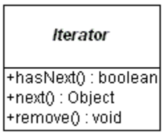
and will be described in Enumeration interface on page 26. With the `Iterator` interface methods, you can traverse a collection from start to finish and safely remove elements from the underlying `Collection`:



The *`remove()`* method is optionally supported by the underlying collection. When called, and supported, the element returned by the last *`next()`* call is removed. To demonstrate, the following code shows the use of the `Iterator` interface for a general `Collection`:

```
Collection collection = ...;
 Iterator iterator = collection.iterator();
 while (iterator.hasNext()) {
   Object element = iterator.next();
   if (removalCheck(element)) {
     iterator.remove();
   }
 }
```

Group operations

Other operations the `Collection` interface supports are tasks done on groups of elements or the entire collection at once:

* `boolean containsAll(Collection collection)`
* `boolean addAll(Collection collection)`
* `void clear()`
* `void removeAll(Collection collection)`
* `void retainAll(Collection collection)`

The *`containsAll()`* method allows you to discover if the current collection contains all the elements of another collection, a *subset*. The remaining methods are optional, in that a specific collection might not support the altering of the collection. The *`addAll()`* method ensures all elements from another collection are added to the current collection, usually a *union*. The *`clear()`* method removes all elements from the current collection. The *`removeAll()`* method is like *`clear()`* but only removes a subset of elements. The *`retainAll()`* method is similar to the *`removeAll()`* method but does what might be perceived as the opposite: it removes from the current collection those elements not in the other collection, an *intersection*.

The remaining two interface methods, which convert a `Collection` to an array, will be discussed in Converting from new collections to historical collections on page 32.

AbstractCollection class

The `AbstractCollection` class provides the basis for the concrete collections framework classes. While you are free to implement all the methods of the `Collection` interface yourself, the `AbstractCollection` class provides implementations for all the methods, except for the `iterator()` and `size()` methods, which are implemented in the appropriate subclass. Optional methods like *`add()`* will throw an exception if the subclass doesn't override the behavior.

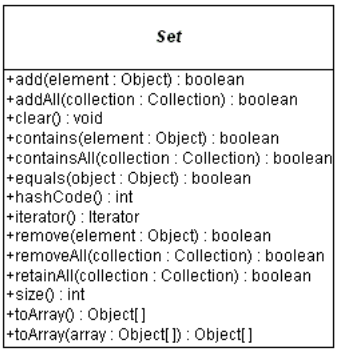
Collections Framework design concerns

In the creation of the Collections Framework, the Sun development team needed to provide flexible interfaces that manipulated groups of elements. To keep the design simple, instead of providing separate interfaces for optional capabilities, the interfaces define all the methods an implementation class may provide. However, *some* of the interface methods are optional. Because an interface implementation must provide implementations for all the interface methods, there needed to be a way for a caller to know if an optional method is not supported. The manner the framework development team chose to signal callers when an optional method is called was to throw an `UnsupportedOperationException`. If in the course of using a collection an `UnsupportedOperationException` is thrown, then the operation failed because it is not supported. To avoid having to place all collection operations within a `try-catch` block, the `UnsupportedOperationException` class is an extension of the `RuntimeException` class.

In addition to handling optional operations with a run-time exception, the iterators for the concrete collection implementations are *fail-fast* . That means that if you are using an `Iterator` to traverse a collection while the underlying collection is being modified by another thread, then the `Iterator` fails immediately by throwing a `ConcurrentModificationException` (another `RuntimeException`). That means the next time an `Iterator` method is called, and the underlying collection has been modified, the `ConcurrentModificationException` exception gets thrown.

Set interface

The `Set` interface extends the `Collection` interface and, by definition, forbids duplicates within the collection. All the original methods are present and no new methods are introduced. The concrete `Set` implementation classes rely on the *`equals()`* method of the object added to check for equality.



HashSet and TreeSet classes

The Collections Framework provides two general-purpose implementations of the `Set` interface: `HashSet` and `TreeSet`. More often than not, you will use a `HashSet` for storing your duplicate-free collection. For efficiency, objects added to a `HashSet` need to implement the `hashCode()` method in a manner that properly distributes the hash codes. While most system classes override the default *`hashCode()`* implementation in `Object`, when creating your own classes to add to a HashSet remember to override hashCode(). The TreeSet implementation is useful when you need to extract elements from a collection in a sorted manner. In order to work properly, elements added to a TreeSet must be sortable. The Collections Framework adds support for Comparable elements and will be covered in detail in "Comparable interface" in Sorting on page 17. For now, just assume a tree knows how to keep elements of the java.lang wrapper classes sorted. It is generally faster to add elements to a HashSet, then convert the collection to a TreeSet for sorted traversal.

To optimize HashSet space usage, you can tune the initial capacity and load factor. The TreeSet has no tuning options, as the tree is always balanced, ensuring log(n) performance for insertions, deletions, and queries.

Both HashSet and TreeSet implement the Cloneable interface.

Set usage example

To demonstrate the use of the concrete Set classes, the following program creates a HashSet and adds a group of names, including one name twice. The program then prints out the list of names in the set, demonstrating the duplicate name isn't present. Next, the program treats the set as a TreeSet and displays the list sorted.

```
import java.util.*;

 public class SetExample {
   public static void main(String args[]) {
     Set set = new HashSet();
     set.add("Bernadine");
     set.add("Elizabeth");
     set.add("Gene");
     set.add("Elizabeth");
     set.add("Clara");
     System.out.println(set);
     Set sortedSet = new TreeSet(set);
     System.out.println(sortedSet);
   }
 }
```

Running the program produces the following output. Notice that the duplicate entry is only present once, and the second list output is sorted alphabetically.

```
[Gene, Clara, Bernadine, Elizabeth]
 [Bernadine, Clara, Elizabeth, Gene]
```

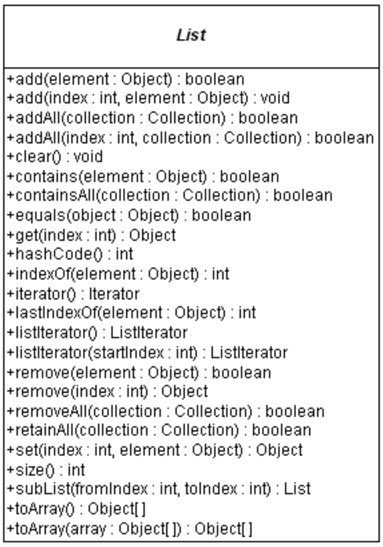
AbstractSet class

The AbstractSet class overrides the *equals()* and *hashCode()* methods to ensure two equal sets return the same hash code. Two sets are equal if they are the same size and contain the same elements. By definition, the hash code for a set is the sum of the hash codes for the elements of the set. Thus, no matter what the internal ordering of the sets, two equal sets will report the same hash code.

Exercises

List interface

The `List` interface extends the `Collection` interface to define an ordered collection, permitting duplicates. The interface adds position-oriented operations, as well as the ability to work with just a part of the list.



The position-oriented operations include the ability to insert an element or `Collection`, get an element, as well as remove or change an element. Searching for an element in a `List` can be started from the beginning or end and will report the position of the element, if found.

* `void add(int index, Object element)`
* `boolean addAll(int index, Collection collection)`
* `Object get(int index)`
* `int indexOf(Object element)`
* `int lastIndexOf(Object element)`
* `Object remove(int index)`
* `Object set(int index, Object element)`

The `List` interface also provides for working with a subset of the collection, as well as iterating through the entire list in a position-friendly manner:

* `ListIterator listIterator()`
* `ListIterator listIterator(int startIndex)`
* `List subList(int fromIndex, int toIndex)`

In working with *`subList()`* , it is important to mention that the element at `fromIndex` is in the sublist, but the element at `toIndex` is not. This loosely maps to the following `for-loop` test cases:

```
for (int i=fromIndex; i<toIndex; i++) {
```