

The Collection interface contains methods that perform basic operations, such as int size(), boolean isEmpty(), boolean contains(Object element), boolean add(E element), boolean remove(Object element), and Iterator<E> iterator().

It also contains methods that operate on entire collections, such as boolean containsAll(Collection<?> c), boolean addAll(Collection<? extends E> c), boolean removeAll(Collection<?> c), boolean retainAll(Collection<?> c), and void clear().

Additional methods for array operations (such as Object[] toArray() and <T> T[] toArray(T[] a) exist as well.

The Collection interface does about what you'd expect given that a Collection represents a group of objects. It has methods that tell you how many elements are in the collection (size, isEmpty), methods that check whether a given object is in the collection (contains), methods that add and remove an element from the collection (add, remove), and methods that provide an iterator over the collection (iterator).

The add method is defined generally enough so that it makes sense for collections that allow duplicates as well as those that don't. It guarantees that the Collection will contain the specified element after the call completes, and returns true if the Collection changes as a result of the call. Similarly, the remove method is designed to remove a single instance of the specified element from the Collection, assuming that it contains the element to start with, and to return true if the Collection was modified as a result.

6.1.1.1 Collection Interface Bulk Operations

Bulk operations perform an operation on an entire Collection. You could implement these shorthand operations using the basic operations, though in most cases such implementations would be less efficient. The following are the bulk operations:

- containsAll — returns true if the target Collection contains all of the elements in the specified Collection.
- addAll — adds all of the elements in the specified Collection to the target Collection.
- removeAll — removes from the target Collection all of its elements that are also contained in the specified Collection.
- retainAll — removes from the target Collection all its elements that are *not* also contained in the specified Collection. That is, it retains only those elements in the target Collection that are also contained in the specified Collection.
- clear — removes all elements from the Collection.

The addAll(), removeAll(), and retainAll() methods all return true if the target Collection was modified in the process of executing the operation.

As a simple example of the power of bulk operations, consider the following idiom to remove *all* instances of a specified element, e, from a Collection, c.
c.removeAll(Collections.singleton(e));

More specifically, suppose you want to remove all of the null elements from a Collection.
c.removeAll(Collections.singleton(null));

This idiom uses Collections.singleton, which is a static factory method that returns an immutable Set containing only the specified element.

6.1.1.2 Collection Interface Array Operations

The toArray methods are provided as a bridge between collections and older APIs that expect arrays on input. The array operations allow the contents of a Collection to be translated into an array. The simple form with no arguments creates a new array of Object. The more complex form allows the caller to provide an array or to choose the runtime type of the output array.

For example, suppose that c is a Collection. The following snippet dumps the contents of c into a newly allocated array of Object whose length is identical to the number of elements in c.

```
Object[] a = c.toArray();
```

Suppose that c is known to contain only strings (perhaps because c is of type Collection<String>). The following snippet dumps the contents of c into a newly allocated array of String whose length is identical to the number of elements in c.

```
String[] a = c.toArray(new String[0]);
```

6.1.2 The Set Interface

A Set is a Collection that cannot contain duplicate elements. It models the mathematical set abstraction. The Set interface contains *only* methods inherited from Collection and adds the restriction that duplicate elements are prohibited. Set also adds a stronger contract on the behavior of the equals and hashCode operations, allowing Set instances to be compared meaningfully even if their implementation types differ. Two Set instances are equal if they contain the same elements.

The Java platform contains three general-purpose Set implementations: HashSet, TreeSet, and LinkedHashSet. HashSet, which stores its elements in a hash table, is the best-performing implementation; however it makes no guarantees concerning the order of iteration. TreeSet, which stores its elements in a red-black tree, orders its elements based on their values; it is substantially slower than HashSet. LinkedHashSet, which is implemented as a hash table with a linked list running through it, orders its elements based on the order in which they were inserted into the set (insertion-order). LinkedHashSet spares its clients from the unspecified, generally chaotic ordering provided by HashSet at a cost that is only slightly higher.

Here's a simple but useful Set idiom. Suppose you have a Collection, c, and you want to create another Collection containing the same elements but with all duplicates eliminated. The following one-liner does the trick.

```
Collection<Type> noDups = new HashSet<Type>(c);
```

It works by creating a Set (which, by definition, cannot contain duplicates), initially containing all the elements in c. It uses the standard conversion constructor defined in the the Collection Interface.

And the following is a minor variant of the first idiom that preserves the order of the original collection while removing duplicate elements:

```
Collection<Type> noDups = new LinkedHashSet<Type>(c);
```

The following is a generic method that encapsulates the preceding idiom, returning a Set of the same generic type as the one passed.

```
public static <E> Set<E> removeDups(Collection<E> c) {
   return new LinkedHashSet<E>(c);
}
```

6.1.2.1 Set Interface Basic Operations

The size operation returns the number of elements in the Set (its *cardinality*). The isEmpty method does exactly what you think it would. The add method adds the specified element to the Set if it is not already present and returns a boolean indicating whether the element was added. Similarly, the remove method removes the specified element from the Set if it is present and returns a boolean indicating whether the element was present. The iterator method returns an Iterator over the Set.

The following program prints out all distinct words in its argument list. It uses the for-each construct.

```
import java.util.*;

public class FindDups {
   public static void main(String[] args) {
      Set<String> s = new HashSet<String>();
      for (String a : args)
            s.add(a);
            System.out.println(s.size() + " distinct words: " + s);
   }
}
```

Now run the program.

```
java FindDups i came i saw i left
```

The following output is produced:

```
4 distinct words: [left, came, saw, i]
```

Note that the code always refers to the Collection by its interface type (Set) rather than by its implementation type. This is a *strongly* recommended programming practice because it gives you the flexibility to change implementations merely by changing the constructor. If either of the variables used to store a collection or the parameters used to pass it around are declared to be of the Collection's implementation type rather than its interface type, *all* such variables and parameters must be changed in order to change its implementation type. Furthermore, there's no guarantee that the resulting program will work. If the program uses any nonstandard operations present in the original implementation type but not in the new

one, the program will fail. Referring to collections only by their interface prevents you from using any nonstandard operations.

The implementation type of the Set in the preceding example is HashSet, which makes no guarantees as to the order of the elements in the Set. If you want the program to print the word list in alphabetical order, merely change the Set's implementation type from HashSet to TreeSet. Making this trivial one-line change causes the command line in the previous example to generate the following output.

```
java FindDups i came i saw i left

4 distinct words: [came, i, left, saw]
```

6.1.2.2 Set Interface Bulk Operations

Bulk operations are particularly well suited to Sets; when applied, they perform standard set-algebraic operations. Suppose s1 and s2 are sets. Here's what bulk operations do:

- s1.containsAll(s2) — returns true if s2 is a **subset** of s1. (s2 is a subset of s1 if set s1 contains all of the elements in s2.)
- s1.addAll(s2) — transforms s1 into the **union** of s1 and s2. (The union of two sets is the set containing all of the elements contained in either set.)
- s1.retainAll(s2) — transforms s1 into the intersection of s1 and s2. (The intersection of two sets is the set containing only the elements common to both sets.)
- s1.removeAll(s2) — transforms s1 into the (asymmetric) set difference of s1 and s2. (For example, the set difference of s1 minus s2 is the set containing all of the elements found in s1 but not in s2.)

To calculate the union, intersection, or set difference of two sets *nondestructively* (without modifying either set), the caller must copy one set before calling the appropriate bulk operation. The following are the resulting idioms.

```
Set<Type> union = new HashSet<Type>(s1);
union.addAll(s2);

Set<Type> intersection = new HashSet<Type>(s1);
intersection.retainAll(s2);

Set<Type> difference = new HashSet<Type>(s1);
difference.removeAll(s2);
```

The implementation type of the result Set in the preceding idioms is HashSet, which is, as already mentioned, the best all-around Set implementation in the Java platform. However, any general-purpose Set implementation could be substituted.

Let's revisit the FindDups program. Suppose you want to know which words in the argument list occur only once and which occur more than once, but you do not want any duplicates printed out repeatedly. This effect can be achieved by generating two sets — one containing

every word in the argument list and the other containing only the duplicates. The words that occur only once are the set difference of these two sets, which we know how to compute.

Here's how the resulting program looks.

```
import java.util.*;
public class FindDups2 {
   public static void main(String[] args) {
      Set<String> uniques = new HashSet<String>();
      Set<String> dups    = new HashSet<String>();

      for (String a : args)
         if (!uniques.add(a))
            dups.add(a);

      // Destructive set-difference
      uniques.removeAll(dups);

      System.out.println("Unique words:    " + uniques);
      System.out.println("Duplicate words: " + dups);
   }
}
```

When run with the same argument list used earlier (i came i saw i left), the program yields the following output.

```
Unique words:    [left, saw, came]
Duplicate words: [i]
```

6.1.2.3 Set Interface Array Operations

The array operations don't do anything special for Sets beyond what they do for any other Collection. These operations are described in The Collection Interface.

6.1.3 The Map Interface

A Map is an object that maps keys to values. A map cannot contain duplicate keys: Each key can map to at most one value. It models the mathematical *function* abstraction. The Map interface includes methods for basic operations (such as put, get, remove, containsKey, containsValue, size, and empty), bulk operations (such as putAll and clear), and collection views (such as keySet, entrySet, and values).

The Java platform contains three general-purpose Map implementations: HashMap, TreeMap, and LinkedHashMap. Their behavior and performance are precisely analogous to HashSet, TreeSet, and LinkedHashSet, as described in The Set Interface.

6.1.3.1 Map Interface Basic Operations

The basic operations of Map (put, get, containsKey, containsValue, size, and isEmpty) behave exactly like Hashtable methods. The following program generates a frequency table of the

words found in its argument list. The frequency table maps each word to the number of times it occurs in the argument list.

```
import java.util.*;
public class Freq {
   public static void main(String[] args) {
      Map<String, Integer> m = new HashMap<String, Integer>();

      // Initialize frequency table from command line
      for (String a : args) {
         Integer freq = m.get(a);
         m.put(a, (freq == null) ? 1 : freq + 1); //uses ternary operator for condition
                                                  //check
      }

      System.out.println(m.size() + " distinct words:");
      System.out.println(m);
   }
}
```

The only tricky thing about this program is the second argument of the put statement. That argument is a conditional expression that sets the frequency to one if frequency == null (word never seen) or one more than its current value if the word has already been seen. Run the program like this command-line input:

```
java Freq how do you do
```

The program yields the following output.

```
3 distinct words:
{do=2, how=1, you=1}
```

All general-purpose Map implementations provide constructors that take a Map object and initialize the new Map to contain all the key-value mappings in the specified Map. This standard Map conversion constructor is entirely analogous to the standard Collection constructor: It allows the caller to create a Map of a desired implementation type that initially contains all of the mappings in another Map, regardless of the other Map's implementation type.

For example, suppose you have a Map, named m. The following one-liner creates a new HashMap initially containing all of the same key-value mappings as m.

```
Map<K, V> copy = new HashMap<K, V>(m);
```

6.1.3.2 Map Interface Bulk Operations

The clear operation does exactly what you would think it could do: It removes all the mappings from the Map. The putAll operation is the Map analogue of the Collection interface's addAll operation.

```
}
```

6.1.3.3 Collection Views

The Collection view methods allow a Map to be viewed as a Collection in these three ways:

- keySet — the Set of keys contained in the Map.
- values — The Collection of values contained in the Map. This Collection is not a Set, because multiple keys can map to the same value.
- entrySet — the Set of key-value pairs contained in the Map. The Map interface provides a small nested interface called Map.Entry, the type of the elements in this Set.

The Collection views provide the *only* means to iterate over a Map. This example illustrates the standard idiom for iterating over the keys in a Map with a for-each construct:

```
for (KeyType key : m.keySet())
    System.out.println(key);
```

and with an iterator:

```
// Filter a map based on some
// property of its keys.
for (Iterator<Type> it = m.keySet().iterator(); it.hasNext(); )
    if (it.next().isBogus())
        it.remove();
```

The idiom for iterating over values is analogous. Following is the idiom for iterating over key-value pairs.

```
for (Map.Entry<KeyType, ValType> e : m.entrySet())
    System.out.println(e.getKey() + ": " + e.getValue());
```

At first, many people worry that these idioms may be slow because the Map has to create a new Collection instance each time a Collection view operation is called. Rest easy: There's no reason that a Map cannot always return the same object each time it is asked for a given Collection view. This is precisely what all the Map implementations in java.util do.
With all three Collection views, calling an Iterator's remove operation removes the associated entry from the backing Map, assuming that the backing Map supports element removal to begin with. This is illustrated by the preceding filtering idiom.

With the entrySet view, it is also possible to change the value associated with a key by calling a Map.Entry's setValue method during iteration (again, assuming the Map supports value modification to begin with). Note that these are the *only* safe ways to modify a Map during iteration; the behavior is unspecified if the underlying Map is modified in any other way while the iteration is in progress.

The Collection views support element removal in all its many forms — remove, removeAll, retainAll, and clear operations, as well as the Iterator.remove operation. (Yet again, this assumes that the backing Map supports element removal.)
The Collection views *do not* support element addition under any circumstances. It would make no sense for the keySet and values views, and it's unnecessary for the entrySet view, because the backing Map's put and putAll methods provide the same functionality.

The example below illustrates how to add and retrieve data from a Map.For retrieving the entrySet view is used.

```java
package com.tcs.ilp;

import java.util.Map;

public class AddRetrieveDataMap {
    //declare a hashmap object to accept key,value pairs of Integers,Strings
    Map<Integer, String> areaCodes = new HashMap<Integer,String>();

    public void addData(){
        areaCodes.put(40, "Hyderabad");
        areaCodes.put(44, "Chennai");
        areaCodes.put(80, "Bangalore");
        areaCodes.put(22, "Mumbai");
    }

    public void getData(){
        //iterate the hashmap using the for-each loop and by using the
        //collections views entryset
    for(Map.Entry<Integer, String> entry: areaCodes.entrySet()){
    System.out.println("Code:[" + entry.getKey()+"],Area:[" + entry.getValue()+"]");
        }
    }

    public static void main(String[] args) {
        AddRetrieveDataMap am = new AddRetrieveDataMap();
        // call addData() method to add data into the HashMap
        am.addData();
        // call getData method to retrieve data from HashMap
        System.out.println("List of areacodes and areas:\n");
        am.getData();

    }
}
```

Summary of Interfaces

The core collection interfaces are the foundation of the Java Collections Framework.
The Java Collections Framework hierarchy consists of two distinct interface trees:

- The first tree starts with the Collection interface, which provides for the basic functionality used by all collections, such as add and remove methods. Its subinterfaces — Set, List, and Queue — provide for more specialized collections.
- The Set interface does not allow duplicate elements. This can be useful for storing collections such as a deck of cards or student records. The Set interface has a subinterface, SortedSet, that provides for ordering of elements in the set.
- The List interface provides for an ordered collection, for situations in which you need precise control over where each element is inserted. You can retrieve elements from a List by their exact position.
- The Queue interface enables additional insertion, extraction, and inspection operations. Elements in a Queue are typically ordered in on a FIFO basis.
- The Deque interface enables insertion, deletion, and inspection operations at both the ends. Elements in a Deque can be used in both LIFO and FIFO.
- The second tree starts with the Map interface, which maps keys and values similar to a Hashtable.
- Map's subinterface, SortedMap, maintains its key-value pairs in ascending order or in an order specified by a Comparator.

These interfaces allow collections to be manipulated independently of the details of their representation.

6.1.4 Implementations:

Implementations are the data objects used to store collections, which implement the interfaces described in the Interfaces. They are the classes that implement the interfaces. Some of the implementing classes have additional or special-purpose methods to do additional operations. The different types of implementations(classes ultimately) are listed below:

- **General-purpose implementations** are the most commonly used implementations, designed for everyday use. They are summarized in the table titled General-purpose-implementations.
- **Special-purpose implementations** are designed for use in special situations and display nonstandard performance characteristics, usage restrictions, or behavior.
- **Concurrent implementations** are designed to support high concurrency, typically at the expense of single-threaded performance. These implementations are part of the java.util.concurrent package.
- **Wrapper implementations** are used in combination with other types of implementations, often the general-purpose ones, to provide added or restricted functionality.
- **Convenience implementations** are mini-implementations, typically made available via static factory methods, that provide convenient, efficient alternatives to general-purpose implementations for special collections (for example, singleton sets).
- **Abstract implementations** are skeletal implementations that facilitate the construction of custom implementations

The general-purpose implementations are summarized in the following table.

General-purpose Implementations

Interfaces	Hash table	Resizable array	Tree	Linked list	Hash table + Linked list
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue					
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

As you can see from the table, the Java Collections Framework provides several general-purpose implementations of the Set, List , and Map interfaces. In each case, one implementation — HashSet, ArrayList, and HashMap — is clearly the one to use for most

applications, all other things being equal. Note that the SortedSet and the SortedMap interfaces do not have rows in the table. Each of those interfaces has one implementation (TreeSet and TreeMap) and is listed in the Set and the Map rows. There are two general-purpose Queue implementations — LinkedList, which is also a List implementation, and PriorityQueue, which is omitted from the table. These two implementations provide very different semantics: LinkedList provides FIFO semantics, while PriorityQueue orders its elements according to their values(natural ordering).

Each of the general-purpose implementations provides all optional operations contained in its interface. All permit null elements, keys, and values. None are synchronized (thread-safe). All have *fail-fast iterators*, which detect illegal concurrent modification during iteration and fail quickly and cleanly rather than risking arbitrary, nondeterministic behavior at an undetermined time in the future. All are Serializable and all support a public clone method.

The fact that these implementations are unsynchronized represents a break with the past: The legacy collections Vector and Hashtable are synchronized. The present approach was taken because collections are frequently used when the synchronization is of no benefit. Such uses include single-threaded use, read-only use, and use as part of a larger data object that does its own synchronization. In general, it is good API design practice not to make users pay for a feature they don't use. Furthermore, unnecessary synchronization can result in deadlock under certain circumstances.

If you need thread-safe collections, the synchronization wrappers, allow *any* collection to be transformed into a synchronized collection. Thus, synchronization is optional for general-purpose implementations, whereas it is mandatory for legacy implementations. Moreover, the java.util.concurrent package provides concurrent implementations of the BlockingQueue interface, which extends Queue, and of the ConcurrentMap interface, which extends Map. These implementations offer much higher concurrency than mere synchronized implementations.

As a rule, you should be thinking about the interfaces, *not* the implementations. For the most part, the choice of implementation affects only performance. The preferred style, as mentioned in the Interfaces section, is to choose an implementation(class) when a Collection is created and to immediately assign the new collection to a variable of the corresponding interface type (or to pass the collection to a method expecting an argument of the interface type). In this way, the program does not become dependent on any added methods in a given implementation, leaving the programmer free to change implementations anytime that it is warranted by performance concerns or behavioral details.

6.1.4.1 Set Implementations

The Set implementations are grouped into general-purpose and special-purpose implementations. We will only discuss the general implementations.

General-Purpose Set Implementations

There are three general-purpose Set implementations — HashSet, TreeSet, and LinkedHashSet. Which of these three to use is generally straightforward. HashSet is much faster than TreeSet (constant-time versus log-time for most operations) but offers no ordering guarantees. If you need to use the operations in the SortedSet interface, or if value-ordered iteration is required, use TreeSet; otherwise, use HashSet. It's a fair bet that you'll end up using HashSet most of the time.

LinkedHashSet is in some sense intermediate between HashSet and TreeSet. Implemented as a hash table with a linked list running through it, it provides *insertion-ordered* iteration (least recently inserted to most recently) and runs nearly as fast as HashSet. The LinkedHashSet implementation spares its clients from the unspecified, generally chaotic ordering provided by HashSet without incurring the increased cost associated with TreeSet.

One thing worth keeping in mind about HashSet is that iteration is linear in the sum of the number of entries and the number of buckets (the *capacity*). Thus, choosing an initial capacity that's too high can waste both space and time. On the other hand, choosing an initial capacity that's too low wastes time by copying the data structure each time it's forced to increase its capacity. If you don't specify an initial capacity, the default is 16. Internally, the capacity is always rounded up to a power of two. The initial capacity is specified by using the int constructor. The following line of code allocates a HashSet whose initial capacity is 64.

```
Set<String> s = new HashSet<String>(64);
```

The HashSet class has one other tuning parameter called the *load factor*. If you care a lot about the space consumption of your HashSet, read the HashSet documentation for more information. Otherwise, just accept the default.

If you accept the default load factor but want to specify an initial capacity, pick a number that's about twice the size to which you expect the set to grow. If your guess is way off, you may waste a bit of space, time, or both, but it's unlikely to be a big problem.

LinkedHashSet has the same tuning parameters as HashSet, but iteration time is not affected by capacity. TreeSet has no tuning parameters.

6.1.4.2 List Implementations

List implementations are grouped into general-purpose and special-purpose implementations. We will only discuss general-purpose implementations.

General-Purpose List Implementations

There are two general-purpose List implementations — ArrayList and LinkedList. Most of the time, you'll probably use ArrayList, which offers constant-time positional access and is just plain fast. It does not have to allocate a node object for each element in the List, and it can take advantage of System.arraycopy when it has to move multiple elements at the same time. Think of ArrayList as Vector without the synchronization overhead.

If you frequently add elements to the beginning of the List or iterate over the List to delete elements from its interior, you should consider using LinkedList. These operations require constant-time in a LinkedList and linear-time in an ArrayList. But you pay a big price in performance. Positional access requires linear-time in a LinkedList and constant-time in an ArrayList. Furthermore, the constant factor for LinkedList is much worse. If you think you want to use a LinkedList, measure the performance of your application with both LinkedList and ArrayList before making your choice; ArrayList is usually faster.

ArrayList has one tuning parameter — the *initial capacity*, which refers to the number of elements the ArrayList can hold before it has to grow. LinkedList has no tuning parameters and seven optional operations, one of which is clone. The other six are addFirst, getFirst, removeFirst, addLast, getLast, and removeLast. LinkedList also implements the Queue interface.

6.1.4.3 Map Implementations

Map implementations are grouped into general-purpose, special-purpose, and concurrent implementations.

General-Purpose Map Implementations

The three general-purpose Map implementations are HashMap, TreeMap and LinkedHashMap. If you need SortedMap operations or key-ordered Collection-view iteration, use TreeMap; if you want maximum speed and don't care about iteration order, use HashMap; if you want near-HashMap performance and insertion-order iteration, use LinkedHashMap. In this respect, the situation for Map is analogous to Set. Likewise, everything else in the Set Implementations section also applies to Map implementations.

LinkedHashMap provides two capabilities that are not available with LinkedHashSet. When you create a LinkedHashMap, you can order it based on key access rather than insertion. In other words, merely looking up the value associated with a key brings that key to the end of the map. Also, LinkedHashMap provides the removeEldestEntry method, which may be overridden to impose a policy for removing stale mappings automatically when new mappings are added to the map. This makes it very easy to implement a custom cache.

For example, this override will allow the map to grow up to as many as 100 entries and then it will delete the eldest entry each time a new entry is added, maintaining a steady state of 100 entries.

```
private static final int MAX_ENTRIES = 100;

protected boolean removeEldestEntry(Map.Entry eldest) {
   return size() > MAX_ENTRIES;
}
```

Concurrent Map Implementations

The java.util.concurrent package contains the ConcurrentMap interface, which extends Map with atomic putIfAbsent, remove, and replace methods, and the ConcurrentHashMap implementation of that interface.

ConcurrentHashMap is a highly concurrent, high-performance implementation backed up by a hash table. This implementation never blocks when performing retrievals and allows the client to select the concurrency level for updates. It is intended as a drop-in replacement for Hashtable: in addition to implementing ConcurrentMap, it supports all the legacy methods peculiar to Hashtable. Again, if you don't need the legacy operations, be careful to manipulate it with the ConcurrentMap interface.

Summary of Implementations

Implementations are the data objects(classes) used to store collections, which implement the interfaces in the Collections Framework.

The Java Collections Framework provides several general-purpose implementations of the core interfaces:

- For the Set interface, HashSet is the most commonly used implementation.
- For the List interface, ArrayList is the most commonly used implementation.
- For the Map interface, HashMap is the most commonly used implementation.
- For the Queue interface, LinkedList is the most commonly used implementation.

Each of the general-purpose implementations provides all optional operations contained in its interface.

The Java Collections Framework also provides several special-purpose implementations for situations that require nonstandard performance, usage restrictions, or other unusual behavior.

The java.util.concurrent package contains several collections implementations, which are thread-safe but not governed by a single exclusion lock.

The Collections class (as opposed to the Collection interface), provides static methods that operate on or return collections, which are known as Wrapper implementations.

Finally, there are several Convenience implementations, which can be more efficient than general-purpose implementations when you don't need their full power. The Convenience implementations are made available through static factory methods.

6.1.5 Algorithms:

The *polymorphic algorithms* described here are pieces of reusable functionality provided by the Java platform. All of them come from the Collections class, and all take the form of static methods whose first argument is the collection on which the operation is to be performed. The great majority of the algorithms provided by the Java platform operate on List instances, but a few of them operate on arbitrary Collection instances. The following are some of the operations that use the polymorphic algorithms on the collections.

- Sorting
- Shuffling
- Routine Data Manipulation
- Searching

6.2 Using Collections:

6.2.1 Store and Retrieve data from ArrayList

ArrayList implements the List interface of the collection framework. It can have duplicate elements and is one of the most commonly used collection object in java programming. Data elements can be added into an ArrayList using the add(Object o) method.

The data elements can be retrieved using multiple ways such as using a for-each loop, or using the iterator to traverse the collection and retrieve the elements or simply using the built-in methods. While retrieving data the data is returned in the order of insertion.

Let us see how to add elements into an ArrayList and retrieve them.

```java
import java.util.ArrayList;
import java.util.List;

public class AddRetrieveData {
    // declare an ArrayList that will store only String type data
    List<String> cityList = new ArrayList<String>();

    // add data into the cityList object
    public void addData(){
        cityList.add(new String("Hyderabad"));
        cityList.add(new String("Chennai"));
        cityList.add(new String("Bangalore"));
        cityList.add(new String("Mumbai"));
    }
    // retrieve data from the cityList object
    public void retrieveData(){
        // retrieve the String elements from cityList using for-each lool
        for(String s: cityList){
            System.out.println(s);
        }
    }
    //the main method
    public static void main(String[] args) {
        AddRetrieveData ad = new AddRetrieveData();
        //call addData() method to add data elements into the ArrayList
        ad.addData();

        //call retrieveData() method to retrieve and print the elements
        System.out.println("List of Cities:\n");
        ad.retrieveData();
    }
}
```

The output of the program looks like this:

```
List of Cities:

Hyderabad
Chennai
Bangalore
Mumbai
```

6.2.2 Store and Retrieve data from HashMap

The data on a hashmap is stored as Key,Value pairs. A given value or object will be identified by a unique key. To obtain the value the key will be used for searching and obtain the value. The order in which values will be retrieved in not in any given order but random.

```
import java.util.Map;
import java.util.HashMap;

public class AddRetrieveDataMap {
    //declare a hashmap object to accept key,value pairs of Integers,Strings
    Map<Integer, String> areaCodes = new HashMap<Integer,String>();

    public void addData(){
        areaCodes.put(40, "Hyderabad");
        areaCodes.put(44, "Chennai");
        areaCodes.put(80, "Bangalore");
        areaCodes.put(22, "Mumbai");
    }

    public void getData(){
        //iterate the hashmap using the for-each loop and by using the
        //collections views entryset
    for(Map.Entry<Integer, String> entry: areaCodes.entrySet()){
    System.out.println("Code:[" + entry.getKey()+"],Area:[" + entry.getValue()+"]");
        }
    }

    public static void main(String[] args) {
        AddRetrieveDataMap am = new AddRetrieveDataMap();
        // call addData() method to add data into the HashMap
        am.addData();
        // call getData method to retrieve data from HashMap
        System.out.println("List of areacodes and areas:\n");
        am.getData();

    }
}
```

The output of the program looks like this:

List of areacodes and areas:
Code:[80],Area:[Bangalore]
Code:[22],Area:[Mumbai]
Code:[40],Area:[Hyderabad]
Code:[44],Area:[Chennai]

6.2.3 Modifying Collections

Modifying or updating the collections is possible if Iterator or ListIterator is used, which obtain a reference to the collection and its elements and hence making it possible to update a collection.

Iterator and ListIterator can be used to remove elements from a collection. Whereas to update the existing objects in-place on the collection it is possible only by using the ListIterator.

6.2.3.1 Updating Collections:

The program below updates or makes changes to the String objects that are already existing on the collection by using the ListIterator. Note that Listiterator only works on collections of type List and its subtypes.

The program below updates the strings on the collection

```
import java.util.ListIterator;
public class AddRetrieveData {
    // declare an ArrayList that will store only String type data
    List<String> cityList = new ArrayList<String>();
    // add data into the cityList object
    public void addData(){
        cityList.add(new String("Hyderabad"));
        cityList.add(new String("Chennai"));
        cityList.add(new String("Bangalore"));
        cityList.add(new String("Mumbai"));
    }

    public void retrieveData(){
        Iterator<String> itr = cityList.iterator();
        while(itr.hasNext())
            System.out.println(itr.next());
    }
    public void updateData(){
        ListIterator<String> litr = cityList.listIterator();
        while(litr.hasNext()){
            String city = litr.next();
            //change or update the String object in the arraylist
            litr.set(city + " city");
        }
    }
    public static void main(String[] args) {
        AddRetrieveData ad = new AddRetrieveData();
        //call addData() method to add data elements into the ArrayList
        ad.addData();
        System.out.println("Before updating the collection");
        ad.retrieveData();
        //call retrieveData() method to retrieve and print the elements
        ad.updateData();
        System.out.println("After updating the collection");
        ad.retrieveData();
        }
```

The output of the program will look like this:
Before updating the collection
Hyderabad
Chennai
Bangalore
Mumbai

After updating the collection
Hyderabad city
Chennai city

Bangalore city
Mumbai city

6.2.3.2 Deleting elements from Collections

The program below removes elements from the collection. In order to remove elements from a collection you need to use Iterator. The remove() method will remove the element at which the hasNext() method points to (which is the current element).

```
import java.util.ListIterator;
public class AddRetrieveData {
    // declare an ArrayList that will store only String type data
    List<String> cityList = new ArrayList<String>();
    // add data into the cityList object
    public void addData(){
        cityList.add(new String("Hyderabad"));
        cityList.add(new String("Chennai"));
        cityList.add(new String("Bangalore"));
        cityList.add(new String("Mumbai"));
    }

    public void retrieveData(){
        Iterator<String> itr = cityList.iterator();
        while(itr.hasNext())
            System.out.println(itr.next());
    }

    public void deleteData(){
        Iterator<String> itr = cityList.iterator();
        while(itr.hasNext()){
            String city = itr.next();
            if (city.equals("Chennai"))
                itr.remove();
        }
    }
        public static void main(String[] args) {
        AddRetrieveData ad = new AddRetrieveData();
        //call addData() method to add data elements into the ArrayList
        ad.addData();
        System.out.println("Before deleting from collection");
        ad.retrieveData();
        // call delete data
        ad.deleteData();
        System.out.println("After deleting from collection");
        ad.retrieveData();
        }
```

6.3 Traversing Collections:

The collections need to be traversed or iterated through in order to retrieve the elements of the collections. There are multiple ways of traversing collections and retrieving the elements.

6.3.1 Traversing Collections using Iterators

Iterator enables you to cycle through a collection, obtaining or removing elements. ListIterator extends Iterator to allow bidirectional traversal of a list, and the modification of elements.

Before you can access a collection through an iterator, you must obtain one. Each of the collection classes provides an iterator( ) method that returns an iterator to the start of the collection. By using this iterator object, you can access each element in the collection, one element at a time.

In general, to use an iterator to cycle through the contents of a collection, follow these steps:

- Obtain an iterator to the start of the collection by calling the collection's iterator( ) method
- Set up a loop that makes a call to hasNext( ). Have the loop iterate as long as hasNext( ) returns true
- Within the loop, obtain each element by calling next( )

For collections that implement List, you can also obtain an iterator by calling ListIterator. ListIterator allows to traverse collection in both directions. The hasPrevious() method is used to traverse the collection in the reverse order.

The Methods Declared by Iterator:

SN	Methods with Description
1	boolean hasNext( ) Returns true if there are more elements. Otherwise, returns false.
2	Object next( ) Returns the next element. Throws NoSuchElementException if there is not a next element.
3	void remove( ) Removes the current element. Throws IllegalStateException if an attempt is made to call remove( ) that is not preceded by a call to next( ).

The Methods Declared by ListIterator:

SN	Methods with Description
1	void add(Object obj) Inserts obj into the list in front of the element that will be returned by the next call to next( ).
2	boolean hasNext( ) Returns true if there is a next element. Otherwise, returns false.
3	boolean hasPrevious( ) Returns true if there is a previous element. Otherwise, returns false.
4	Object next( ) Returns the next element. A NoSuchElementException is thrown if there is not a next element.

5	int nextIndex( ) Returns the index of the next element. If there is not a next element, returns the size of the list.
6	Object previous( ) Returns the previous element. A NoSuchElementException is thrown if there is not a previous element.
7	int previousIndex( ) Returns the index of the previous element. If there is not a previous element, returns -1.
8	void remove( ) Removes the current element from the list. An IllegalStateException is thrown if remove( ) is called before next( ) or previous( ) is invoked.
9	void set(Object obj) Assigns obj to the current element. This is the element last returned by a call to either next( ) or previous( ).

6.3.1.1 Traversing collection (ArrayList) using Iterator

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
public class AddRetrieveData {
    // declare an ArrayList that will store only String type data
    List<String> cityList = new ArrayList<String>();
    // add data into the cityList object
    public void addData(){
        cityList.add(new String("Hyderabad"));
        cityList.add(new String("Chennai"));
        cityList.add(new String("Bangalore"));
        cityList.add(new String("Mumbai"));
    }
    // retrieve data from the cityList object
    public void retrieveData(){
        //creates an iterator on the cityList collection
        Iterator<String> itr = cityList.iterator();
        //if the collection has a next element hasNext() returns true
        while(itr.hasNext()){
            //next() method retrieves the object
            System.out.println(itr.next());
        }
    }
    //the main method
    public static void main(String[] args) {
        AddRetrieveData ad = new AddRetrieveData();
        //call addData() method to add data elements into the ArrayList
        ad.addData();
        //call retrieveData() method to retrieve and print the elements
        System.out.println("List of Cities:\n");
        ad.retrieveData();
        }
```

6.3.1.2 Traversing Collection (ArrayList) using ListIterator:

ListIterator allows to traverse a collection in both directions. Once the end of the collection is reached it can be traversed in the reverse direction. ListIterator is obtained by calling the listIterator() method on a collection which returns a listiterator instance. ListIterator works only on collections of type List and its sub-types.

```
import java.util.ListIterator;
public class AddRetrieveData {
    // declare an ArrayList that will store only String type data
    List<String> cityList = new ArrayList<String>();
    // add data into the cityList object
    public void addData(){
        cityList.add(new String("Hyderabad"));
        cityList.add(new String("Chennai"));
        cityList.add(new String("Bangalore"));
        cityList.add(new String("Mumbai"));
    }
    public void retrieveData(){
        // obtain a listiterator on the collection
        ListIterator<String> litr = cityList.listIterator();
        //first traverse to the end of the list
        System.out.println("\nList of Cities while traversing in forward direction:\n");
        while(litr.hasNext()){
            System.out.println(litr.next());
        }
        //traverse in the reverse direction
        System.out.println("\nList of Cities while traversing in reverse direction:\n");
        while(litr.hasPrevious()){
            System.out.println(litr.previous());
        }
    }
    public static void main(String[] args) {
        AddRetrieveData ad = new AddRetrieveData();
        //call addData() method to add data elements into the ArrayList
        ad.addData();
        //call retrieveData() method to retrieve and print the elements
        ad.retrieveData();
        }
```

The output of the program looks likes this:
List of Cities while traversing in forward direction:
Hyderabad
Chennai
Bangalore
Mumbai

List of Cities while traversing in reverse direction:
Mumbai
Bangalore
Chennai
Hyderabad

6.3.1 Traversing Collections using enhanced for-loop or for-each loop:
The enhanced for-loop retrieves the data elements on the collection and reads them into a temporary variable.

```
import java.util.ArrayList;
import java.util.List;
public class AddRetrieveData {
    // declare an ArrayList that will store only String type data
    List<String> cityList = new ArrayList<String>();
    // add data into the cityList object
    public void addData(){
        cityList.add(new String("Hyderabad"));
        cityList.add(new String("Chennai"));
        cityList.add(new String("Bangalore"));
        cityList.add(new String("Mumbai"));
    }

    public void retrieveData(){
        // retrieve the String elements from cityList using for-each loop
        // the for-each loop retrieves the String objects into a temporary
        // String variable
        for(String s: cityList){
            System.out.println(s);
        }
    }
    //the main method
    public static void main(String[] args) {
        AddRetrieveData ad = new AddRetrieveData();
        //call addData() method to add data elements into the ArrayList
        ad.addData();
        //call retrieveData() method to retrieve and print the elements
        System.out.println("List of Cities:\n");
        ad.retrieveData();
        }
}
```

6.3.2 Traversing Collections using standard methods:

The collections can be traversed using the standard for loop. The size() method returns the int size of the collection allowing to traverse the collection size() times. The get(index) method retrieves object at the index specified.

```
import java.util.ArrayList;
import java.util.List;

public class AddRetrieveData {
    // declare an ArrayList that will store only String type data
    List<String> cityList = new ArrayList<String>();
    // add data into the cityList object
    public void addData(){
        cityList.add(new String("Hyderabad"));
        cityList.add(new String("Chennai"));
        cityList.add(new String("Bangalore"));
        cityList.add(new String("Mumbai"));
    }

    //retrieve data using the regular for loop
    public void retrieveData(){
        for(int i=0; i<cityList.size();i++)
            System.out.println(cityList.get(i));
        }
        public static void main(String[] args) {
        AddRetrieveData ad = new AddRetrieveData();
        //call addData() method to add data elements into the ArrayList
        ad.addData();
        System.out.println("List of Cities\n");
        //call retrieveData method to traverse and retrieve data from collection
        ad.retrieveData();
        }
}
```

6.4 Performing Operations on Collections Using Polymorphic Algorithms

6.4.1 Sorting Lists

The sort algorithm reorders a List so that its elements are in ascending order according to an ordering relationship. Two forms of the operation are provided. The simple form takes a List and sorts it according to its elements' *natural ordering*.

If the List consists of String elements, it will be sorted into alphabetical order. If it consists of Date elements, it will be sorted into chronological order. How does this happen? String and Date both implement the Comparable interface. Comparable implementations provide a natural ordering for a class, which allows objects of that class to be sorted automatically.

The sort operation uses a slightly optimized *merge sort* algorithm that is fast and stable:

- **Fast**: It is guaranteed to run in n log(n) time and runs substantially faster on nearly sorted lists. Empirical tests showed it to be as fast as a highly optimized quicksort. A

quicksort is generally considered to be faster than a merge sort but isn't stable and doesn't guarantee n log(n) performance.

- **Stable**: It doesn't reorder equal elements. This is important if you sort the same list repeatedly on different attributes. If a user of a mail program sorts the inbox by mailing date and then sorts it by sender, the user naturally expects that the now-contiguous list of messages from a given sender will (still) be sorted by mailing date. This is guaranteed only if the second sort was stable.

The following trivial program prints out its arguments in lexicographic (alphabetical) order.

```
import java.util.*;

public class Sort {
   public static void main(String[] args) {
      List<String> list = Arrays.asList(args);
      Collections.sort(list);
      System.out.println(list);
   }
}
```

Run the program.

% java Sort i walk the line

The following output is produced.

[i, line, the, walk]

The program below sorts the elements in their alphabetical order.

```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;
import java.util.List;

public class AddRetrieveData {
    // declare an ArrayList that will store only String type data
    List<String> cityList = new ArrayList<String>();
    // add data into the cityList object
    public void addData(){
        cityList.add(new String("Hyderabad"));
        cityList.add(new String("Chennai"));
        cityList.add(new String("Bangalore"));
        cityList.add(new String("Mumbai"));
    }

    public void retrieveData(){
        Iterator<String> itr = cityList.iterator();
        while(itr.hasNext())
            System.out.println(itr.next());
    }

    public void sortData(){
        Collections.sort(cityList);
    }
        public static void main(String[] args) {
        AddRetrieveData ad = new AddRetrieveData();
        //call addData() method to add data elements into the ArrayList
        ad.addData();
        System.out.println("Before sorting the collection");
        ad.retrieveData();
        ad.sortData();
        System.out.println("After sorting the collection");
        ad.retrieveData();
        }
}
```

The output will look like this:

--Before sorting the collection

Hyderabad
Chennai
Bangalore
Mumbai
--After sorting the collection
Bangalore
Chennai
Hyderabad
Mumbai

The second form of sort takes a Comparator in addition to a List and sorts the elements with the Comparator.

6.4.2 Searching Collections with Binary Search

The binarySearch algorithm searches for a specified element in a sorted List. This algorithm has two forms. The first takes a List and an element to search for (the "search key"). This form assumes that the List is sorted in ascending order according to the natural ordering of its elements. The second form takes a Comparator in addition to the List and the search key, and assumes that the List is sorted into ascending order according to the specified Comparator. The sort algorithm can be used to sort the List prior to calling binarySearch.

The return value is the same for both forms. If the List contains the search key, its index is returned. If not, the return value is (-(insertion point) - 1), where the insertion point is the point at which the value would be inserted into the List, or the index of the first element greater than the value or list.size() if all elements in the List are less than the specified value. This admittedly ugly formula guarantees that the return value will be >= 0 if and only if the search key is found. It's basically a hack to combine a boolean (found) and an integer (index) into a single int return value.

The following idiom, usable with both forms of the binarySearch operation, looks for the specified search key and inserts it at the appropriate position if it's not already present.

```
int pos = Collections.binarySearch(list, key);
if (pos < 0)
  l.add(-pos-1, key);
```

6.4.3 Other Operations on Collections using Algorithms

6.4.3.1 Shuffling

The shuffle algorithm does the opposite of what sort does, destroying any trace of order that may have been present in a List. That is, this algorithm reorders the List based on input from a source of randomness such that all possible permutations occur with equal likelihood, assuming a fair source of randomness. This algorithm is useful in implementing games of chance. For example, it could be used to shuffle a List of Card objects representing a deck. Also, it's useful for generating test cases.

This operation has two forms: one takes a List and uses a default source of randomness, and the other requires the caller to provide a Random object to use as a source of randomness.

6.4.3.2 Routine Data Manipulation

The Collections class provides five algorithms for doing routine data manipulation on List objects, all of which are pretty straightforward:

- reverse — reverses the order of the elements in a List.
- fill — overwrites every element in a List with the specified value. This operation is useful for reinitializing a List.

- copy — takes two arguments, a destination List and a source List, and copies the elements of the source into the destination, overwriting its contents. The destination List must be at least as long as the source. If it is longer, the remaining elements in the destination List are unaffected.
- swap — swaps the elements at the specified positions in a List.
- addAll — adds all the specified elements to a Collection. The elements to be added may be specified individually or as an array.

6.4.3.3 Finding Extreme Values

The min and the max algorithms return, respectively, the minimum and maximum element contained in a specified Collection. Both of these operations come in two forms. The simple form takes only a Collection and returns the minimum (or maximum) element according to the elements' natural ordering. The second form takes a Comparator in addition to the Collection and returns the minimum (or maximum) element according to the specified Comparator.

6.5 Understanding Comparable and Comparator

6.5.1 Object Ordering

A List l may be sorted as follows.

Collections.sort(l);

If the List consists of String elements, it will be sorted into alphabetical order. If it consists of Date elements, it will be sorted into chronological order. How does this happen? String and Date both implement the Comparable interface. Comparable implementations provide a *natural ordering* for a class, which allows objects of that class to be sorted automatically. The following table summarizes some of the more important Java platform classes that implement Comparable.

Classes Implementing Comparable

Class	Natural Ordering
Byte	Signed numerical
Character	Unsigned numerical
Long	Signed numerical
Integer	Signed numerical
Short	Signed numerical
Double	Signed numerical
Float	Signed numerical
BigInteger	Signed numerical
BigDecimal	Signed numerical

Boolean	Boolean.FALSE < Boolean.TRUE
File	System-dependent lexicographic on path name
String	Lexicographic
Date	Chronological
CollationKey	Locale-specific lexicographic

If you try to sort a list, the elements of which do not implement Comparable, Collections.sort(list) will throw a ClassCastException. Similarly, Collections.sort(list, comparator) will throw a ClassCastException if you try to sort a list whose elements cannot be compared to one another using the comparator. Elements that can be compared to one another are called *mutually comparable*. Although elements of different types may be mutually comparable, none of the classes listed here permit interclass comparison.
This is all you really need to know about the Comparable interface if you just want to sort lists of comparable elements or to create sorted collections of them.

6.5.2 Implementing or Overriding Comparable and Comparator
The Comparable interface consists of the following method.

```
public interface Comparable<T> {
  public int compareTo(T o);
}
```

The compareTo method compares the receiving object with the specified object and returns a negative integer, 0, or a positive integer depending on whether the receiving object is less than, equal to, or greater than the specified object. If the specified object cannot be compared to the receiving object, the method throws a ClassCastException.
The following class representing a person's name implements Comparable.

```
import java.util.*;
public class Name implements Comparable<Name> {
  private final String firstName, lastName;

  public Name(String firstName, String lastName) {
    if (firstName == null || lastName == null)
      throw new NullPointerException();
    this.firstName = firstName;
    this.lastName = lastName;
  }

  public String firstName() { return firstName; }
  public String lastName()  { return lastName;  }

  public boolean equals(Object o) {
```

```
        if (!(o instanceof Name))
            return false;
        Name n = (Name) o;
        return n.firstName.equals(firstName) && n.lastName.equals(lastName);
    }

    public int hashCode() {
        return 31*firstName.hashCode() + lastName.hashCode();
    }

    public String toString() {
            return firstName + " " + lastName;
    }

    public int compareTo(Name n) {
        int lastCmp = lastName.compareTo(n.lastName);
        return (lastCmp != 0 ? lastCmp : firstName.compareTo(n.firstName));
    }
}
```

To keep the preceding example short, the class is somewhat limited: It doesn't support middle names, it demands both a first and a last name, and it is not internationalized in any way. Nonetheless, it illustrates the following important points:

- Name objects are *immutable*. All other things being equal, immutable types are the way to go, especially for objects that will be used as elements in Sets or as keys in Maps. These collections will break if you modify their elements or keys while they're in the collection.
- The constructor checks its arguments for null. This ensures that all Name objects are well formed so that none of the other methods will ever throw a NullPointerException.
- The hashCode method is redefined. This is essential for any class that redefines the equals method. (Equal objects must have equal hash codes.)
- The equals method returns false if the specified object is null or of an inappropriate type. The compareTo method throws a runtime exception under these circumstances. Both of these behaviors are required by the general contracts of the respective methods.
- The toString method has been redefined so it prints the Name in human-readable form. This is always a good idea, especially for objects that are going to get put into collections. The various collection types' toString methods depend on the toString methods of their elements, keys, and values.

Let's talk a bit more about Name's compareTo method. It implements the standard name-ordering algorithm, where last names take precedence over first names. This is exactly what you want in a natural ordering. It would be very confusing indeed if the natural ordering were unnatural!

Take a look at how compareTo is implemented, because it's quite typical. First, you compare the most significant part of the object (in this case, the last name). Often, you can just use the natural ordering of the part's type. In this case, the part is a String and the natural (lexicographic) ordering is exactly what's called for. If the comparison results in anything other than zero, which represents equality, you're done: You just return the result. If the most significant parts are equal, you go on to compare the next most-significant parts. In this case, there are only two parts — first name and last name. If there were more parts, you'd proceed in the obvious fashion, comparing parts until you found two that weren't equal or you were comparing the least-significant parts, at which point you'd return the result of the comparison.
Just to show that it all works, here's a program that builds a list of names and sorts them.

```
import java.util.*;
public class NameSort {
   public static void main(String[] args) {
      Name nameArray[] = {
         new Name("John", "Smith"),
         new Name("Karl", "Ng"),
         new Name("Jeff", "Smith"),
         new Name("Tom", "Rich")
      };

      List<Name> names = Arrays.asList(nameArray);
      Collections.sort(names);
      System.out.println(names);
   }
}
```

If you run this program, here's what it prints.

```
[Karl Ng, Tom Rich, Jeff Smith, John Smith]
```

There are four restrictions on the behavior of the compareTo method, which we won't go into now because they're fairly technical and boring and are better left in the API documentation. It's really important that all classes that implement Comparable obey these restrictions, so read the documentation for Comparable if you're writing a class that implements it. Attempting to sort a list of objects that violate the restrictions has undefined behavior. Technically speaking, these restrictions ensure that the natural ordering is a *total order* on the objects of a class that implements it; this is necessary to ensure that sorting is well defined.

6.5.3 Comparators

What if you want to sort some objects in an order other than their natural ordering? Or what if you want to sort some objects that don't implement Comparable? To do either of these things, you'll need to provide a Comparator — an object that encapsulates an

ordering. Like the Comparable interface, the Comparator interface consists of a single method.

```
public interface Comparator<T> {
   int compare(T o1, T o2);
}
```

The compare method compares its two arguments, returning a negative integer, 0, or a positive integer depending on whether the first argument is less than, equal to, or greater than the second. If either of the arguments has an inappropriate type for the Comparator, the compare method throws a ClassCastException.

Much of what was said about Comparable applies to Comparator as well. Writing a compare method is nearly identical to writing a compareTo method, except that the former gets both objects passed in as arguments. The compare method has to obey the same four technical restrictions as Comparable's compareTo method for the same reason — a Comparator must induce a total order on the objects it compares.

Suppose you have a class called Employee, as follows.

```
public class Employee implements Comparable<Employee> {
   public Name name()     { ... }
   public int number()    { ... }
   public Date hireDate() { ... }
      ...
}
```

Let's assume that the natural ordering of Employee instances is Name ordering (as defined in the previous example) on employee name. Unfortunately, the boss has asked for a list of employees in order of seniority. This means we have to do some work, but not much. The following program will produce the required list.

```
import java.util.*;
public class EmpSort {
   static final Comparator<Employee> SENIORITY_ORDER =
                                 new Comparator<Employee>() {
         public int compare(Employee e1, Employee e2) {
            return e2.hireDate().compareTo(e1.hireDate());
         }
   };

   // Employee database
   static final Collection<Employee> employees = ... ;

   public static void main(String[] args) {
      List<Employee> e = new ArrayList<Employee>(employees);
```

```
        Collections.sort(e, SENIORITY_ORDER);
        System.out.println(e);
    }
}
```

The Comparator in the program is reasonably straightforward. It relies on the natural ordering of Date applied to the values returned by the hireDate accessor method. Note that the Comparator passes the hire date of its second argument to its first rather than vice versa. The reason is that the employee who was hired most recently is the least senior; sorting in the order of hire date would put the list in reverse seniority order. Another technique people sometimes use to achieve this effect is to maintain the argument order but to negate the result of the comparison.

```
// Don't do this!!
return -r1.hireDate().compareTo(r2.hireDate());
```

You should always use the former technique in favor of the latter because the latter is not guaranteed to work. The reason for this is that the compareTo method can return any negative int if its argument is less than the object on which it is invoked. There is one negative int that remains negative when negated, strange as it may seem.

```
-Integer.MIN_VALUE == Integer.MIN_VALUE
```

The Comparator in the preceding program works fine for sorting a List, but it does have one deficiency: It cannot be used to order a sorted collection, such as TreeSet, because it generates an ordering that is *not compatible with* equals. This means that this Comparator equates objects that the equals method does not. In particular, any two employees who were hired on the same date will compare as equal. When you're sorting a List, this doesn't matter; but when you're using the Comparator to order a sorted collection, it's fatal. If you use this Comparator to insert multiple employees hired on the same date into a TreeSet, only the first one will be added to the set; the second will be seen as a duplicate element and will be ignored.
To fix this problem, simply tweak the Comparator so that it produces an ordering that *is compatible with* equals. In other words, tweak it so that the only elements seen as equal when using compare are those that are also seen as equal when compared using equals. The way to do this is to perform a two-part comparison (as for Name), where the first part is the one we're interested in — in this case, the hire date — and the second part is an attribute that uniquely identifies the object. Here the employee number is the obvious attribute. This is the Comparator that results.

```
static final Comparator<Employee> SENIORITY_ORDER =
                                    new Comparator<Employee>() {
    public int compare(Employee e1, Employee e2) {
        int dateCmp = e2.hireDate().compareTo(e1.hireDate());
        if (dateCmp != 0)
```

```
        return dateCmp;

    return (e1.number() < e2.number() ? -1 :
        (e1.number() == e2.number() ? 0 : 1));
  }
};
```

One last note: You might be tempted to replace the final return statement in the Comparator with the simpler:

```
return e1.number() - e2.number();
```

Don't do it unless you're *absolutely sure* no one will ever have a negative employee number! This trick does not work in general because the signed integer type is not big enough to represent the difference of two arbitrary signed integers. If i is a large positive integer and j is a large negative integer, i - j will overflow and will return a negative integer. The resulting comparator violates one of the four technical restrictions we keep talking about (transitivity) and produces horrible, subtle bugs.

6.6 Best Practices

- Choose the right data structure: Based on usage patterns like required to grow or fixed size, duplicates allowed or not, ordering is required to be maintained or not, traversal is forward only or bi-directional, inserts at the end only or any arbitrary position, more reads or more inserts, concurrently accessed or not, modification is allowed or not, heterogeneous or homogeneous collection, etc. Also, keep multi-threading, atomicity, memory usage and performance considerations in mind.
- Do not assume that your collection is always going to be small: As it can potentially grow bigger with time. So your collection should scale well.
- Program to interface not implementation: For example, you might decide that LinkedList is the best choice for some application, but then later decide ArrayList might be a better choice for performance reason.

  Bad:

  ```
  ArrayList list = new ArrayList(110);
  ```

  Good:

  ```
  // program to interface so that you can change the implementation in future if
      required
  List list1 = new ArrayList(110);
  List list2 = new LinkedList(110);
  ```

- Return zero length collections or arrays instead of returning a null: in the context of the fetched list is actually empty. Returning a null value instead of a zero length

collection is more error prone, since the programmer writing the calling method might forget to handle a return value of null.

```
List emptyList = Collections.emptyList( );

Set emptySet = Collections.emptySet( );
```

- Use generics: For type safety, readability and robustness.
- Encapsulate collections: In general, collections are not immutable objects. So care should always be taken not to unintentionally expose the collection fields to the caller. The caller may not perform any necessary validation.

6.7 Take Away Points

- The core collection interfaces are the foundation of the Java Collections Framework.
- The Java Collections Framework hierarchy consists of two distinct interface trees: The first tree starts with the Collection interface, which provides for the basic functionality used by all collections, such as add and remove methods. Its subinterfaces — Set, List, and Queue — provide for more specialized collections. The second tree starts with Map interface.
- The Set interface does not allow duplicate elements. This can be useful for storing collections such as a deck of cards or student records. The Set interface has a subinterface, SortedSet, that provides for ordering of elements in the set.
- The List interface provides for an ordered collection, for situations in which you need precise control over where each element is inserted. You can retrieve elements from a List by their exact position.
- Implementations are the data objects(classes) used to store collections, which implement the interfaces in the Collection Framework.
- The Java Collections Framework also provides several special-purpose implementations for situations that require nonstandard performance, usage restrictions, or other unusual behavior.
- The Collections class (as opposed to the Collection interface), provides static methods that operate on or return collections, which are known as Wrapper implementations or polymorphic algorithms.

6.8 Quiz

6.9 Practice Problems

6.11 References and Bibliography

1. http://docs.oracle.com/javase/tutorial/collections/index.html
2. http://www.tutorialspoint.com/java/java_using_iterator.htm
3. http://www.tutorialspoint.com/java/java_collections.htm
4. http://thecafetechno.com/tutorials/interview-questions/what-are-the-best-practices-related-to-java-collection-framework/