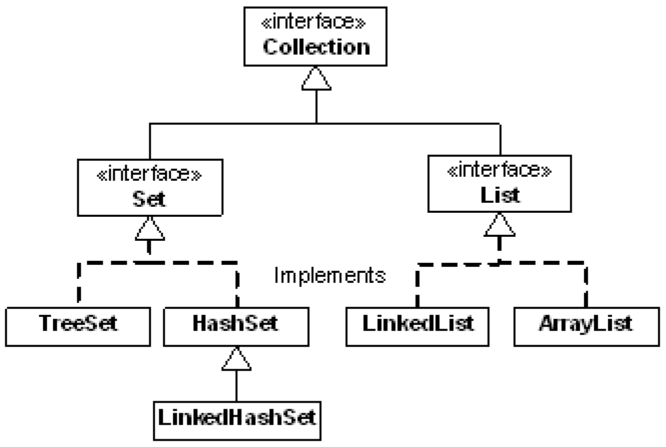
Containers

Some programs create too many objects and deal with them. In such a program, it is not feasible to declare a separate variable to hold reference to each of these objects. The proper way of keeping references may be to use arrays. However, arrays has a fixed size after they have created. So, if the number of objects that will be created is unknown at the array creation time, you should provide extra code to enlarge the array whenever new objects are created. This may involve creating a larger array and copying the elements to this array.
There is a better way of holding references to objects in the Java platform: using The Collections Framework. The collections framework in the Java platform includes high-performance implementations of useful data structures and algorithms to store and manipulate a group of objects. This increases the performance of your programs. In addition, using a common framework reduces the programming effort and fosters software reuse.
The collections framework consists of interfaces forming the base of the framework and general purpose implementations of those interfaces. There are three main container interfaces: `List`, `Set`, and `Map`. The `List` and `Set` interfaces are derived from a common interface `Collection`. A `Collection` deals with a group of individual elements and a `Map` deals with a key-value pairs of objects. The framework provides two or three implementations, which are concrete classes, for each of these interfaces. You choose one of these implementations according to your needs (especially performance considerations play an important role in choosing the proper implementation) to create a container.
Since containers only keep instances of subclasses of `Object`, in order to store primitive data types, corresponding wrapper classes of primitive data types may be used. The `toString()` method of containers may be used to list elements in the container. `toString()` method of a container makes use of `toString()` methods of individual objects in the container to print elements into the result.

Collections

Collections store and manipulate a group of individual objects. The following diagram shows the interfaces and implementations of various types of collections. For the sake of simplicity only concrete classes and key interfaces are shown in this diagram.



There are two kinds of collections: `Set` and `List`. `Set` and `List` are different in that a `List` can contain duplicate elements whereas a `Set` cannot contain duplicate elements. Moreover, insertion order is important in `List`s in contrast to `Set`s. There are three `Set` implementations `TreeSet`, `HashSet`, and `LinkedHashSet`, and two `List` implementations `LinkedList` and `ArrayList`. The names of the classes are of the form <implementation><interface> where implementation indicates the underlying data structure used the hold elements and interface indicates the type of collection. Meaning and significance of implementation types will be discussed later.
As shown in the diagram, all classes here implement the `Collection` interface. The `Collection` interface provides following methods:

Method	Description
`boolean add(Object o)`	Adds object o to the collection and return true. If the collection already contains o, does nothing and returns false.
`boolean addAll(Collection c)`	Adds all objects in collection c to this collection. If any object is added it returns true.
`void clear()`	Removes all elements from the collection
`boolean equals(Object o)`	Compares the current collection with the specified object o. If object o is implements the same interface (Set or List), compares two collections for equality. Note that, in List comparisons insertion order is important and in Set comparisons, both collections must have the objects equal to each other.
`Iterator iterator()`	Returns an iterator over the objects in the collection (Iterators will be discussed in the next section).
`boolean contains(Object o)`	Returns true if object o is contained in the collection.
`boolean containsAll(Collection c)`	Returns true if all objects in collection c is in the current collection.
`boolean isEmpty()`	Returns true if the collection has no objects.
`boolean remove(Object o)`	Removes the specified object from the collection. Returns false if the collection does not have that object.
`boolean removeAll(Collection c)`	Removes all objects in collection c from the current collection. If any object is removed, it returns true.
`boolean retainAll(Collection c)`	Removes all objects in the current collection which are not in the collection c. Returns true if any object is removed.
`int size()`	Returns the number of objects in the current collection
`Object[] toArray()`	Returns an array containing all objects in the current collection.
`Object[] toArray(Object[] a)`	Returns an array of the same type with a.

`Set` interface adds nothing to `Collection` interface. However, `List` interface defines and overloads some methods which allow indexed access to objects. In the following `index` varies between `0` and `size()-1`.

Method	Description
`boolean add(Object o)` `void add(int index, Object o)`	Adds specified object to the list at the specified index. If the index is not specified, adds object to the end of the list.
`boolean addAll(Collection c)` `void addAll(int index, Collection c)`	Inserts all objects in the specified list to the current list starting from the specified index. If the index is not specified, objects are appended to the list.
`Object get(int index)`	Returns the object at the specified index.
`Object set(int index, Object o)`	Returns the object at the specified index, and replaces it with the specified object.
`int indexOf(Object o)` `int lastIndexOf(Object o)`	Returns the index of the first/last occurrence of the specified object in the list. Returns -1 if the object o is not contained in the list.
`ListIterator listIterator()` `ListIterator listIterator(int index)`	Returns a list iterator over the objects in the list starting from the position index if specified. (Iterators will be discussed in the next section).
`List subList(int fromIndex, int toIndex)`	Returns a portion list having objects from the current list at positions fromIndex..toIndex-1. Note that any operations made to returned list is also made in the current list. This is useful for carrying out bulk operations on the list (e.g. remove all elements between the specified indexes).

Iterators

An `Iterator` is an object which enables you to move through a sequence regardless of the underlying implementation of the collection. By this way, you can access individual objects contained in the collection. All `List` and `Set` implementations provide a method to get an iterator. By using this iterator, objects may be accessed one by one calling the `next()` method and if desired current object may be removed by calling the `remove()` method of the `Iterator`. The `Iterator` also provide `hasNext()` method to check whether the end of list is reached.
`List` implementations also return more powerful `ListIterator`s to navigate within the `List` back and forth. The `ListIterator` interface defines the methods `add()`, `hasPrevious()`, `previous()`, `nextIndex()`, `previousIndex()`, and `set(Object o)` in addition to `next()`, `hasNext()`, and `remove()`.

Maps

A `Map` is a container which maps keys to values. A `Map` cannot contain duplicate key objects, and each key object maps onto only one value object. All `Map` implementations implement the `Map` interface and there are three mostly used `Map` implementations: `TreeMap`, `HashMap`, and `LinkedHashMap`.
The `Map` interface defines the following methods:

Method	Description
`void clear()`	Removes all key-value pairs from the map.
`boolean containsKey(Object key)`	Returns true if the specified key is in the map.
`boolean containsValue(Object value)`	Returns true if the specified value is in the map.
`Set entrySet()`	Returns the set of entries (key-value pairs) within the map. Each element in the set is an object of Map.Entry type, and getValue() and getKey() methods of Map.Entry may be used to access key and value of each entry.
`boolean equals(Object o)`	Compares the current map with the specified object o. If object o is implements the map interface, it compares two maps for equality. For equality, both maps must have the same key-value pairs.
`Object get(Object key)`	Returns the value mapped to the key.
`boolean isEmpty()`	Returns true if the map contains no entries.
`Set keySet()`	Returns the Set of keys in the map. Changes to Set are reflected to the map and vice versa.
`Object put(Object key, Object value)`	Adds specified key-value pair to the map. If the key is already in the map, the value mapped to key is replaced by the new one, and old one is returned.
`void putAll(Map m)`	Adds all mappings in the specified map m to this map.
`Object remove(Object key)`	Removes the entry corresponding to the specified key, and returns the mapped value.
`int size()`	Returns the number of mappings in the current map.
`Object[] toArray()`	Returns an array containing all objects in the current collection.
`Collection values()`	Returns a collection of values in the map. Changes to collection are reflected to map and vice versa.

As the table shows, individual mappings within the `Map` may be accessed using the `entrySet()` method which returns a `Set`, keys and values may be accessed using the `keys()` and `values()` which return collections. Since the returned `Collection` is backed by the `Map`, the operations carried out on the returned `Collection` are also reflected to the `Map` and vice versa. Moreover, it is possible to get an `Iterator` over mappings, keys or values by using the `iterator()` method of the returned `Collection` interface.

IMPLEMENTATIONS

There are five categories of implementations for collections and

maps: Array, linked list, hash, linked hash and tree. The implementation method reflects the underlying data structure used to hold entries. Each implementation category has its own pros and cons. Choosing the right implementation can considerably improve the performance of your applications.
Array implementations use an array for storing entries. `ArrayList` is the mostly preferred implementation for `List` interface because it offers constant time positional access. However, insertions to a specific index and deletions may be costly since creating a new array and/or copying existing array elements may be needed for insertions and deletions.
`LinkedList` is another option to hold sequence of elements. The `LinkedList` implementation use a `Entry` object to hold individual objects and references to next and previous entries. Therefore, adding elements to the beginning, and adding and removing elements while iterating is much faster since only next and previous fields of next and previous entries needs to be modified. If your program frequently adds elements, remove elements, and iterate over the elements you should consider `LinkedList` implementation. But, it should be kept in mind that, positional access is very costly in linked list implementations.
`Set` and `Map` are implemented by hash tables, trees or linked hash tables. Tree implementations use red-black trees to keep entries sorted within the data structure. If it is important to keep entries ordered, `TreeSet` and `TreeMap` implementations should be preferred. Tree implementations may use `Comparable` interface provided by the classes of objects or a `Comparator` interface may be provided through constructors of `TreeSet` and `TreeMap` to order objects in the container. Wrapper classes for primitive data types, `String` and `Date` classes implement the `Comparable` interface. However, if you use your own classes your class must implement the `Comparable` interface by implementing `compareTo()` method as:

```
class MyClass implements Comparable {
    ...
    public int compareTo(Object o){
        ...
        // returns -1 if this object is less than o
        // returns 0 if this object is equal to o
        // returns 1 if this object is greater than o
    }
    ...
}
```

While declaring your classes implementing comparable interface, you should make sure that the fields used in the comparison are immutable (do not change after creation). Otherwise, `Map` or `Set` may break as the objects’ state change after they are put into the container.
If your objects do not implement `Comparable` interface you may create a class implementing `Comparator` interface as:

```
class MyComp implements Comparator{
    int compare(Object o1, Object o2){
        ...
        // returns -1 if o1 is less than o2
        // returns 0 if o2 is equal to o2
        // returns 1 if o1 is greater than o2
    }
}
```
Then you can create an instance of this class and pass it to the `Set` or `Map` constructor as:

```
TreeSet s = new TreeSet(new MyComp());
```
`TreeSet` and `TreeMap` also implements the `SortedSet` and `SortedMap` interfaces, respectively. These interfaces define useful functions to access individual element (or set of elements with a given criterion) in the `Set` or `Map` quickly. For more information about `sortedSet` and `SortedMap` interfaces refer to the Java documentation.
Hash based implementations uses `hashCode()` function, which return an integer value, provided by the objects to locate objects within the underlying data structure. Therefore, `HashSet` and `HashMap` is much faster than the `TreeSet` and `TreeMap`. However, ordering objects in the data structure is arbitrary, i.e., order is not preserved. If entry ordering is not important and performance is the main concern, hash implementations should be preferred. Linked hash based implementations preserve the order of insertion while using hashes for better performance.

Collections Class

`Collections` class provides many useful methods that can operate on collections. Since the methods are declared as static you do not need to create an instance of `Collections` class to make use of its services. Most of the methods operate on `List`s such as `sort` (to sort objects within a `List`), `binarySearch` (searching an object within a `List` which is sorted before in ascending order), `shuffle` (randomly permute objects within the `List`), `copy` (copy objects from one `List` to another), `reverse` (reverse the ordering within a `List`), `rotate` (rotate the order of objects within a `List` by the specified distance), etc.
There are also methods to return a synchronized (thread-safe), and immutable versions of containers of different types.
For further information about the `Collections` class please refer to the Java documentation.