Set Implementations

1. [HashSet](https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html)
2. [TreeSet](https://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html)
3. [LinkedHashSet](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashSet.html)
4. [EnumSet](https://docs.oracle.com/javase/8/docs/api/java/util/EnumSet.html)
5. [CopyOnWriteArraySet](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CopyOnWriteArraySet.html)

List Implementations

1. [ArrayList](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html)
2. [LinkedList](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html)
3. [CopyOnWriteArrayList](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CopyOnWriteArrayList.html)

**Map Implementations**

1. [HashMap](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)
2. [TreeMap](https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html)
3. [LinkedHashMap](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html)
4. [EnumMap](https://docs.oracle.com/javase/8/docs/api/java/util/EnumMap.html)
5. [WeakHashMap](https://docs.oracle.com/javase/8/docs/api/java/util/WeakHashMap.html)
6. [IdentityHashMap](https://docs.oracle.com/javase/8/docs/api/java/util/IdentityHashMap.html)
7. [ConcurrentMap](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentMap.html)

# Queue Implementations

1. [LinkedBlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedBlockingQueue.html) — an optionally bounded FIFO blocking queue backed by linked nodes
2. [ArrayBlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ArrayBlockingQueue.html) — a bounded FIFO blocking queue backed by an array
3. [PriorityBlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/PriorityBlockingQueue.html) — an unbounded blocking priority queue backed by a heap
4. [DelayQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/DelayQueue.html) — a time-based scheduling queue backed by a heap
5. [SynchronousQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/SynchronousQueue.html) — a simple rendezvous mechanism that uses the BlockingQueue interface
6. [PriorityQueue](https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html)
7. [LinkedTransferQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedTransferQueue.html) — an unbounded TransferQueue based on linked nodes

# Deque Implementations

1. LinkedList
2. ArrayDeque
3. [LinkedBlockingDeque](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedBlockingDeque.html)

# Wrapper Implementations

The synchronization wrappers add automatic synchronization (thread-safety) to an arbitrary collection. Each of the six core collection interfaces — [Collection](https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html), [Set](https://docs.oracle.com/javase/8/docs/api/java/util/Set.html), [List](https://docs.oracle.com/javase/8/docs/api/java/util/List.html), [Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html), [SortedSet](https://docs.oracle.com/javase/8/docs/api/java/util/SortedSet.html), and [SortedMap](https://docs.oracle.com/javase/8/docs/api/java/util/SortedMap.html) — has one static factory method.

public static <T> Collection<T> synchronizedCollection(Collection<T> c);

public static <T> Set<T> synchronizedSet(Set<T> s);

public static <T> List<T> synchronizedList(List<T> list);

public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m);

public static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s);

public static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m);

## Unmodifiable Wrappers

Unlike synchronization wrappers, which add functionality to the wrapped collection, the unmodifiable wrappers take functionality away. In particular, they take away the ability to modify the collection by intercepting all the operations that would modify the collection and throwing an UnsupportedOperationException. Unmodifiable wrappers have two main uses, as follows:

* To make a collection immutable once it has been built. In this case, it's good practice not to maintain a reference to the backing collection. This absolutely guarantees immutability.
* To allow certain clients read-only access to your data structures. You keep a reference to the backing collection but hand out a reference to the wrapper. In this way, clients can look but not modify, while you maintain full access.

Like synchronization wrappers, each of the six core Collection interfaces has one static factory method.

public static <T> Collection<T> unmodifiableCollection(Collection<? extends T> c);

public static <T> Set<T> unmodifiableSet(Set<? extends T> s);

public static <T> List<T> unmodifiableList(List<? extends T> list);

public static <K,V> Map<K, V> unmodifiableMap(Map<? extends K, ? extends V> m);

public static <T> SortedSet<T> unmodifiableSortedSet(SortedSet<? extends T> s);

public static <K,V> SortedMap<K, V> unmodifiableSortedMap(SortedMap<K, ? extends V> m);

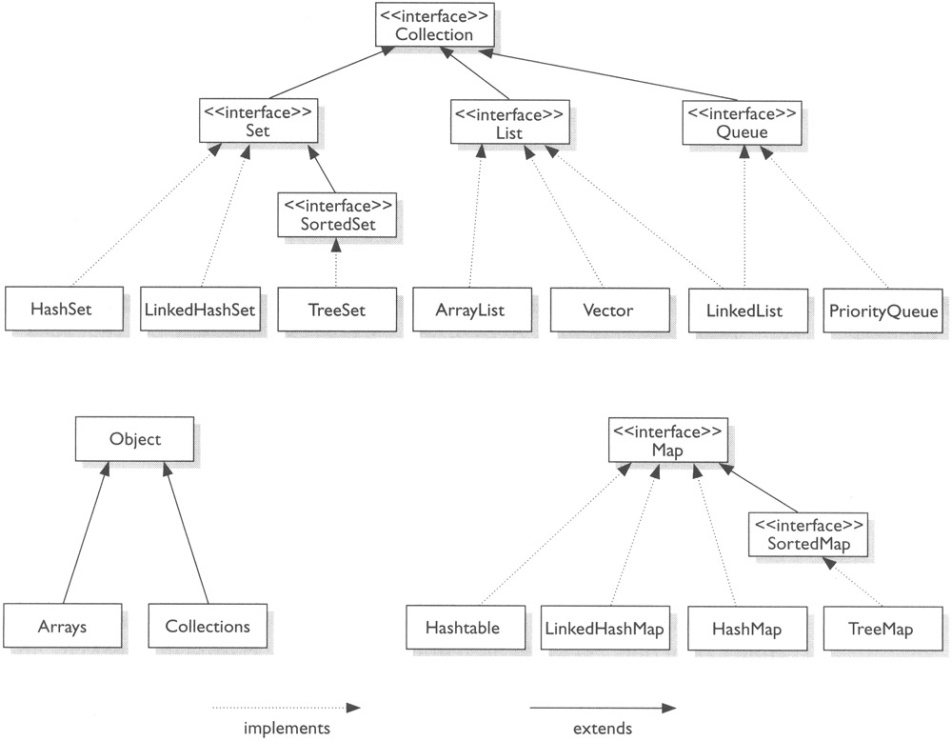
## Checked Interface Wrappers

The Collections.checked interface wrappers are provided for use with generic collections. These implementations return a dynamically type-safe view of the specified collection, which throws a ClassCastException if a client attempts to add an element of the wrong type. The generics mechanism in the language provides compile-time (static) type-checking, but it is possible to defeat this mechanism. Dynamically type-safe views eliminate this possibility entirely.

<https://docs.oracle.com/javase/tutorial/collections/implementations/index.html>

**XC:Collection**

1. **What is the Collections API?**
2. The Collections API is a set of classes and interfaces that support operations on collections of objects.

****



1. **What are the advantages of ArrayList over arrays?**
2. ArrayList can grow dynamically and provides more powerful insertion and search mechanisms than arrays.
3. **Why deletion in LinkedList is fast than ArrayList?**
4. The ArrayList class is a wrapper class for an array. It contains an inner array.

public ArrayList<T> {

private Object[] array;

private int size;

}

A LinkedList is a wrapper class for a linked list, with an inner node for managing the data.

public LinkedList<T> {

class Node<T> {

T data;

Node next;

Node prev;

}

private Node<T> first;

private Node<T> last;

private int size;

}

The ArrayList must move all the elements from array[index] to array[index-1] starting by the item to delete index. The LinkedList should navigate until that item and then erase that node by decoupling it from the list.

Because when you insert a new element and the array is full, you need to create a new array with more size (you can calculate the new size with a formula like 2 \* size or 3 \* size / 2). The LinkedList just add a new node next to the last.

1. **How do you decide when to use ArrayList and LinkedList?**
2. If you need to frequently add and remove elements from the middle of the list and only access the list elements sequentially, then LinkedList should be used. If you need to support random access, without inserting or removing elements from any place other than the end, then ArrayList should be used.
3. **What is a Values Collection View?**
4. It is a collection returned by the values() method of the Map Interface, It contains all the objects present as values in the map.

HashMap newmap = new HashMap();

// populate hash map

newmap.put(1, "tutorials");

newmap.put(2, "point");

newmap.put(3, "is best");

// checking collection view of the map

System.out.println("Collection view is: "+ newmap.values());

**Result:**

Collection view is: [tutorials, point, is best]

1. **What It Means If You Don't Override equals()**
   * 1. When you really need to know if two references are identical, use ==. But when you need to know if the objects themselves (not the references) are equal, use the equals() method.

The equals() method in class Object uses only the == operator for comparisons, so unless you override equals(), two objects are considered equal only if the two references refer to the same object.

Imagine you have a car, a very specific car (say, John's red Subaru Outback as opposed to Mary's purple Mini) that you want to put in a HashMap (a type of hashtable we'll look at later in this chapter), so that you can search on a particular car and retrieve the corresponding Person object that represents the owner. So you add the car instance as the key to the HashMap (along with a corresponding Person object as the value).

But now what happens when you want to do a search? You want to say to the HashMap collection, "Here's the car, now give me the Person object that goes with this car." But now you're in trouble unless you still have a reference to the exact object you used as the key when you added it to the Collection. *In other words, you can't make an identical Car object and use it for the search.*

The bottom line is this: if you want objects of your class to be used as keys for a hashtable (or as elements in any data structure that uses equivalency for searching for—and/or retrieving—an object), then you must override equals() so that two different instances can be considered the same. So how would we fix the car? You might override the equals() method so that it compares the unique VIN (Vehicle Identification Number) as the basis of comparison. That way, you can use one instance when you add it to a Collection, and essentially re-create an identical instance when you want to do a search based on that object as the key. Of course, overriding the equals() method for Car also allows the potential that more than one object representing a single unique car can exist, which might not be safe in your design. Fortunately, the String and wrapper classes work well as keys in hashtables—they override the equals() method. So rather than using the actual car instance as the key into the car/owner pair, you could simply use a String that represents the unique identifier for the car. That way, you'll never have more than

one instance representing a specific car, but you can still use the car—or rather, one of the car's attributes—as the search key.

1. public boolean equals(Object o) {

2. if ((o instanceof Moof) && (((Moof)o).getMoofValue()== this.moofValue)) {

3. return true;

4. } else {

5. return false;

6. }

7. }

Be sure that the object being tested is of the correct type! It comes in polymorphically as type Object, so you need to do an instanceof test on it. Having two objects of different class types be considered equal is usually not a good idea, but that's a design issue we won't go into here. Besides, you'd still have to do the instanceof test just to be sure that you could cast the object argument to the correct type so that you can access its methods or variables in order to actually do the comparison. Remember, if the object doesn't pass the instanceof test, then

you'll get a runtime ClassCastException.

***Remember that the*** equals()***,*** hashCode()***, and*** toString() ***methods are all*** public***. The following would not be a valid override of the*** equals() ***method, although it might appear to be if you don’t look closely enough during the exam:***

class Foo { boolean equals(Object o) { } } 🡪 Default equal method is not valid

***And watch out for the argument types as well. The following method is an overload, but not an override of the*** equals() ***method:***

class Boo { public boolean equals(Boo b) { } } 🡪 Object type Boo makes it overloaded

1. **What are equals() contract?**
2. Here are the contracts:
3. It is **reflexive**. For any reference value x, x.equals(x) should return true.
4. It is **symmetric**. For any reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
5. It is **transitive**. For any reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) must return true.
6. It is **consistent**. For any reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the object is modified.
7. For any non-null reference value x, x.equals(null) should return false.
8. **What It Means If You Don't Override Hashcode()**
9. Hashcodes are typically used to increase the performance of large collections of data. The hashcode value of an object is used by some collection classes. It is kind of an object ID number, it isn't necessarily unique. Collections such as HashMap and HashSet use the hashcode value of an object to determine how the object should be *stored* in the collection, and the hashcode is used again to help *locate* the object in the collection.

**Understanding Hashcodes**

In order to understand what's appropriate and correct, we have to look at how some of the collections use hashcodes.

Imagine a set of buckets lined up on the floor. Someone hands you a piece of paper with a name on it. You take the name and calculate an integer code from it by using A is 1, B is 2, and so on, and adding the numeric values of all the letters in the name together.

Key Hashcode Algorithm Hashcode

Alex A(1) + L(12) + E(5) + X(24) = 42

Bob B(2) + O(15) + B(2) = 19

Dirk D(4) + I(9) + R(18) + K(11) = 42

Fred F(6) + R(18) + E(5) + D(4) = 33

We don't introduce anything random, we simply have an algorithm that will always run the same way given a specific input, so the output will always be identical for any two identical inputs. So far so good? Now the way you use that code (and we'll call it a hashcode now) is to determine which bucket to place the piece of paper into (imagine that each bucket represents a different code number you might get). Now imagine that someone comes up and shows you a name and says, "Please retrieve the piece of paper that matches this name." So you look at the name they show you, and run the same hashcode-generating algorithm. The hashcode tells you in which bucket you should look to find the name. You might have noticed a little flaw in our system, though. Two different names might result in the same value. For example, the names Amy and May have the same letters, so the hashcode will be identical for both names. That's acceptable, but it does mean that when someone asks you (the bucket-clerk) for the Amy piece of paper, you'll still have to search through the target bucket reading each name until we find Amy rather than May. The hashcode tells you only which bucket to go into, but not how to locate the name once we're in that bucket.

***1. Find the right bucket (using*** hashCode()***)***

***2. Search the bucket for the right element (using*** equals() ***).***

So for efficiency, your goal is to have the papers distributed as evenly as possible across all buckets. Ideally, you might have just one name per bucket so that when someone asked for a paper you could simply calculate the hashcode and just grab the one paper. The least efficient (but still functional) hashcode generator would return the same hashcode (say, 42) regardless of the name, so that all the papers landed in the same bucket while the others stood empty.

When you put an object in a collection that uses hashcodes, the collection uses the hashcode of the object to decide in which bucket/slot the object should land. As long as the object (stored in the collection, like a paper in the bucket) you're trying to search for has the same hashcode as the object you're using for the search (the name you show to the person working the buckets), then the object will be found.

Now can you see why if two objects are considered equal, their hashcodes must also be equal? Otherwise, you'd never be able to find the object since the default hashcode method in class Object virtually always comes up with a unique number for each object, even if the equals() method is overridden in such a way that two or more objects are considered equal. It doesn't matter how equal the objects are if their hashcodes don't reflect that. So one more time: If two objects are equal, their hashcodes must be equal as well.

**Implementing hashCode()**

Your hashCode()implementation should use the same instance variables. Here's an example:

class HasHash {

public int x;

HasHash(int xVal) { x = xVal; }

public boolean equals(Object o) {

HasHash h = (HasHash) o;//Don't try at home without instanceof test

if (h.x == this.x) {

return true;

} else {

return false;

}

}

public int hashCode() { return (x \* 17); }

}

This equals() method says two objects are equal if they have the same x value, so objects with the same x value will have to return identical hashcodes.

1. **What are hashCode() contract?**
2. Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode() method must consistently return the same integer, provided no information used in equals() comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
3. If two objects are equal according to the equals(Object) method, then calling the hashCode() method on each of the two objects must produce the same integer result.
4. It is NOT required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode() method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables.
5. **Explain TreeSet?**

**A:**  TreeSet provides an implementation of the Set interface that uses a tree for storage. Objects are stored in sorted, ascending order.

**import** java.util.Iterator;

**import** java.util.TreeSet;

**public** **class** TreeSetExample {

**public** **static** **void** main(String[] args) {

System.out.println("Tree Set Example!**\n**");

TreeSet<Integer> tree = **new** TreeSet<Integer>();

tree.add(12);

tree.add(63);

tree.add(34);

tree.add(45);

*// here it test it's sorted, 63 is the last element. see output below*

Iterator<Integer> iterator = tree.iterator();

System.out.print("Tree set data: ");

*// Displaying the Tree set data*

**while** (iterator.hasNext()) {

System.out.print(iterator.next() + " ");

}

System.out.println();

*// Check empty or not*

**if** (tree.isEmpty()) {

System.out.print("Tree Set is empty.");

} **else** {

System.out.println("Tree Set size: " + tree.size());

}

*// Retrieve first data from tree set*

System.out.println("First data: " + tree.first());

*// Retrieve last data from tree set*

System.out.println("Last data: " + tree.last());

**if** (tree.remove(45)) { *// remove element by value*

System.out.println("Data is removed from tree set");

} **else** {

System.out.println("Data doesn't exist!");

}

System.out.print("Now the tree set contain: ");

iterator = tree.iterator();

*// Displaying the Tree set data*

**while** (iterator.hasNext()) {

System.out.print(iterator.next() + " ");

}

System.out.println();

System.out.println("Now the size of tree set: " + tree.size());

*// Remove all*

tree.clear();

**if** (tree.isEmpty()) {

System.out.print("Tree Set is empty.");

} **else** {

System.out.println("Tree Set size: " + tree.size());

}

}

}

Tree Set Example!

Tree set data: 12 34 45 63

Tree Set size: 4

First data: 12

Last data: 63

Data is removed from tree set

Now the tree set contain: 12 34 63

Now the size of tree set: 3

Tree Set is empty.

1. **What is Comparable Interface?**
2. When we say Comparable (please note the ‘able’ in the word), it actually means that the interface enables or gives the classes (which implements it) the ability to compare their instances with each other.

Also, the interface provides a method compareTo which takes a parameter T and returns an integer. This means that if a class implements this interface, it has to provide implementation for the compareTo method as well. Now the logic inside can be anything, anything at all. You are free to write your own comparison logic and say how the object makes a comparison with other objects of the same kind (I say same kind because, its pointless to compare two things of different kind, something like comparing a train and a dog might not make sense).

<http://techieme.in/comparable-and-comparato/>

1. **What are various methods in queue interface?**

|  |  |
| --- | --- |
| **Insert** | |
| **add** | **offer** |
| boolean add(E e) Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning true upon success and throwing an IllegalStateException if no space is currently available. **Specified by:** add in interface Collection<E> Parameters: e - the element to add **Returns:** true (as specified by Collection.add(E)) **Throws:** IllegalStateException - if the element cannot be added at this time due to capacity restrictions ClassCastException - if the class of the specified element prevents it from being added to this queue NullPointerException - if the specified element is null and this queue does not permit null elements IllegalArgumentException - if some property of this element prevents it from being added to this queue | boolean offer(E e) Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions. When using a capacity-restricted queue, this method is generally preferable to add(E), which can fail to insert an element only by throwing an exception. **Parameters:** e - the element to add **Returns:** true if the element was added to this queue, else false **Throws:** ClassCastException - if the class of the specified element prevents it from being added to this queue NullPointerException - if the specified element is null and this queue does not permit null elements IllegalArgumentException - if some property of this element prevents it from being added to this queue |
| **Delete** | |
| **remove** | **poll** |
| E remove() Retrieves and removes the head of this queue. This method differs from poll only in that it throws an exception (NoSuchElementException) if this queue is empty. | E poll() Retrieves and removes the head of this queue, or returns null if this queue is empty. |
| **Read** | |
| **element** | **peek** |
| E element() Retrieves, but does not remove, the head of this queue. This method differs from peek only in that it throws an exception(NoSuchElementException) if this queue is empty. | E peek() Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty. |

# Collections

How you write a LinkedList program?

<http://crunchify.com/how-to-implement-a-linkedlist-class-from-scratch-in-java/>

How does get(Key key) method works internally in HashMap, and Hashtable in Java?  
  
<http://java67.blogspot.com/2013/06/how-get-method-of-hashmap-or-hashtable-works-internally.html#ixzz45ioG9Rsd>

<https://tekmarathon.com/2013/03/11/creating-our-own-hashmap-in-java/>

1. What is blocked queue in jdk 5?
2. How put works in hashmap?
3. What is concurenthashmap?

<http://crunchify.com/hashmap-vs-concurrenthashmap-vs-synchronizedmap-how-a-hashmap-can-be-synchronized-in-java/>

1. How to write own concurrenthash map?
2. What are copytowrite\* collection class?
3. What is ConcurrentModificationException?
4. How can you write a code which will create ConcurrentModificationException?

<https://examples.javacodegeeks.com/java-basics/exceptions/java-util-concurrentmodificationexception-how-to-handle-concurrent-modification-exception/>

1. What is difference between collection.remove() and iteration.remove()?
2. What are differences between hashmap, concurrenthashmap and synchronizedhashmap?
3. How concurrent collections works? How to write own concurrent collections?
4. Fail Fast Vs Fail Safe Iterator In Java

<http://javahungry.blogspot.com/2014/04/fail-fast-iterator-vs-fail-safe-iterator-difference-with-example-in-java.html>