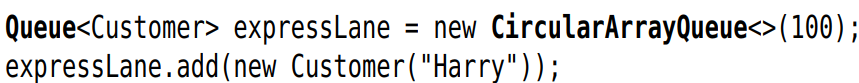
Chapter 9: Collections

# 9.1 The Java Collections Framework

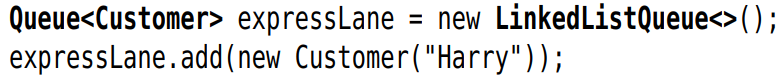
## 9.1.1 Separating Collection Interfaces and Implementation

-The Java collection library separates **interfaces** and **implementations**. Each implementation can be expressed by a class that implements the interface of a **data structure** of collection. Example: Data structure Queue have Queue interface and 2 implementations ArrayDeque and LinkedList.

-When using a queue, you don’t need to know which implementation is actually used once the collection has been constructed -> Use the concrete class only when constructing the collection object and use the **interface type** to hold the collection reference.



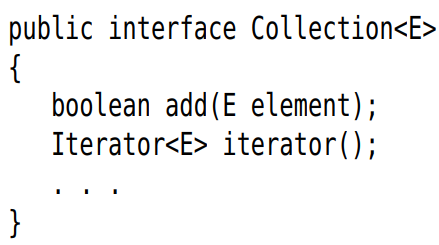
+Hence you can use a different implementation, change in the constructor call:



-In API doc, it has classes whose name begins with **Abstract**, You can extend them to implement the own collection class than making a new class,

## 9.1.2 The Collection Interface

-The fundamental interface for collection classes in Java library is **Collection** interface. It has 2 fundamental methods:

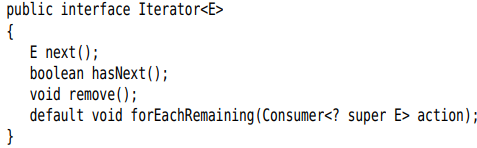


+add() adds an element to the collection

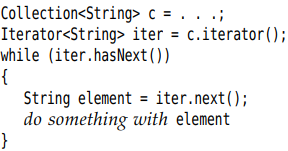
+iterator() return an object that implements Iterator interface. You can use iterator object to visit the elements in the collection one by one.

## 9.1.3 Iterators

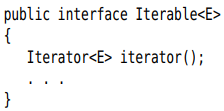
-The Iterator interface has 4 methods:



-Inspect all elements in a collection:



-The compiler translates the for each loop into a loop with an iterator. It works with any object that implements Iterable interface:

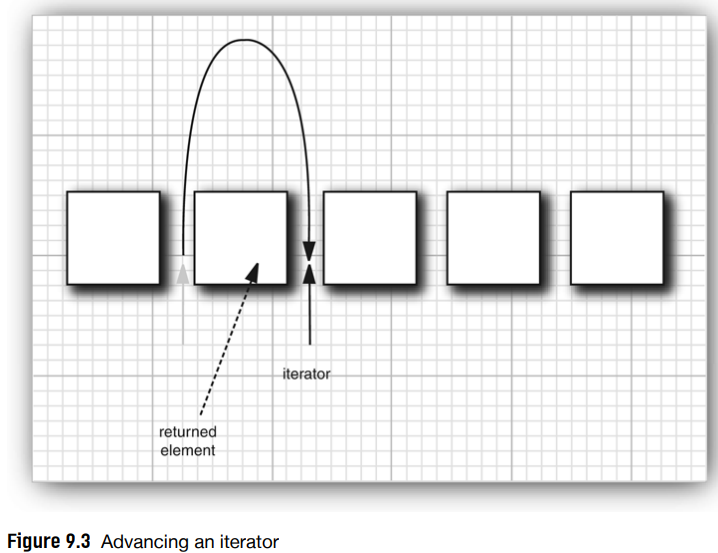


+The Collection interface extends Iterable interface -> Use for each with any collection in standard library.

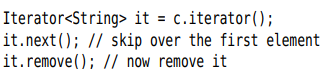
+forEachRemaining() with lambda:



-The order depends on the collection type.

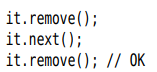
-Java iterators as being between elements. When you call next(), iterator jumps over the next element, returns a reference to the element. 

-remove(): if you remove an element, you need to skip past the element.



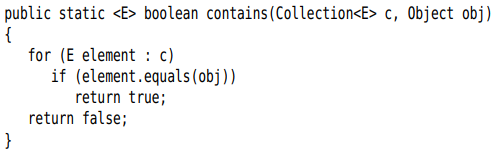
+next() + remove()

+Remove 2 adjacent elements:

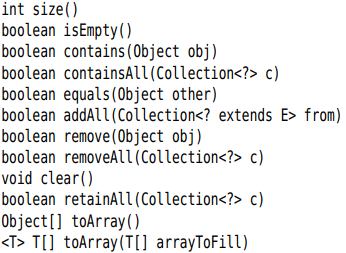


## 9.1.4 Generic Utility Methods

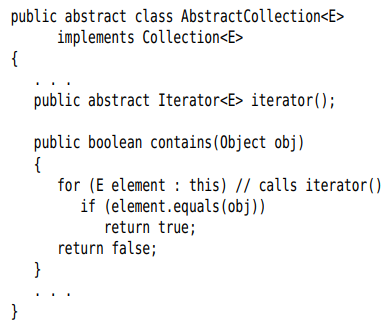
-Collection and Iterator interface are generic, you can write utility methods that operate on any kind of collection.



-The Collection interface declares useful methods that all implementing classes must supply:



-**AbstractCollection** class leaves size() and interator() abstract but implement the routine methods. A Collection class can extend this class:



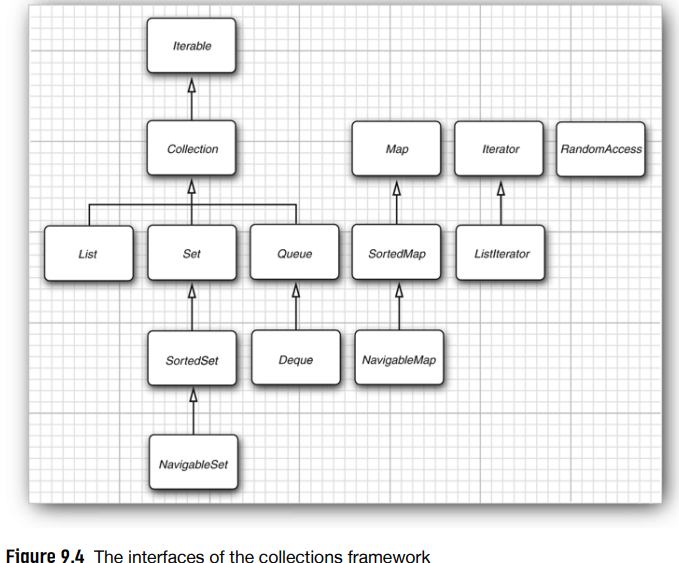
+This approach is a bit outdated. Now, Several default methods have been added and most of them deal with streams. A useful method removeIf():



-APIs Collection: p490

# 9.2 Interfaces in the Collections Framework

-Java collection framework defines interfaces for different types of collections.

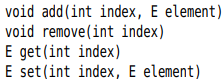


+There are 2 fundamental collections interfaces: **Collection** and **Map**.

+You insert elements with **add()** and **put()** in Map.

+Read elements with an iterator or read from a map with get().

-**List** is an **ordered collection**. Element can accessed by an iterator or by an integer index (**random access**). This interface defines methods for random access:

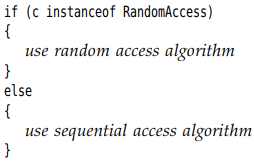


+**ListIterator** defines a method for adding an element:



+There are 2 kinds of ordered collections: List with fast random access and linked list with slow random access

+**Note**: Test a particular collection support efficient random access, use **RandomAccess**:



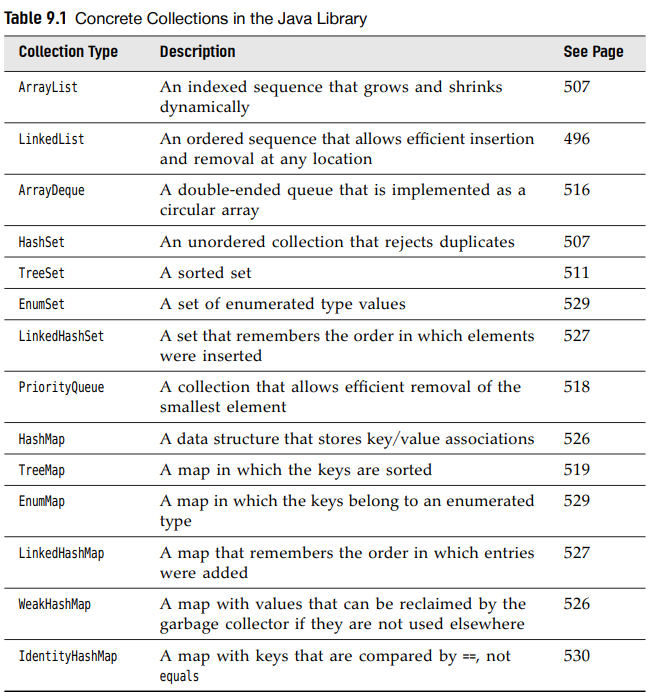
-**Set** interface is identical to the Collection interface, but methods is tightly defined: add() rejects duplicates, equal(), hasCode()…

+**SortedSet** and **SortedMap** interfaces expose comparator object for sorting, methods to obtain views of subsets.

+**NavigableSet** and **NavigableMap** contain methods for searching and traversal in sorted set and maps. They are implemented by **TreeSet** and **TreeMap**.

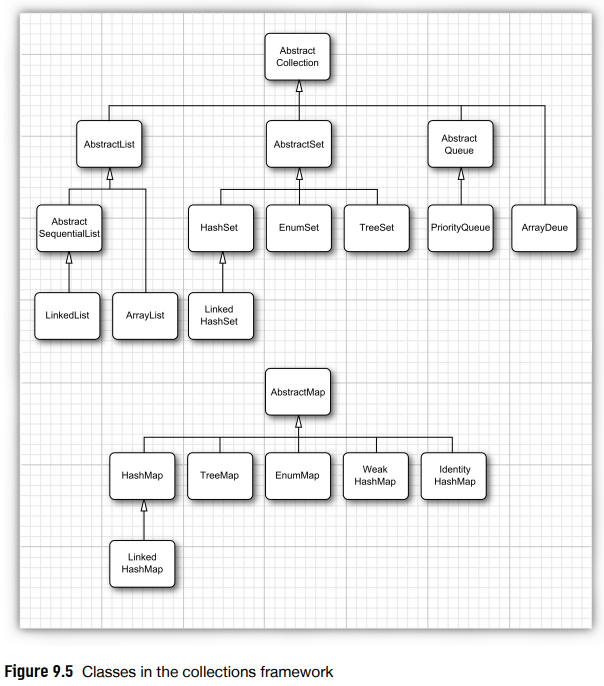
# 9.3 Concrete Collections

-This table shows the collections in Java library:



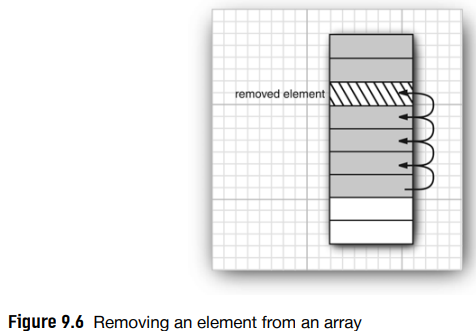
+Those classes implement Collection interface with exception of classes with Map.

-The relationship between these classes:

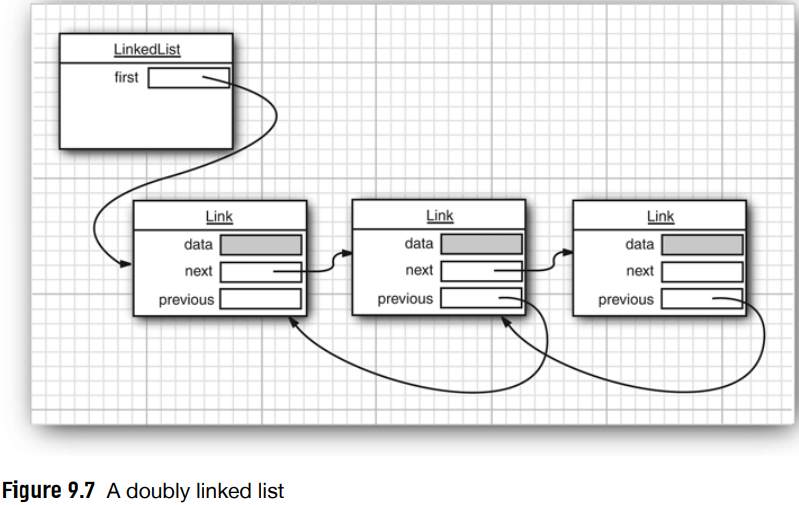


## 9.3.1 Linked Lists

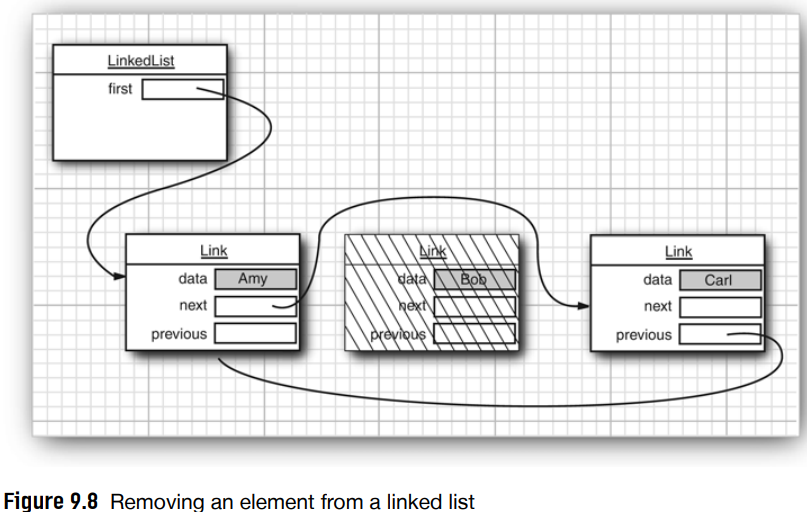
-Remove an element from the middle of an array is expensive (or insert elements in the middle).



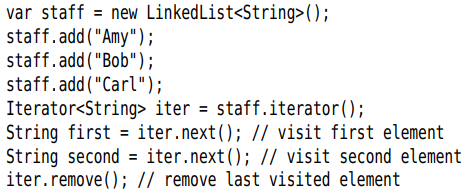
-**Linked list** stores each object in a separate **link**. Each link stores a reference to the next link. In Java, all linked lists are **doubly linked**:



-Remove an element in the middle of linked list:

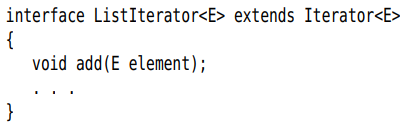


-Add elements and remove:



+This is an important different between linked list and generic collection. Linked list is an ordered collection, add() adds the object to the end of the list.

-Collection library supplies a subinterface **ListIterator** that contains an add():

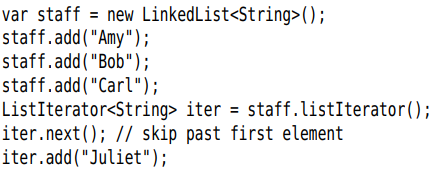


+ListIterator interface has 2 methods:



+listIterator():

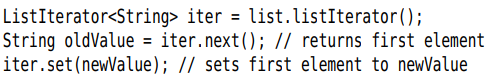
+add() adds element before the iterator position:



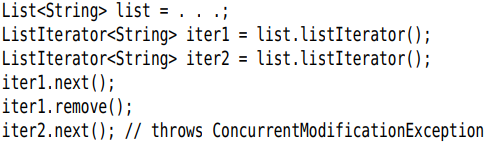
+If the linked list has n elements, there are n+1 positions of the iterator.



+set():



+If an iterator traverses a collection while another iterator is modifying it -> Exception:



+To avoid this: you can attach as many iterators which are only readers. Or you can attach a single iterator that can both read and write.

+Note: Linked list only keeps track of **structural modifications** (add, remove). So you can attach iterators which call set()

+Tell the index of the current position: nextIndex() and previousIndex(). List.listIterator(n) returns an iterator that points just before the element with index n.

-Many other useful methods on linked list are declared in Collection interface and implemented in AbstractCollection superclass.

-Linked list don’t support fast random access. But LinkedList class supplies get() that lets you access a particular element:



+But never use get() to step through linked list:



-The only reason to use linked list is to minimize the cost of insertion and removal in the middle of the list. Stay away from methods that use integer index to denote position. If you want random access, use ArrayList.

-APIs: p505

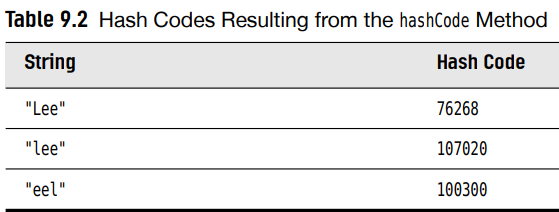
## 9.3.2 Array Lists

-List interface describes an ordered collection which has 2 protocols for visiting elements: iterator and random access with get() and set(). **ArrayList** class that also implements List interface encapsulates a dynamically reallocated array of objects.

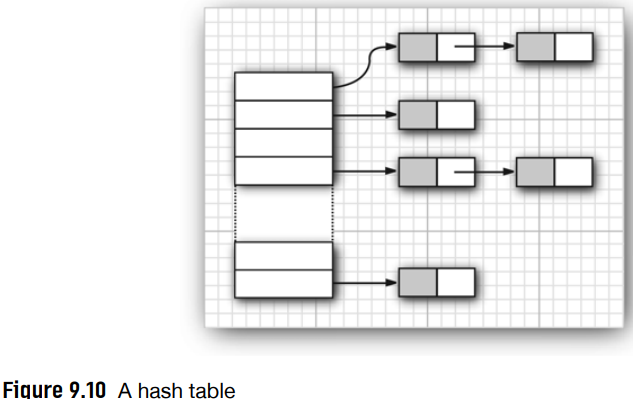
## 9.3.3 Hash Sets

-If you don’t care about the ordering, there are data structures that let you find elements much faster. The drawback is that those data structures give you no control over the order in which the elements appear.

-A well-known data structure for finding objects quickly is **hash table**. It computes an integer **hash code**. If a.equals(b), a and b must have the same hash code.



-In Java, hash tables are implemented as arrays of linked lists. Each list is a **bucket**



-The bucket count gives the number of buckets used to collect object with identical hash values.

-If the hash table gets too full, it needs to be **rehashed**. The **load factor** determines when a hash table is rehashed. For most apps, It’s reasonable to leave the load factor at 0.75

-Hash tables can be used to implement important data structures. The simplest among them is **set** type. A set is a collection of elements without duplicates. Add() tries to find the object to be added and adds it only if it is not yet present.

-Java collections library supplies HashSet class that implements a set based on a hash table.

+contains() check only the elements in one bucket and not all elements in collection.

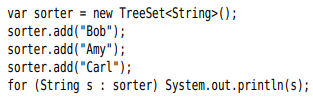
+Hash set iterator visits all buckets. Elements are visited in random order. You use HashSet if you don’t care about the ordering of the elements.

+Note: Be careful when you mutate set elements. If the hash code of an element were to change, the element would no longer be in the correct position.

-APIs: p511

## 9.3.4 Tree Sets

-**TreeSet** class is similar to hash set, but is a sorted collection.



+Adding an element to a tree is slower than adding it to a hash table but checking duplicate of tree is faster.

+The elements must implement Comparable or Comparator interface.

-Using this set depends on the data. The sort order for a tree must be a **total ordering**

-API: p515

## 9.3.5 Queues and Deques

-A queue lets you add elements at the tail and remove elements from the head. A double-ended queue, or deque, lets you add or remove elements at the head and tail. Adding in the middle is not supported.

-Deque interface: ArrayDeque and LinkedList.

-Api: p516

## 9.3.6 Priority Queues

-A priority queue retrieves elements in sorted order: remove() get the smallest element currently. But the priority queue doesn’t sort all its elements. It uses heap. You can implement Comparable and Comparator.

-A typical use for a priority queue is job scheduling.

-API: p519

# 9.4 Maps

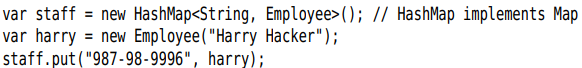
-A map stores key/value pairs. You can find a value if you provide. Example: keys are the employee IDs and the values are Employee objects.

## 9.4.1 Basic Map Operations

-Java library supplies 2 general-purpose implementations for maps: HashMap and TreeMap. Both classes implement Map interface.

-A hash map hashes the keys, a tree map uses an ordering on the keys. The function is applied **only to the keys.** Hashing is a bit faster than tree.

-Example: use a hash map for employees



+To retrieve an object, use the key:



+You can use **get(null)** or **getOrDefault()**



-Key must be unique. If you put() twice, the 2nd value replaces the 1st one.

-**remove()** removes an element with a given key. **Size()** return the number of entries.

-The easiest way of iterating over the keys and values of a map is **forEach():**



## 9.4.2 Updating Map Entries

-Normally, you get the old value associated with a key, update it, put back the updated value.

-Example: count how often a word occurs

+Use getOrDefault()



+putIfAbsent()



+merge()

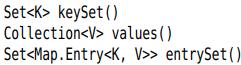


-API: p524

## 9.4.3 Map Views

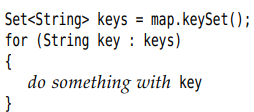
-The collection framework doesn’t consider a map as a collection. But you can obtain **views** of the map – objects that implement the Collection interface or one of its subinterfaces.

-There are 3 views: the set of keys, the collection of values (not a set), the set of key/value pair:

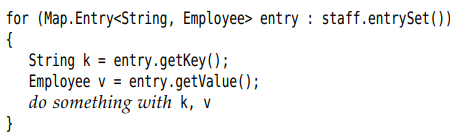


+The elements of the entry set are objects of a class implementing **Map.Entry** interface.

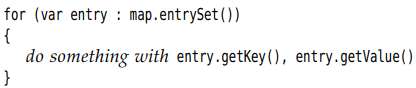
+**keyset()** is not HashSet or TreeSet but an object of some other class that implements **Set** interface. Enumerate **all keys** of a map:



+Look up **both** keys and values by enumerating the **entries**:



+Tip: You can avoid the Map.Entry by var:



Or use forEach()

-If you remove() of the iterator on the key set view, you remove the key and **its value from the map.** But you can’t add element to the key set view.

-API: p526

## 9.4.4 Weak Hash Maps

-Suppose the last reference to a key has gone away, but the key/value pair cannot be removed from the map.

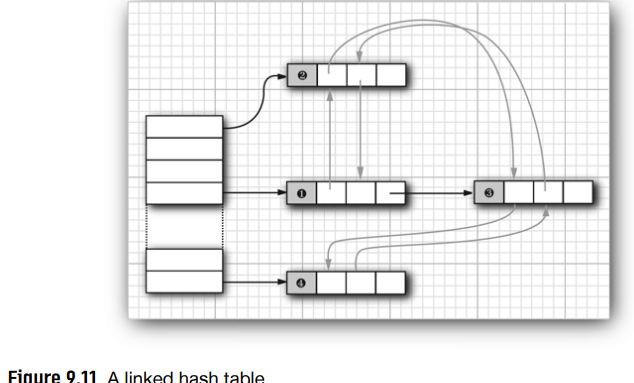
+The garbage collector traces **live** objects. As long as the map object is live, all buckets in it are live and won’t be reclaimed.

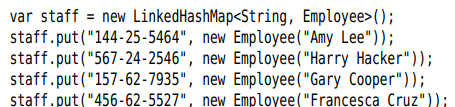
-**WeakHashMap** class co-operates with the garbage collector to remove key/value pair when the only reference to the key is the one from the hash table entry.

+WeakHashMap uses weak references to hold keys. A WeakReference object holds a reference to another object. Objects of this type are treated in a special way by garbage collector.

## 9.4.5 Linked Hash Sets and Maps

-LinkedHashSet and LinkedHashMap classes remember in which order you inserted items. As entries are inserted into the table, they are joined in a doubly linked list.





## 9.4.6 Enumeration Sets and Maps

-**EnumSet** is an efficient set implementation with elements that belong to an enumerated type.

-EnumSet class has no public constructor. Use a static factory method:



-**EnumMap** is a map with keys that belongs to an enumerated type. You need to specify the key type in constructor:



+Note: E extends Enum<E>: This means E is an enumerated type. All enumerated types extend the generic Enum class.

## 9.4.7 Identity Hash Maps

-IdentityHashMap: the hash values for the keys are computed by System.identityHashCode().

-For comparison: use ==, not equals()

-Different key objects are considered distinct even if they have equal contents. This class is useful for implementing object traversal algorithms, such as object serialization (keep track of which objects have already been traversed)

-API: p530

# 9.5 Views and Wrappers

-By using **views**, you can obtain objects that implement the Collection. Whose methods **manipulate** the collection

## 9.5.1 Small Collections

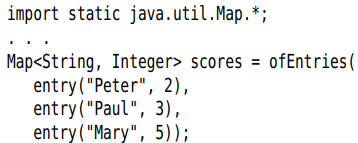
-Static methods yield a set or list with given elements, and a map with given key/value pairs:





+List and Set interface has 11 of() with 0-10 arguments and 1 of() with a variable number of arguments.

+Map interface has a static method ofEntries() that accepts an arbitrary number of Map.Entry<K,V> objects, which you can create with static entry():



+These collection objects are **unmodifiable**. If you want a mutable collection, pass to ArrayList:



-: returns an immutable object that implements List interface, the illusion of having n elements, each of which appears anObject.



-Note: Before of(), there was a static Arrays.asList() returns a list that is mutable but not resizable (can call set() but not add() or remove(). There are also legacy method Collections.emptySet() and Collections.singleton. Do not confuse Collections class and Collection interface

-Tip: Some programmers use a Map.Entry as a poor man’s pair. Nowadays, you can use Map.entry(1st,2nd)

## 9.5.2 Subranges

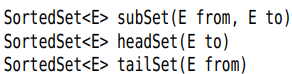
-**subList():** obtain a view into the subrange of list:



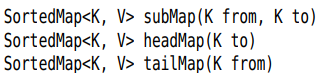
+Any operation to the subrange automatically reflects the entire list.

-For sorted sets and maps, you use the sort order to form subrange.

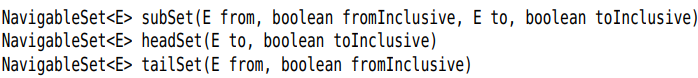
+SortedSet interface declare 3 methods:



+SortedMap interface:

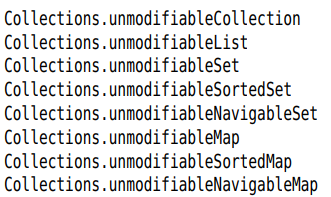


+Navigable interface: you can specify whether the bounds are included:



## 9.5.3 Unmodifiable Views

-Collections class has methods that produce unmodifiable views:



## 9.5.4 Synchronized Views

-If you access a collection from multiple threads, you need to ensure that the collection is not accidentally damaged. Static **synchronizedMap():**

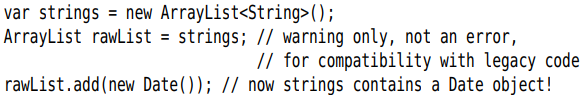


+You can now access the map object from multiple threads. Get() and put() are **synchronized**-each method call must be finished completely before another thread can call another method.

## 9.5.5 Checked Views

-**Checked views are** intended as debugging support for a problem that can occur with generic types.

-It’s possible to smuggle elements of the wrong type into a generic example:



+A checked view can detect the problem.





## 9.5.6 A Note on Optional Operations

-In API doc for collection and iterator interfaces, many methods are “optional operations”. But the purpose of an interface is lay out the methods that a class must implement.

-Should you extend the “optional” methods to your own designs. We think not.

-API: p538

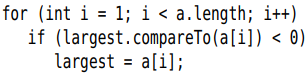
# 9.6 Algorithms

## 9.6.1 Why Generic Algorithms

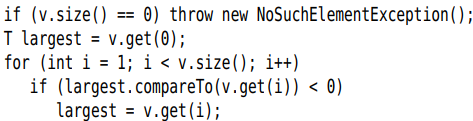
-By generic collection interfaces, you only need to implement your algorithms once.

+Example: Find the largest element of an array

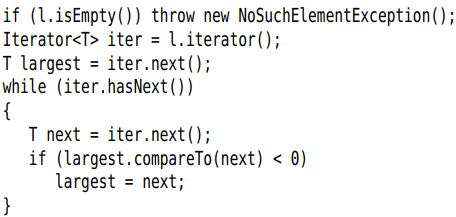




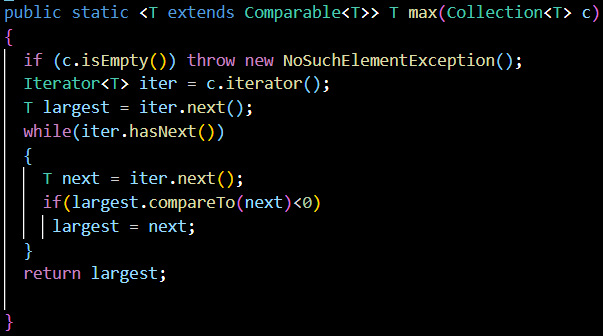
+Find the maximum of an array list:



+Find the maximum of a linked list:



-Think of the **minimal** collection interface that you need to carry out the algorithms. You can implement the max() to tank any objet that implements Collection interface:

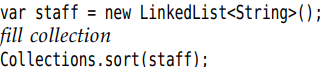


-That’s a powerful concept. Java library contain the basic algorithms: sorting, binary search, some utility algorithms

## 9.6.2 Sorting and Shuffling

-Sorting algorithms are part of the standard library for most programming languages.

-**sort()** in Collections class sorts a collection that implement the List interface:



+This method assumes that the elements implement Comparable interface.

+If you want to sort the list in some **other way**, use sort() of List interface and pass a **Comparator** object.



+Sort in **descending** order: Comparator.reverseOrder():

 sort the reverse according to the order given by compareTo().



-The sort algorithm used in collections library is a bit slower than **Quick-Sort**. It is **stable** – It doesn’t switch equal elements.  
+Example: You have employee list already sorted by name. Now you sort by salary. The ordering by name is preserved. The outcome is sorted 1st by salary, then by name.

-List can be passed to Collections.sort() must be **modifiable** (set) but need not be **resizable** (add, remove)

-**Collections.shuffle()** randomly permutes the order:



-API: p545

## 9.6.3 Binary Search

-To find an object in an array, you normally visit all elements until you find a match.

-**Collections.binarySearch():** if the array is **sorted**, you can look at the middle element. The collection must implement the List interface:



+A non-negative return value denotes the index. If it is negative, there is no matching element. You can use the return value to compute the location where you should insert **element** into the collection to keep it sorted:



-API: p547

## 9.6.4 Simple Algorithms

-Collections class contains simple but useful algorithms.





-API: p548

## 9.6.5 Bulk Operations

-: remove all elements from coll1 that are present in coll2.

- remove all elements from coll1 that are not present in coll2.

-Find the **intersection** of 2 sets:





-Apply a bulk operation to a view. Fired employee:





+Using **subrange** view, you can restrict bulk operations to sublists and subsets:



+Subrange can also be target of a mutating operation:



## 9.6.6 Converting between Collections and Arrays

-Turn an **array into a collection**: **List.of()** wrapper



-Turn a **collection into array**:

+**toArray()** but the return type Object and you can’t change its type.

+Give toArray() an array of length 0 of the type you want:



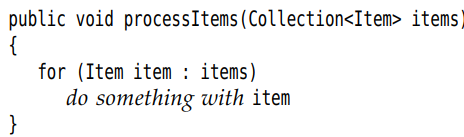
Or to correct size: 

## 9.6.7 Writing your own Algorithms

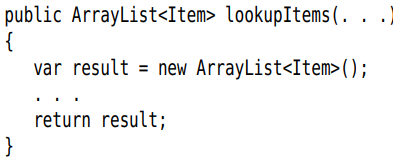
-If you write your own algorithm (method that has a collection as a parameter), you should work with **interface** whenever possible.

-Ask: What is the most general collection interface that can do the job. Do you care about the order -> List. If order doesn’t matter -> any kind

+Tip: In this case, you can use **Iterable<Item>**

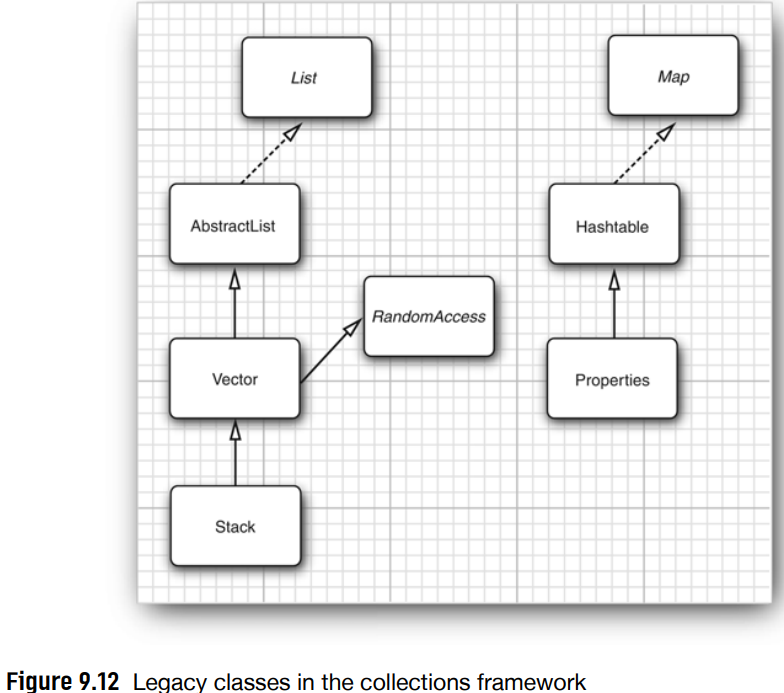


+Conversely, returns multiple elements:



# 9.7 Legacy Collections

-Legacy container classes have been present since the 1st release of Java before there was a collections framework. They have been integrated into the framework.



9.7.1 The Hashtable Class

-**Hashtable** class servers the same purpose as HashMap. Its methods are **synchronized**.

-If you don’t require compatibility with legacy code, use **HashMap**. If you need concurrent access, use **ConcurrentHashMap**.

9.7.2 Enumerations

-Enumeration interface has 2 methods: hasMoreElements() and nextElement() like hasNext() and next().

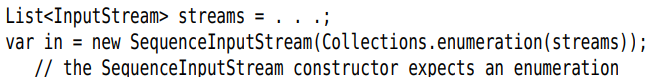
-You can use Collections.list to collect the elements of enum in ArrayList:

’

+Or turn an enum into an iterator:



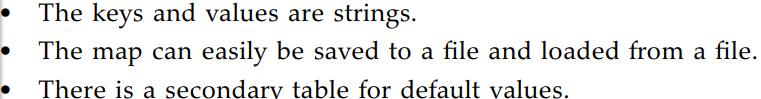
-static Colllections.enumearation():



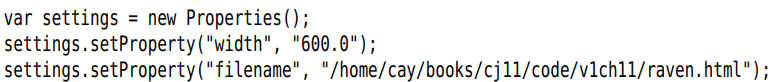
-API: p555

9.7.3 Property Maps

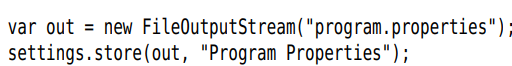
-**Property map** is a map structure of a special type. It has 3 characteristics:



-**Properties** maps are useful in specifying configuration options for programs.



+**store()**: save map list of properties to a file



+**load** the properties from a file:



+**System.getProperties()**: yield a Properties object to describe information

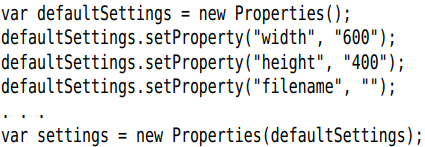
+Note:Properties class implements **Map**. So you can use get() and put(). But you should use getProperty() and setProperty.

+Get the Java version of the virtual machine



-Properties class has 2 mechanism for providing **defaults**:





-Sample program that shows how you can use properties for storing and loading program state: **ImageViewer** from **Chapter 2**.

-**Properties** are simple tables without a hierarchical structure. If you store complex configuration information, use **Preference**.

-API: p557

9.7.4 Stacks

-Stack class: push() and pop().

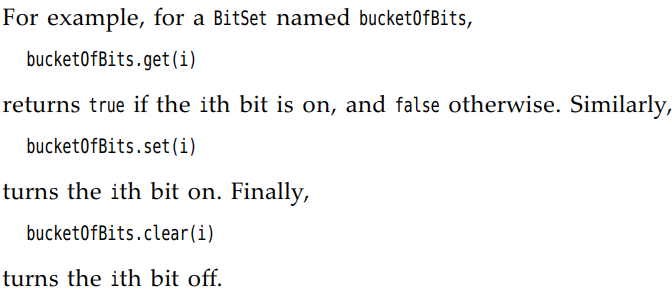
-Stack extends Vector class, you can apply un-stack operations as insert() or remove().

-API: p559 push(), pop(), peek()

9.7.5 Bit Sets

-**BitSet** class stores a sequence of bits. (bit vector or bit array are more appropriate)

-Use a bit set if you need to store a sequence of bits.



-API: p560