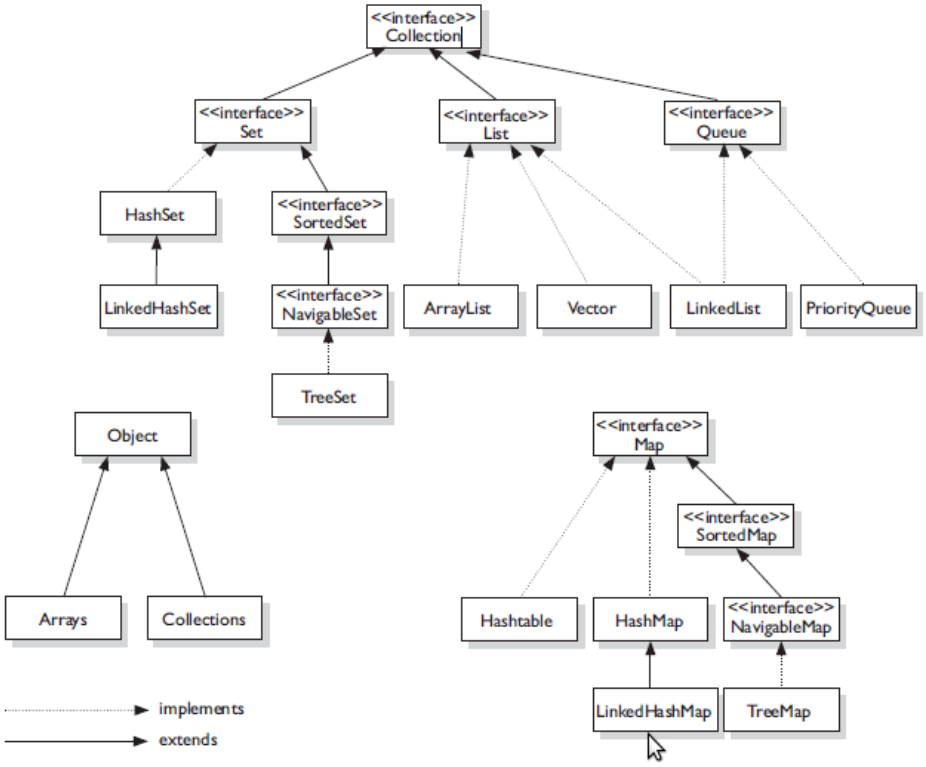
03/11 2013

# Java Collection Framework cheat sheet

A *collection* is a data structure—actually, an object—that can hold references to other objects. Usually, collections contain references to objects that are all of the same type. The collections-framework interfaces declare the operations to be performed generically on various types of collections.

| Interface | Description |
|---|---|
| Collection | The root interface in the collections hierarchy from which interfaces Set, Queue and List are derived. |
| Set | A collection that does not contain duplicates. |
| List | An ordered collection that can contain duplicate elements. |
| Map | A collection that associates keys to values and cannot contain duplicate keys. |
| Queue | Typically a first-in, first-out collection that models a waiting line; other orders can be specified. |




Quote

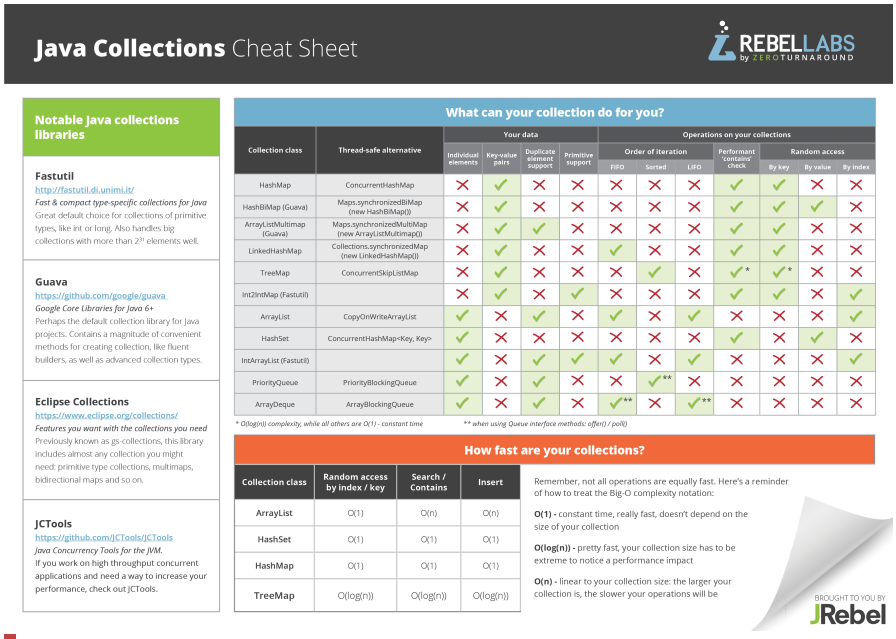
My interest is in the future because I am going to spend the rest of my life there.

*Charles F. Kettering*

Categories

- Android
- AngularJS
- Databases
- Development
- Django
- Java
- JavaScript
- LaTex
- Linux
- Meteor JS
- Python
- Science

Archive ↓



**Java Collections** Cheat Sheet

REBELLABS by ZEROTURNAROUND

**Notable Java collections libraries**

**Fastutil**
http://fastutil.di.unimi.it/
*Fast & compact type-specific collections for Java*
Great default choice for collections of primitive types, like int or long. Also handles big collections with more than $2^{31}$ elements well.

**Guava**
https://github.com/google/guava
*Google Core Libraries for Java 6+*
Perhaps the default collection library for Java projects. Contains a magnitude of convenient methods for creating collection, like fluent builders, as well as advanced collection types.

**Eclipse Collections**
http://www.eclipse.org/collections/
*Features you want with the collections you need*
Previously known as gs-collections, this library includes almost any collection you might need: primitive type collections, multimaps, bidirectional maps and so on.

**JCTools**
https://github.com/JCTools/JCTools
*Java Concurrency Tools for the JVM.*
If you work on high throughput concurrent applications and need a way to increase your performance, check out JCTools.

**What can your collection do for you?**

| Collection class | Thread-safe alternative | Your data: Individual elements | Your data: Key-value pairs | Your data: Duplicate element support | Your data: Primitive support | Operations on your collections: Order of iteration: FIFO | Order of iteration: Sorted | Order of iteration: LIFO | Performant 'contains' check | Random access: By key | Random access: By value | Random access: By index |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| HashMap | ConcurrentHashMap | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ |
| HashBiMap (Guava) | Maps.synchronizedBiMap (new HashBiMap()) | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ | ✗ |
| ArrayListMultimap (Guava) | Maps.synchronizedMultiMap (new ArrayListMultimap()) | ✗ | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ |
| LinkedHashMap | Collections.synchronizedMap (new LinkedHashMap()) | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ |
| TreeMap | ConcurrentSkipListMap | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✓* | ✓* | ✗ | ✗ |
| Int2IntMap (Fastutil) | | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ | ✓ | ✗ | ✓ |
| ArrayList | CopyOnWriteArrayList | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ |
| HashSet | ConcurrentHashMap<Key, Key> | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✓ | ✗ |
| IntArrayList (Fastutil) | | ✓ | ✗ | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ |
| PriorityQueue | PriorityBlockingQueue | ✓ | ✗ | ✓ | ✗ | ✗ | ✓** | ✗ | ✗ | ✗ | ✗ | ✗ |
| ArrayDeque | ArrayBlockingQueue | ✓ | ✗ | ✓ | ✗ | ✓** | ✗ | ✓** | ✗ | ✗ | ✗ | ✗ |

*O(log(n)) complexity, while all others are O(1) - constant time ** when using Queue interface methods: offer() / poll()

**How fast are your collections?**

| Collection class | Random access by index / key | Search / Contains | Insert |
|---|---|---|---|
| ArrayList | O(1) | O(n) | O(n) |
| HashSet | O(1) | O(1) | O(1) |
| HashMap | O(1) | O(1) | O(1) |
| TreeMap | O(log(n)) | O(log(n)) | O(log(n)) |

Remember, not all operations are equally fast. Here's a reminder of how to treat the Big-O complexity notation:

**O(1) -** constant time, really fast, doesn't depend on the size of your collection

**O(log(n)) -** pretty fast, your collection size has to be extreme to notice a performance impact

**O(n) -** linear to your collection size: the larger your collection is, the slower your operations will be

BROUGHT TO YOU BY JRebel

## Array

Array are fixed length data structure.

Creating an array

```
int[] intArray = new int[10];
```

Creating and initializing array in same line

```
int[] intArray3 = new int[]{1,2,3,4};
```

Loop array

```
for(int i: numbers){
   System.out.println(i);
}
```

Sort array

```
String[] companies = { "Google", "Apple", "Sony" };
Arrays.sort(companies);
```

Sort array in reverse order

```
Arrays.sort(companies, Collections.reverseOrder());
```

Creating an multi-dimensional array

```
int[][] multiArray = new int[2][3];
```

Creating and initializing multi-dimensional array in same line

```
int[][] multiArray = {{1,2,3},{10,20,30}};
```

## ArrayList

Advantage of *ArrayList* is that it can resize itself. Since we can not modify size of an array after creating it, we prefer to use ArrayList in Java which re-size itself automatically once it gets full. *ArrayList* in Java implements List interface and allow *null*. Java *ArrayList* also maintains insertion order of elements and allows duplicates opposite to any *Set* implementation which doesn't allow duplicates.

You can use ArrayList in Java with or without Generics both are permitted. Generics version is recommended because of enhanced type-safety.

```
ArrayList<String> stringList = new ArrayList<String>();
```

Putting an item into *ArrayList*

```
stringList.add("one");
stringList.add("two");
```

To assign to a position, we use `set`. The index must be valid. The `ArrayList` must already *contain* a reference at the index.

```
collection.set(1, 10);
```

An empty `ArrayList` has no elements. When we call `clear()` on an `ArrayList`, it will become empty. This method is useful when we want to reuse an existing `ArrayList`.

`isEmpty()` method returns true if the `ArrayList` has zero elements.

With `remove()` method we delete an element. The element slot is removed and any later elements are shifted forward. We can use an index or a value argument.

If we pass a value, `remove()` searches for the first occurrence and removes that element. This is slower than using an index.

```
collection.remove(1); // by position
collection.remove("item1"); // by value
```

With `Collections.addAll` we add many elements to an `ArrayList` at once. The second argument is either an array of the elements to add, or those elements as arguments.

```
//import java.util.ArrayList;
//import java.util.Collections;

ArrayList<Integer> values = new ArrayList<>();
Integer[] array = { 10, 20, 30 };

// add all elements in array to ArrayList
Collections.addAll(values, array);

// add more elements
Collections.addAll(values, 40, 50);
```

*Checking Index of an Item in Java Arraylist*

You can use `indexOf()` method of *ArrayList* in Java to find out index of a particular object.

```
int index = stringList.indexOf("Item"); //location of Item object in List
```

*Creating ArrayList from Array in Java*

```
ArrayList<String> stringList = Arrays.asList(new String[]{"one", "two", "three");
```

Convert Array to ArrayList in Java

```
String[] numbers = {"one", "two", "three"};
List numberList = Arrays.asList(numbers);
```

Iterate over list

```
for(String item : numberList){
   System.out.println(item);
}
```

Sort ArrayList

```
List collection = new ArrayList();
collection.add(1);
collection.add(4);
collection.add(2);
collection.add(3);
Collections.sort(collection);
```

Sort ArrayList in reverse order

```
List collection = new ArrayList();
collection.add(1);
collection.add(4);
collection.add(2);
```

```
collection.add(3);
Collections.sort(collection, Collections.reverseOrder());
```

Sometimes we need to check whether an element *exists in ArrayList* in Java or not for this purpose we can use `contains()` method of Java. `contains()` method takes the type of object defined in ArrayList creation and returns true if this list contains the specified element.

```
List array = Arrays.asList(1, 3, 5, 2, 4);
if (array.contains(3)) {
    System.out.println("Element found inside ArrayList");
};
```

### Difference between LinkedList and ArrayList

Main difference between ArrayList and LinkedList is that ArrayList is implemented using resizable array while LinkedList is implemented using doubly LinkedList.

Here are another differences

1) Insertions are easy and fast in LinkedList as compared to ArrayList because there is no risk of resizing array and copying content to new array if array gets full which makes adding into ArrayList of O(n) in worst case, while adding is O(1) operation in LinkedList. ArrayList also needs to update its index if you insert something anywhere except at the end of array.

2) Removal is like insertions better in LinkedList than ArrayList.

3) LinkedList has more memory overhead than ArrayList because in ArrayList each index only holds actual object (data) but in case of LinkedList each node holds both data and address of next and previous node.

```
LinkedList l = new LinkedList();
l.add("A1");
l.add("B");
l.add("C");
l.addLast("Z");
l.add(1, "A0");

l.remove("B");
l.remove(2);

Object v = l.get(2);
l.set(2, (String) v + "0");
```

There are few similarities between these classes which are as follows:

- Both `ArrayList` and `LinkedList` are implementation of `List` interface.
- They both maintain the elements insertion order which means while displaying `ArrayList` and `LinkedList` elements the result set would be having the same order in which the elements got inserted into the `List`.
- Both these classes are non-synchronized and can be made synchronized explicitly by using `Collections.synchronizedList` method.
- The `iterator` and `listIterator` returned by these classes are fail-fast (if list is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove or add methods, the iterator will throw a `ConcurrentModificationException`).

Let's compare `LinkedList` and `ArrayList` by following parameters:

*1. Implementation*. `ArrayList` is the resizable array implementation of `List` interface, while `LinkedList` is the Doubly-linked list implementation of the `List` interface.

*2. Performance*. `ArrayList` `get(int index)` operation runs in constant time i.e O(1) while `LinkedList` `get(int index)` operation run time is O(n) .

The reason behind `ArrayList` being faster than `LinkedList` is that `ArrayList` uses index based system for its elements as it internally uses array data structure, on the other hand, `LinkedList` does not provide index based access for its elements as it iterates either from the beginning or end (whichever is closer) to retrieve the node at the specified element index.

`insert()` or `add(Object)` operation. Insertions in `LinkedList` are generally fast as compare to `ArrayList`. In `LinkedList` adding or insertion is O(1) operation. While in `ArrayList`, if array is full i.e worst case, there

is extra cost of resizing array and copying elements to the new array, which makes runtime of add operation in `ArrayList` O(n), otherwise it is O(1) .

`remove(int)` operation. Remove operation in `LinkedList` is generally same as `ArrayList` i.e. O(n). In `LinkedList`, there are two overloaded `remove` methods. Oe is `remove()` without any parameter which removes the head of the list and runs in constant time O(1). The other overloaded remove method in `LinkedList` is `remove(int)` or `remove(Object)` which removes the `Object` or int passed as parameter. This method traverses the `LinkedList` until it found the `Object` and unlink it from the original list. Hence this method run time is O(n).

While in `ArrayList` `remove(int)` method involves copying elements from old array to new updated array, hence its run time is O(n).

*3. Reverse Iterator*. `LinkedList` can be iterated in reverse direction using `descendingIterator()` while there is no `descendingIterator()` in `ArrayList`, so we need to write our own code to iterate over the `ArrayList` in reverse direction.

*4. Initial Capacity*. If the constructor is not overloaded, then `ArrayList` creates an empty list of initial capacity 10, while `LinkedList` only constructs the empty list without any initial capacity.

*5. Memory Overhead*. Memory overhead in `LinkedList` is more as compared to `ArrayList` as node in `LinkedList` needs to maintain the addresses of next and previous node. While in `ArrayList` each index only holds the actual object(data).

## Set

Unlike List, Set doesn't keep insertion order and doesn't allow any duplicates.

```
ArrayList numbers = new ArrayList();
numbers.add("1");
numbers.add("2");
numbers.add("3");
numbers.add("2");

//Converting ArrayList into HashSet in Java

HashSet numberSet = new HashSet(numbers);
numberSet.add("2");
numberSet.contains("3")
```

Popular implementation of List interface in Java includes ArrayList, Vector and LinkedList. While popular implementation of Set interface includes HashSet, TreeSet and LinkedHashSet.

## Map

Map in Java allows duplicate value which is fine with List which also allows duplicates but Map doesn't allow duplicate key.

```
HashMap<String, String> animals = new HashMap<String, String>();
animals.put("cat", "one");
animals.put("dog", "two");
animals.put("mouse", "one");

// converting HashMap keys into ArrayList
List<String> keyList = new ArrayList<String>(animals.keySet());
System.out.println("Size of Key list from Map: " + keyList.size());

// converting HashMap Values into ArrayList
List<String> valueList = new ArrayList<String>(animals.values());
System.out.println("Size of Value list from Map: " + valueList.size());
```

Iterating map

```
for (String key : animals.keySet()) {
  System.out.println("Key: " + key + " Value: " + animals.get(key));
}
```

`get()` method looks into the `HashSet` and, if found, returns the value for the key. Please be careful not to call the `get()` method on a key that does not exist. An exception will be thrown.

We can use the `containsKey` to see if the key exists. It returns *true* if the key is found, and *false* otherwise.

```
if (animals.containsKey('cat')) {
    System.out.println("cat was found");
}
```

`containsValue()` returns *true* if a specified value exists. To get keys with a value, we must use a loop, more than one key may have a single value.

```
if (animal.containsValue("two")) {
    System.out.println("two value detected!");

    // loop over all keys and print them if they have "two" values.
    for (String key : animals.keySet()) {
        if (animals.get(key) == "two") {
            System.out.println(key);
        }
    }
}
```

`size()` is the count of entries (or of keys).

`isEmpty()` returns *true* if the `HashMap` has a size of zero.

`getOrDefault` method safely get a value from `HashMap`. If the key does not exist, no error occurs. Instead, the default value (argument 2) is returned.

```
animals.getOrDefault("bird", -1);
```

`put()` will replace an existing value. But `putIfAbsent()` will not. It only adds the value to the `HashMap` if no key currently exists for it.

```
animals.putIfAbsent("mouse", "three");
```

A `HashMap` is unordered. It cannot be directly sorted, but we can *sort its keys* and process them (and their values) in order. We use `keySet` and add the keys to an `ArrayList`.

```
// put keys into an ArrayList and sort it
Set<String> set = animals.keySet();
ArrayList<String> list = new ArrayList<String>();
list.addAll(set);
Collections.sort(list);

// display sorted keys and their values
for (String key : list) {
    System.out.println(key + ": " + hash.get(key));
}
```

## Hashtable

Hashtable is similar to HashMap, but is synchronized. Like HashMap, Hashtable stores key/value pairs in a hash table. When using a Hashtable, you specify an object that is used as a key, and the value that you want linked to that key. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.

HashMap allows null values as key and value whereas Hashtable doesn't allow nulls.

```
Hashtable<Integer, String> mapToString = new Hashtable<Integer,String>();
mapToString.put(new Integer(1), "Two");
mapToString.put(new Integer(2), "One");
mapToString.put(new Integer(4), "Four");
mapToString.put(new Integer(3), "Three");
```

## Takeaway

Java has dozens of collection classes and interfaces. Below are some of the considerations that may help you to choose one.

- If you need to access data by index, consider using `ArrayList`.
- If you need to often insert or remove data in/from a collection, a `LinkedList` should be a good choice
- If you need a collection that doesn’t allow duplicate elements, use one of the collections that implements `Set` interface. For fast access use `HashSet`. For sorted set use `TreeSet`.

- For storing key/value pairs use a collection that implements the `Map` interface; e.g. `HashMap` or `HashTable`.
- If you need a collection for a fast search that remains fast regardless of the size of the data set use `HashSet`.

## Additional material

- **Which Java collection to use?**
- **Java Collections Cheat Sheet**
- **Java Collections Tutorial**
- **Java Generics Tutorial**