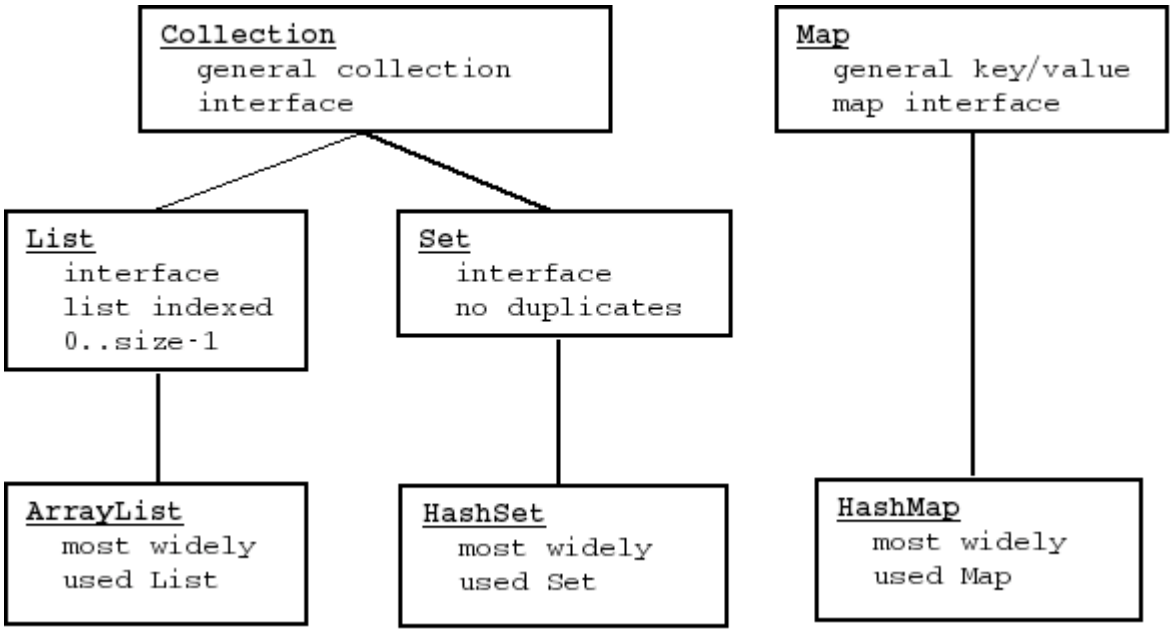
Java Collections -- List Set Map

Copyright 2006-07, by Nick Parlante, (nick.parlante@cs.stanford.edu). The Java "Collection" classes make it easy to store and manipulate collections of information. You should be familiar with the collection classes so you can leverage their many built-in features in your own code.

This document introduces the main features of the java collections framework. The three most important types are "List", "Set", and "Map". A List is like an array, except it grows and shrinks automatically as needed. The Set is like the List, but automatically rejects duplicate elements. The Map is a key/value dictionary that supports the efficient storage and retrieval of information by a key. There are official Sun docs for the collections framework at
http://java.sun.com/javase/6/docs/technotes/guides/collections/index.html

The **Collection** interface is a general interface that includes sub-interfaces **List** and **Set**. If a method has a parameter of type Collection, such as the `addAll(Collection coll)` method below, you can pass it a List or Set and it will work fine. List is an interface, and ArrayList is the typically used class that implements List. Likewise, Set is an interface, and HashSet is the commonly used class that implements Set. The Map interface is separate from the Collection interface. The Map interface defines a key/value lookup dictionary, and HashMap is the most commonly used Map. The sections below explain all of these classes.



Lists

The List is probably the single most useful and widely used type of Collection. List is a general interface, and ArrayList and LinkedList are implementing classes. ArrayList is the best general purpose List, so that's what we'll use here.

A List is a linear structure where each element is known by an index number 0, 1, 2, ... len-1 (like an array). Lists can only store objects, like String and Integer, but not primitives like int. You cannot create a List of int, but you can create a list of Integer objects. This is a common feature of all the Java Collection classes (see boxing below). Another way to say this is that the collection classes can only store pointers.

Basic List

Here is code to create a new list to contain Strings:

```
List<String> words = new ArrayList<String>();
```

The "words" variable is declared to be type "List<String>" -- "List" being the general interface for all lists, and the "<String>" is the generic syntax means this is a list that contains String elements. (Before Java 5, Collections did not have the <String> generic notations, but otherwise worked pretty much the same as shown here.) On the right hand side the "new AarrayList<String>()" creates a new ArrayList of Strings, also using the "List<String>" syntax. The ArrayList class implements the List interface, which is how we can store a pointer to an ArrayList in a List variable. Using the general List type for the variable as shown here is the standard way to store an ArrayList -- that way you can substitute a LinkedList or whatever later if needed.

List add()

A new ArrayList is empty. The **add()** method adds a single element to the end of the list, like this:

```
words.add("this");
words.add("and");
words.add("that");
// words is now: {"this", "and", "that"}
words.size() // returns 3
```

The `size()` method returns the int size of a list (or any collection).

For all the collection classes, creating a new one with the default constructor gives you an empty collection. However, you can also call the constructor passing an existing collection argument, and this creates a new collection that is a copy. So we could copy the elements from words into a second list like this:

```
// Create words2 which is a copy of words
List<String> words2 = new ArrayList<String>(words);
```

Note: this just copies the elements (pointers) that are in "words" into "words2" -- it just does an = for every element in the collection, copying the pointers over.

List Foreach

With Java 5, a very convenient "foreach" syntax was added that iterates over all the elements in a list (also known as the "enhanced for loop"). Here is code to iterate over all the strings and add up their lengths:

```
int lengthSum = 0;
for (String str: words) {
  lengthSum += str.length();
}
```

Each time through the loop, the "str" variable above takes on the next String element in the words list. It is not valid to modify (add or remove) elements from a list while a foreach is iterating over that list -- so it would be an error to put a `words.add("hi")` inside the above loop. The foreach simply goes through the list once from start to finish, and "break" works to exit the loop early. Other, more

powerful forms of iteration are shown below.

List get()

The elements in a List are indexed 0..size-1, with the first element at 0, the next at 1, the next at 2, and so on up through size-1. This zero-based indexing scheme is very common in computer science, for example also being used to index elements in an array and individual chars in a String.

The `get(int index)` method returns an element by its index number:

```
// suppose words is {"this", "and", "that"}
words.get(0)   // returns "this"
words.get(2)   // returns "that"
words.get(3)   // ERROR index out of bounds
```

List For Loop

Here is a for loop that manually calls get() with the index numbers 0, 1, ... size-1 to iterate through all the elements in the list:

```
for (int i=0; i<words.size(); i++) {
  String str = words.get(i);
  // do something with str
}
```

The above int index loop just does the same thing as the foreach but with more code, so there's not much point to it. However, the foreach can only go through the elements one at a time from first to last. However, by manually controlling the index numbers, we can write an int index loop to iterate through the elements in a more complex pattern. For example, here is code that iterates through the elements backwards, skipping every other element:

```
// iterate through the words backwards, skipping every other one
for (int i = words.size()-1; i >= 0; i = i-2) {
  String str = words.get(i);
  // do something with str
}
```

The "Iterator" below is another way to iterate over the elements in a list.

Basic List Methods

Here are other basic list methods that works to set/add/remove elements in a list using index numbers to identify elements (these methods work on Lists but not on general Collection types which don't have index numbers):

- **`get(int index)`** -- returns the element at the given index.
- **`set(int index, Object obj)`** -- sets the element at the given index in a list (the opposite of get). Set() does not change the length of the list, it just changes what element is at an index.
- **`add(Object obj)`** -- adds a new element at the end of the list.
- **`add(int index, Object obj)`** -- adds a new element into the list at the given index, shifting any existing elements at greater index positions over to make a space.
- **`remove(int index)`** -- removes the element at the given index, shifting any elements at greater index positions down to take up the space.

Collection Utility Methods

The Collection interface has many convenience methods for common "bulk" operations. These work on any sort of Collection, including lists and sets. Because these work for any type of Collection, they do not use index numbers. There is code below which demonstrates these methods. It's good to be familiar with these, using them to solve common cases in one line vs. writing the code manually.

- **`int size()`** -- number of elements in the collection
- **`boolean isEmpty()`** -- true if the collection is empty
- **`boolean contains(Object target)`** -- true if the collection contains the given target element anywhere, using equals() for comparisions. For a list, this is a "linear" O(n) cost operation. For a Set, this may be a fast, constant time operation.
- **`boolean containsAll(Collection coll)`** -- true if the collection contains all of the elements in the given collection (order is not significant).
- **`void clear()`** -- removes all the elements in the collection, setting it back to an empty state.
- **`boolean remove(Object target)`** -- searches for and removes the first instance of the target if found. Returns true if an element is found and removed. Uses equals() for comparisons. This is an O(n) cost operation for an ArrayList. Note that this is different from the List method `remove(int index)` which is given the index, so it does not need to search before removing the element.
- **`boolean removeAll(Collection coll)`** -- removes from the receiver collection all of the elements which appear in the given collection (returns true if changed).
- **`boolean addAll(Collection coll)`** -- adds to the receiver collection all of the elements in the given collection (returns true if changed).
- **`boolean retainAll(Collection coll)`** -- retains in the receiver collection only the elements which also appear in the given collection. In other words, removes all the elements which are not in the given collection. After this operation, the receiver will represent effectively a subset of the given collection.
- **`Object[] toArray()`** -- builds and returns an Object[] array containing the elements from the collection. There is a variant which takes an "example" array argument and uses that example array to know what type of array to return -- so call toArray(new String[0]), and it will return a String[] array.

The above bulk methods work for any type of Collection. Here are a couple of the methods which are specific to List, using index numbers:

- **`int indexOf(Object target)`** -- returns the int index of the first appearance of target in the list, or -1 if not found (analogous to the indexOf() method that works on Strings).
- **`List subList(int fromIndex, int toIndex)`** -- returns a new list that represents the part of the original list between fromIndex up to but not including toIndex. Amazingly, add/remove etc. changes to the sublist write through appropriately to modify the original list. It is not valid to make add/remove changes to the original list while the sublist is in use. You can use the subList() method, for example, to pass a part of a list as a parameter to some other method. Making a sublist is much easier than creating and populating a temporary list to pass.

The example code below demonstrates many of the Collection utility methods. It also demonstrates the Arrays.asList() method which takes an array, and returns a List of those elements. It also works just listing the values you want as parameters separated by commas -- `Arrays.asList(1, 2, 3)` -- this is a Java 5 feature, where a method can take a variable number of arguments. The resulting list cannot change size (in reality, it uses the original fixed-size array to store the elements), but it's a quick way to make a list.

```
// To get started, create an array of strings,
// and use Arrays.asList() to convert it to a List.
```

```
String[] array = new String[] {"red", "green", "blue", "ochre"};
List colors1 = Arrays.asList(array);
// Modifications like add/remove will fail on colors1 -- it is not a true list,
// just an array wrapped to look like a list. (The "adapter" pattern.)

// Construct a new ArrayList using the elements of colors1 --
// most collections have a constructor that takes initial contents
// like this.
// (or could have used addAll() to add the elements of colors1).
List colors2 = new ArrayList(colors1);
colors2.add("purple");  // Add "purple" on the end.

// colors 2 is {"red", "green", "blue", "ochre", "purple"
String str = colors2.toString(); // "[red, green, blue, ochre, purple]"
System.out.println(str);
int size = colors2.size(); // 5
boolean hasOchre = colors2.contains("ochre"); // true
int indexOcher = colors2.indexOf("ochre"); // 3

// Search out and remove the first "ochre".
colors2.remove("ochre");
// colors2 is {"red", "green", "blue", "purple"};

// Make up some more colors and add them.
List colors3 = Arrays.asList("yellow", "pink", "purple");
colors2.addAll(colors3);
// colors2 is {"red", "green", "blue", "purple", "yellow", "pink", "purple"}

// Remove all "purple" and "yellow".
colors2.removeAll(Arrays.asList("purple", "yellow"));
// colors2 is {"red", "green", "blue", "pink"}
System.out.println("remove all " + colors2);

// Use subList() to make a "front" List showing just elements 0, 1, 2 of colors2.
List front = colors2.subList(0, 3);
System.out.println("front " + front); // [red, green, blue]
// front is {"red", "green", "blue"}

// Can make changees to front, and they go through to underlying colors2,
// but do not make changes to colors2 while using front.
front.contains("green");  // true
front.remove("green");
front.add("orange");
// front is {"red", "blue", "orange"}
// colors2 is {"red", "blue", "orange", "pink"}
```

Iterator

The foreach syntax -- for (String str: words) ... -- is the easiest way to iterate over a list. "Iterator" objects provide an alternative and more flexible way to iterate over a list. In fact, behind the scenes, the foreach syntax just creates an Iterator object to do the iteration.

Iterator objects are generic by the element type, so for a List<String>, you use an Iterator<String>. Iterators are shown here with lists, but iterators work for all collection types. To use an Iterator, call the collection's iterator() method which returns an Iterator object, ready to iterate over that collection:

```
// Suppose we have a "words" list of strings:
List words = new ArrayList(); // create a list of strings
```

```
// Here's how to create an iterator to go through all the words:
Iterator it = words.iterator();
```

The Iterator object is temporary -- you use it to go through the elements in the list and then discard it. The Iterator has two main methods:

- **`boolean hasNext()`** -- returns true if the iterator has more elements.
- **`T next()`** -- returns the "next" element from the list which will be type T, where T is the generic type of the list (e.g. String). Only call this if hasNext() returns true.

Use the iterator with a loop like this:

```
Iterator<String> it = words.iterator();
while (it.hasNext()) {
  String str = it.next();
  // Do something with str
}
```

Each call to next() yields the next element from the list until there are no more elements, at which point hasNext() returns false. It is an error to call next() when hasNext() is false. During the hasNext()/next() iteration, it is not valid to modify the underlying list with add/remove operations. Code that uses iteration must not modify the collection during the iteration.

Iterator Remove

The Iterator has a `remove()` method that removes the element from the previous call to next(). It is not valid to modify the list during iteration generally, but iterator remove() is an allowed exception. It is only valid to call it.remove() once for each call to it.next(). For example, here is code that iterates over the words list, removing all the words of length 4 or more:

```
// words is: {"this", "and", "that"}
// Remove words length 4 or more.
Iterator<String> it = words.iterator();
while (it.hasNext()) {
  String str = it.next();
  if (str.length() >= 4) it.remove();
}
// words is: {"and"}
```

Using it.remove() during iteration is potentially a very clean and efficient strategy -- it does not have to search the whole list for the element, since the element to remove is at the current spot of the iterator. Contrast this to the more expensive `list.remove(Object target)` above which must search the whole list for the target and then remove it.

It's possible to have multiple iterators going over a collection at the same time, each proceeding through all the elements independently. In that case, no add/remove modifications to the collection are allowed -- it would be too complicated for the multiple iterators to coordinate their changes.

The above code demonstrates the basic Iterator class which works for all collection types, including lists. There is a more powerful type of iterator, the ListIterator, which works for list types, but not other collection types. The ListIterator can go forwards and backwards and can insert and delete. The ListIterator is powerful but more rarely used -- you can get quite far with the plain foreach for common loops, and the Iterator when you want to delete during iteration. Iterators are not restricted to the Collection classes -- any class that implements the Iterable interface to provide an Iterator object with hasNext(), next(), etc. can support iteration just like the collection classes.

Boxing and Unboxing

Lists (and the other collection classes) can only contain pointers to objects such as Strings or Integers or other Lists. The Collection classes can also store **null**, which is a valid pointer. They cannot store primitives like "int" and "boolean". To get around this, you can wrap an int value in an Integer object, or a boolean in a Boolean object, and put that in the list. In Java 5, the "auto boxing" feature automates this conversion between int values and Integer objects, and "auto unboxing" goes the other way. So if you want to store a list of int values, you can create a List<Integer> and when you call add() or get(), the auto boxing/unboxing should do the int/Integer conversions for you.

```
// Create a list of Integer objects
List<Integer> nums = new ArrayList<Integer>();
// Note that List<int> does not work

// Add the squares of 1..10
for (int i=1; i<=10; i++) {
        nums.add(i * i);
        // Here autoboxing boxes the int value into an Integer object
}

if (nums.contains(16)) {  // Autoboxing converts int 16
        System.out.println("contains 16!");
}

// Autoboxing works with Arrays.asList() too
List<Integer> nums2 = Arrays.asList(1, 2, 3, 4);
nums.addAll(nums2);
```

Storing an int value in an Integer object adds some cost, so storing a List of many Integer objects is less efficient than a plain int[] array of the same values. On the other hand, the array is fixed size while the list has useful built-in add/remove/search features. The list uses more memory but is more powerful.

There is this one very unfortunate case with auto unboxing, which is that it does not work properly with == and !=. Using == and !=, you should manually unbox by calling intValue() on the Integer object to extract the int:

```
// Auto unboxing DOES NOT WORK with == -- this code is wrong.
if (nums.get(0) == 126) {
        System.out.println("found 126");
}

// Use .intValue() to unbox manually -- this code is correct.
if (nums.get(0).intValue() == 126) {
        System.out.println("found 126");
}
```

This flaw is due to the need to support Java code that pre-dates auto boxing/unboxing. However, it is annoying and I hope it will be fixed in a future version of Java.

Sets

The Java "Set" is a Collection, like the List, but with the added constraint that each element can be in the Set once, using equals() to test if two elements are the same. If an add() operation tries to add an element that is already in the set, the add() is silently ignored. This can be useful to store values where you want this sort of mathematical set behavior. The standard Collection methods -- contains(),

addAll(), iterator()... -- work for Sets as they do for any Collection. With Sets, the standard utility methods addAll(), retainAll(), containsAll() now give you fully functioning mathematical set operations. To make a set union of x and y, call x.addAll(y). To make the intersection of sets x and y, call x.retainAll(y). To test if x is a subset of y, call y.containsAll(x). Sets do not impose a 0..size-1 indexing of the elements (that's what Lists do), so List methods like get(int index) are not available for sets. Sets are potentially more efficient than lists. In particular, the HashSet can find or insert an element in constant time.

```
// Create a Set of Integer objects
Set<Integer> nums = new HashSet<Integer>();

// Add the numbers 1..10
for (int i=1; i<=10; i++) {
        nums.add(i);
}

// Add 1, 4, 9, 16, 25
// Since it's a Set, only the add() of 16 and 25 do anything.
for (int i=1; i<=5; i++) {
        nums.add(i * i);
}


// Foreach works on a Set.
for (int num:nums) {
        System.out.println(num);
}

// Iterator works on a set.
// (the values will appear in some random order for a HashSet
//  as we have here)
Iterator<Integer> it = nums.iterator();
while (it.hasNext()) {
        int val = it.next();
}

// Other Collection utilties work
nums.contains(9); // true
nums.containsAll(Arrays.asList(1, 2, 3)); // true

// addAll() is essentially a mathematical union operation.
// Change nums to the union with the set {16, 17}
nums.addAll(Arrays.asList(16, 17));

// Accessing by index number DOES NOT work
// (index numbers are List feature only)
// int val2 = nums.get(0);  //NO does not compile
```

The HashSet is the most commonly used, as shown above. HashSet only works with elements, like String and Integer, which have a hashCode() defined. The TreeSet is an alternative which is a little more costly, but keeps the set in sorted order, so iteration will yield the values in sorted order.

Map

A Map is very different from List and Set. A Map is a key/value table that can look up any entry by key very efficiently (know as a "hash table" or "dictionary").

"Map" is a general interface of the basic map features, implemented by two main classes: HashMap and TreeMap. HashMap is by far the more commonly used one, and that's what we will us here.

A Map stores key/value entries, where each key in the map is associated with a single value. For example, here is a map where the each key is the string name of a city and the associated value is the lat/long location of that city (in this case, both key and value are strings):

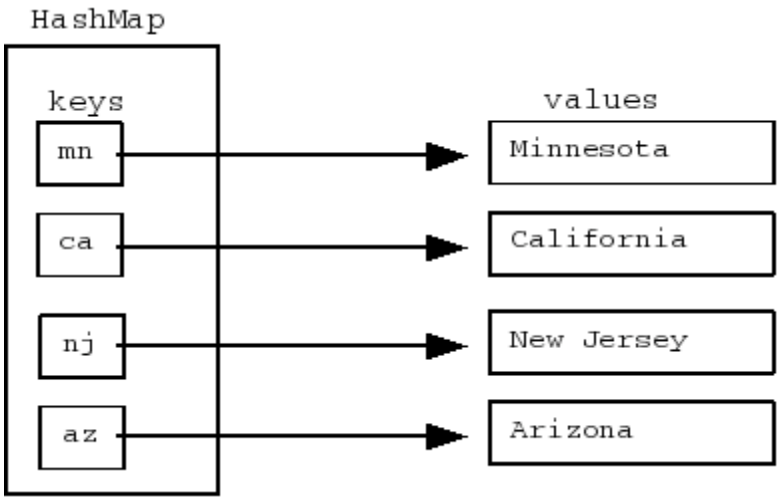
Most uses of a map involve just two methods **put()** and **get()**:

- **`put(Object key, Object value)`** -- puts an entry for the given key into the map with the given value. If there was an existing value for that key, it is overwritten by this new value.
- **`Object get(Object key)`** -- gets the value previously stored for this key, or null if there is no entry for this key in the map.
- **`boolean containsKey(Object key)`** -- returns true if the map contains an entry for the given key.
- **`int size()`** -- returns the number of key/value entries in the map

With Java version 5, the Map is most often used with generics to indicate the type of key and the type of value. Both the key and value must be object types such as String or Integer or List. For example, below is code that creates a new HashMap where both the key and value are Strings. "Map" is the general interface for maps, and "HashMap" is the particular type of map created here. It is standard to declare the variable with the general type, Map, as shown here:

```
Map<String, String> states = new HashMap<String, String>();

states.put("ca", "California");
states.put("az", "Arizona");
states.put("mn", "Minnesota");
states.put("nj", "New Jersey");
```



Once a value is put in the map with put(), it can be retrieved with get():

```
states.get("ca")  // returns "California"
states.get("nj")  // returns "New Jersey"
states.get("xx")  // returns null, there is no entry for "xx"

states.put("nj", "Garden State");  // Put a new entry for "nj", overwriting the old entry
states.get("nj")  // returns "Garden State"
```

For a HashMap, the keys are stored in a seemingly random order. The important feature of get() and put() is that they are **fast**. With a HashMap, even if there are a 100,000 entries in the map, get() and put() can access a particular entry almost instantaneously (constant time). This very fast performance is a feature of HashMap; the TreeMap is slower. That's why everyone uses HashMap. For whatever problem you are solving, if there is part of the problem that involves storing information under some key and retrieving that information later ... use a HashMap. The HashMap is simple to use, reliable, and fast.

Map values() and keySet()

The values() and keySet() methods on a map give handy access to the entries in the map.

- **`Collection values()`** -- returns a "live" read-only Collection showing all the map values in some random order. Iterate over the values collection to see all the values, however the collection cannot be changed, so add/remove operations will fail. The values collection updates live as the map changes (there is no real separate collection, the calls just go through to the underlying map). Copy the values collection into a separate ArrayList to insulate from subsequent changes to the map, or to change/sort the values.
- **`Set keySet()`** -- returns a "live" set of all the keys in the map. Iterate over the keys set to see all the keys of the map. Later changes to the map will be reflected in the key set live (there is no separate set, the calls on the key set go through to the underlying map). As a cute detail, removing an element from the key set removes the corresponding entry from the map. Adding to the key set does not work -- you must put() on the map itself. Note that the common Collection operations work on Sets: foreach, iterator(), add(), remove(), addAll(), removeAll(), etc..

Map Example Code

```
Map<String, String> states = new HashMap<String, String>();

states.put("ca", "California");
states.put("az", "Arizona");
states.put("mn", "Minnesota");
states.put("nj", "New Jersey");

System.out.println(states.get("ca"));  // California
System.out.println(states.get("zz"));  // null (no entry for key "zz")
states.put("ca", "The Golden State");  // Overwrite the old entry

// Pull out live Collection of all the values.
Collection<String> values = states.values();
System.out.println(values);  // [Minnesota, New Jersey, The Golden State, Arizona]


// Pull out live set of the keys -- use to print key->value for
// the whole map. The order of the keys is random for a HashSet.
Set<String> keys = states.keySet();
for (String key:keys) {
        System.out.println(key + "->" + states.get(key));
}
// mn->Minnesota
// nj->New Jersey
// ca->The Golden State
// az->Arizona

// More complex -- create a map of Strings to Lists of Integers
```

```
Map<String, List<Integer>> map = new HashMap<String, List<Integer>>();
List<Integer> nums = new ArrayList<Integer>();
nums.addAll(Arrays.asList(1, 2, 3, 4, 5));
map.put("1-5", nums);
```

Map Entry Set

Values() and keySet() provide the easiest bulk access to a Map. The one problem with those methods is that they provide access to either the keys or the values, but not both. The entrySet() method provides a Set of special `Map.Entry<KEY_TYPE, VALUE_TYPE>` objects (the Entry class is defined inside the Map class). Each Map.Entry object contains one key and one value, accessible through the Map.Entry getKey() and getValue() methods. So if you want to look at all of data in a Map, get the entrySet() and iterate over it. This provides access to the entry data directly from the Map's internal data structures without the cost of calling get() for each key.

```
// Each Map.Entry object contains one String key + one String value.
// entrySet() returns a set of all the entries.
Set<Map.Entry<String, String>> entries = states.entrySet();
for (Map.Entry<String, String> entry: entries) {
        System.out.println(entry.getKey() + "->" + entry.getValue());
}
```

The keySet() makes the most sense where performance is important. If performance is not critical, I find the plain keySet() approach to be more readable.