**Enum:**

* An *enum type* is a type whose *fields* consist of a fixed set of constants. NORTH, SOUTH, EAST, and WEST
* Because they are constants, the names of an enum type's fields are in uppercase letters.
* Define an enum type by using the enum keyword.

public enum Day {

SUNDAY, MONDAY, TUESDAY, WEDNESDAY,

THURSDAY, FRIDAY, SATURDAY

}

**Collection:**

* A *collection* — sometimes called a container — is simply an object that groups multiple elements into a single unit. Collections are used to store, retrieve, manipulate, and communicate aggregate data. Eg. Vector, Hashtable, and array.
* A *collections framework* is a unified architecture for representing and manipulating collections.
* All collections frameworks contain the following:
  + **Interfaces**
  + **Implementations**
  + **Algorithms**

**Benefits of the Java Collections Framework**

* Reduces programming effort
* Increases program speed and quality
* Allows interoperability among unrelated APIs
* Reduces effort to learn and to use new APIs
* Reduces effort to design new APIs
* Fosters software reuse
* The *core collection interfaces* encapsulate different types of collections
* These interfaces allow collections to be manipulated independently of the details of their representation.
* Core collection interfaces are the foundation of the Java Collections Framework

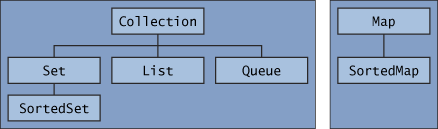


Fig. Source: Oracle Java Tutorial

* A Set is a special kind of Collection, a SortedSet is a special kind of Set, and so forth. Note also that the hierarchy consists of two distinct trees — a Map is not a true Collection.
* **Collection** — the root of the collection hierarchy.
  + A collection represents a group of objects known as its *elements*.
  + The Collection interface is the least common denominator that all collections implement and is used to pass collections around and to manipulate them when maximum generality is desired.
  + Some types of collections allow duplicate elements, and others do not.
  + Some are ordered and others are unordered.
  + 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 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.
  + 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 newArrayList (an implementation of the List interface), initially containing all the elements in c.

List<String> list = new ArrayList<String>(c);

public interface Collection<E> extends Iterable<E> {

// Basic operations

int size();

boolean isEmpty();

boolean contains(Object element);

boolean add(E element); //optional

boolean remove(Object element); //optional

Iterator<E> iterator();

// Bulk operations

boolean containsAll(Collection<?> c);

boolean addAll(Collection<? extends E> c); //optional

boolean removeAll(Collection<?> c); //optional

boolean retainAll(Collection<?> c); //optional

void clear(); //optional

// Array operations

Object[] toArray();

<T> T[] toArray(T[] a);

}

Show the Collection API interface from the java source

* + Traversing Collections
    - There are two ways to traverse collections:
      * With the for-each construct
        + The for-each construct allows you to concisely traverse a collection or array using a for loop
        + The following code uses the for-each construct to print out each element of a collection on a separate line.

for (Object o : collection)

System.out.println(o);

* + - * By using Iterators.
        + An [Iterator](http://java.sun.com/javase/6/docs/api/java/util/Iterator.html) is an object that enables you to traverse through a collection and to remove elements from the collection selectively, if desired. You get an Iterator for a collection by calling its iterator method. The following is the Iterator interface.

public interface Iterator<E> {

boolean hasNext();

E next();

void remove(); //optional

}

Collection<String> c = . . .;  
Iterator<String> iter = c.iterator();  
while (iter.hasNext())  
{   
String element = iter.next();  
do something with element  
}

* + - * + The hasNext method returns true if the iteration has more elements, and the next method returns the next element in the iteration.
        + The remove method removes the last element that was returned by next from the underlying Collection.
        + The remove method may be called only once per call to next and throws an exception if this rule is violated.
        + **Note**: Iterator.remove is the *only* safe way to modify a collection during iteration; the behavior is unspecified if the underlying collection is modified in any other way while the iteration is in progress.
        + Use Iterator instead of the for-each construct when you need to:

Remove the current element. The for-each construct hides the iterator, so you cannot call remove. Therefore, the for-each construct is not usable for filtering.

Iterate over multiple collections in parallel.

**Collection Interface Bulk Operations**

*Bulk operations* perform an operation on an entire Collection.

* containsAll — returns true if the target Collection contains all the elements in the specified Collection.
* addAll — adds all the elements in the specified Collection to the target Collection.
* removeAll — removes from the target Collection all 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.
* **Set** — a collection that cannot contain duplicate elements.
  + The Set interface contains *only* methods inherited from Collection
  + Set adds the restriction that duplicate elements are prohibited.
  + 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**](http://java.sun.com/javase/6/docs/api/java/util/HashSet.html), which stores its elements in a hash table, is the best-performing implementation; however it makes no guarantees concerning the order of iteration.
    - [**TreeSet**](http://java.sun.com/javase/6/docs/api/java/util/TreeSet.html), which stores its elements in a red-black tree, orders its elements based on their values; it is substantially slower than HashSet.
    - [**LinkedHashSet**](http://java.sun.com/javase/6/docs/api/java/util/LinkedHashSet.html), 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).
    - Suppose you have a Collection, dup, 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>(dup);
    - Create a collection that preserves the order of the original collection while removing duplicate element.

Collection<Type> noDups = new LinkedHashSet<Type>(dup);

* **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).
  + In addition to the operations inherited from Collection, the List interface includes operations for the following:
    - Positional access — manipulates elements based on their numerical position in the list
    - Search — searches for a specified object in the list and returns its numerical position
    - Iteration — extends Iterator semantics to take advantage of the list's sequential nature
    - Range-view — performs arbitrary *range operations* on the list.
    - The Java platform contains two general purpose List implementations.
      * [**ArrayList**](http://java.sun.com/javase/6/docs/api/java/util/ArrayList.html), which is usually the better-performing implementation, and
      * [**LinkedList**](http://java.sun.com/javase/6/docs/api/java/util/LinkedList.html) which offers better performance under certain circumstances.
* **Comparison to Vector**
  + List is an interface, while Vector is a concrete implementation.
  + Listfixes several minor API deficiencies in Vector.
* List Algorithms
* sort — sorts a List using a merge sort algorithm, which provides a fast, stable sort. (A *stable sort* is one that does not reorder equal elements.)
* shuffle — randomly permutes the elements in a List.
* reverse — reverses the order of the elements in a List.
* rotate — rotates all the elements in a List by a specified distance.
* swap — swaps the elements at specified positions in a List.
* replaceAll — replaces all occurrences of one specified value with another.
* fill — overwrites every element in a List with the specified value.
* copy — copies the source List into the destination List.
* binarySearch — searches for an element in an ordered List using the binary search algorithm.
* indexOfSubList — returns the index of the first sublist of one List that is equal to another.
* lastIndexOfSubList — returns the index of the last sublist of one List that is equal to another.
* **Queue** — a collection used to hold multiple elements prior to processing.
  + Besides basic **Collection** operations, a Queue provides additional insertion, extraction, and inspection operations.
  + Queues typically, but do not necessarily, order elements in a FIFO (first-in-first-out) manner

public interface Queue<E> extends Collection<E> {

E element();

boolean offer(E e);

E peek();

E poll();

E remove();

}

* + Each Queue method exists in two forms:
    - Throws an exception if the operation fails
    - Returns a special value if the operation fails (either null or false, depending on the operation).
    - The regular structure of the interface is illustrated in the following table.

|  |  |  |
| --- | --- | --- |
| **Queue Interface Structure** | | |
|  | **Throws exception** | **Returns special value** |
| **Insert** | add(e) | offer(e) |
| **Remove** | remove() | poll() |
| **Examine** | element() | peek() |

Fig:Source - Oracle Java Tutorial

* + It is possible for a Queue implementation to restrict the number of elements that it holds; such queues are known as *bounded*. Some Queue implementations injava.util.concurrent are bounded, but the implementations in java.util are not.
  + The add method, which Queue inherits from Collection, inserts an element unless it would violate the queue's capacity restrictions, in which case it throwsIllegalStateException. The offer method, which is intended solely for use on bounded queues, differs from add only in that it indicates failure to insert an element by returning false.
  + The remove and poll methods both remove and return the head of the queue. Exactly which element gets removed is a function of the queue's ordering policy. Theremove and poll methods differ in their behavior only when the queue is empty. Under these circumstances, remove throws NoSuchElementException, whilepoll returns null.
  + The element and peek methods return, but do not remove, the head of the queue. They differ from one another in precisely the same fashion as remove and poll: If the queue is empty,  element throws NoSuchElementException, while peek returns null.
  + Queue implementations generally do not allow insertion of null elements. The LinkedList implementation, which was retrofitted to implement Queue, is an exception. For historical reasons, it permits null elements, but you should refrain from taking advantage of this, because null is used as a special return value by thepoll and peek methods.

Queue<E> queue = new PriorityQueue<E>(c);

Queue<Integer> queue = new LinkedList<Integer>();

* **Map**
  + A [Map](http://java.sun.com/javase/6/docs/api/java/util/Map.html) is an object that maps keys to values.
  + A map cannot contain duplicate keys: each key can map to at most one value.
  + The Java platform contains three general-purpose Map implementations:
    - [HashMap](http://java.sun.com/javase/6/docs/api/java/util/HashMap.html),
    - [TreeMap](http://java.sun.com/javase/6/docs/api/java/util/TreeMap.html), and
    - [LinkedHashMap](http://java.sun.com/javase/6/docs/api/java/util/LinkedHashMap.html).
  + Their behavior and performance are precisely analogous to HashSet, TreeSet, and LinkedHashSet
* **Map Vs HashTable**
  + Map is an interface, while Hashtable is a concrete implementation
  + Map provides Collection views instead of direct support for iteration via Enumeration objects. Collection views greatly enhance the expressiveness of the interface, as discussed later in this section.
  + Map allows you to iterate over keys, values, or key-value pairs; Hashtable does not provide the third option.
  + Map provides a safe way to remove entries in the midst of iteration; Hashtable did not.
  + The basic operations of Map:
    - put, get, containsKey, containsValue, size, and isEmpty

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);

}

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 has the effect of setting the frequency to one if the word has never been seen before or one more than its current value if the word has already been seen. Try running this program with the command:

java Freq if it is to be it is up to me to delegate

The program yields the following output.

8 distinct words:

{to=3, delegate=1, be=1, it=2, up=1, if=1, me=1, is=2}

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

* 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.