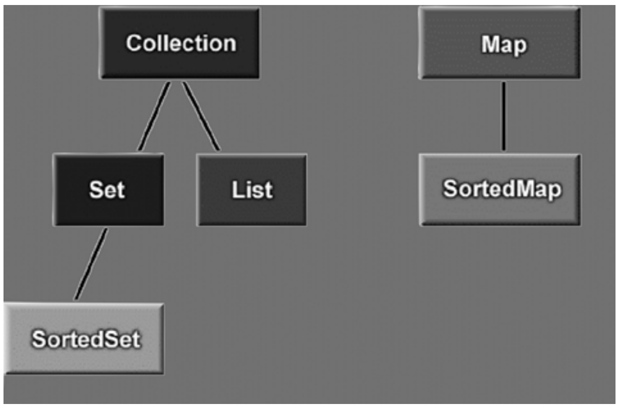
What Is a Collection?

- A collection (sometimes called a container) is simply an object that groups multiple elements into a single unit
- Collections typically represent data items that form a natural group, like a poker hand (a collection of cards), a mail folder (a collection of letters), or a telephone directory (a collection of name-to-phone-number mappings)

What are the Benefits of Collections?

- It reduces programming effort
- It increases program speed and quality
- It allows interoperability among unrelated APIs
- It reduces the effort to learn and use new APIs
- It reduces effort to design new APIs
- It fosters software reuse

Collection: Interfaces



`Collection`: Basic Operations

```
public interface Collection {
…
int size();
boolean isEmpty();
boolean contains(Object element);
boolean add(Object element);    // Optional
boolean remove(Object element); // Optional
Iterator iterator();
…
```

Collection: Bulk Operations

```
…
boolean containsAll(Collection c);
boolean addAll(Collection c);    // Optional
boolean removeAll(Collection c); // Optional
boolean retainAll(Collection c); // Optional
void clear();
…
```

Collection: Array Operations

```
…
Object[] toArray();
Object[] toArray(Object a[]);
…
```

The `Iterator` Interface

```
public interface Iterator {
    boolean hasNext();
    Object next();
    void remove();    // Optional
}
```

Using Iterators

```
static void filter(Collection c) {
    for (Iterator i = c.iterator(); i.hasNext();)
        if (!cond(i.next()))
            i.remove();
}
```

- **Note: the code is polymorphic: it works for any Collection that supports element removal, regardless of implementation. That's how easy it is to write a polymorphic algorithm under the collections framework!**

The `List` interface

A `List` is an ordered Collection(sometimes called a sequence). Lists may contain duplicate elements. In addition to the operations inherited from Collection, the List interface includes operations for:

- Positional Access: manipulate elements based on their numerical position in the list.
- Search: search for a specified object in the list and return its numerical position.
- List Iteration: extend Iterator semantics to take advantage of the list's sequential nature.
- Range-view: perform arbitrary range operations on the list.

`List`: Positional Access

```
Object get(int index);
Object set(int index, Object element);
  // Optional
void add(int index, Object element);
  // Optional
Object remove(int index);
  // Optional
abstract boolean addAll(int index,
  Collection c); // Optional
```

`List`: Search

```
int indexOf(Object o);
int lastIndexOf(Object o);
```

`List`: Iteration

```
ListIterator listIterator();
ListIterator listIterator(int index);
```

`List`: Range-view

```
List subList(int from, int to);
```

Concrete classes implementing `List`

- ArrayList
  - Implements a list using arrays
- LinkedList
  - Implements a list using linked elements
- Vector
  - Similar to `ArrayList`, used for backward compatibility

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. The `Map` interface includes operations for:

- Basic Operations
- Bulk Operations
- Collection Views
- Interface for `entrySet` elements

`Map`: Basic Operations

```
Object put(Object key, Object value);
Object get(Object key);
Object remove(Object key);
boolean containsKey(Object key);
boolean containsValue(Object value);
int size();
boolean isEmpty();
```

`Map`: Bulk Operations

```
void putAll(Map t);
void clear();
```

`Map`: Collection Views

```
public Set keySet();
public Collection values();
public Set entrySet();
```

Concrete classes implementing `Map`

- ➢ `Hashtable`
  - Implements a map using hash tables
- ➢ `TreeMap`
  - Implements a sorted map using Red-Black trees
- ➢ `HashMap`
  - Similar to `Hashtable`, used for backward compatibility

Example

```
import java.util.*;

public class Freq {
    private static final Integer ONE = new Integer(1);
    public static void main(String args[]) {
        Map m = new HashMap();
        // Initialize frequency table from command line
        for (int i=0; i<args.length; i++) {
            Integer freq = (Integer) m.get(args[i]);
            m.put(args[i], (freq==null ? ONE :
                  new Integer(freq.intValue() + 1)));
        }
        System.out.println(m.size()+" distinct words
                           detected:");
        System.out.println(m);
    }
}
```

Object Ordering

There are two ways to order objects:

- The `Comparable` interface provides automatic natural order on classes that implement it
- The `Comparator` interface gives the programmer complete control over object ordering
- `Collections.sort(List l)`
  can be used to order a list using natural order
- `Collections.sort(List l, Comparator c)`
  can be used to order a list depending on the behaviour of `c`