# JAVA COLLECTIONS FRAMEWORK

# <https://www.codejava.net/java-core/collections/java-list-collection-tutorial-and-examples>

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ID | INTERFACE |  | IMPLEMENTATION |  |
|  | **LIST** | | **ArrayList** | * It is based on array. It expands automatically. * Useful for random access and retrieving elements |
|  | **LinkedList** | * It is based on doubly linkedList. * Useful for fast insertion in the beginning or the end |
|  | **SET** | | **HashSet** | * Does not have duplicates * Elements unsorted * Allows nulls (only, because it [Set]) * HashSet is the most common |
|  | **LinkedHashSet** | * Does not have duplicates * Elements sorted in an order of insertion * Allows nulls (only, because it [Set]) |
|  | **TreeSet** | * Does not have duplicates * Elements sorted in a natural order * Does not allow nulls |
|  | **Queue** | **Queue** | **LinkedList** | LinkedList - index this class implements both List and Deque interface, thus having hybrid characteristics and behaviors of list and queue. Consider using a LinkedList when you want fast adding and fast removing elements at both ends |
|  | **ArrayDeque** | is more efficient than LinkedList. It is much better in term of performance |
|  | **PriorityQueue** | this queue orders elements according to their natural ordering, or by a Comparator provided at construction time |
|  | **Dequeue** | **LinkedList** | It is implementation from [Deque] – double ended interface. The same as above, but has method like  [addFirst],[addLast], ...  [offerFirst], [offerLast], ... |
|  | **ArrayDeque** | It is implementation from [Deque] – double ended interface. The same as above, but has method like  [addFirst],[addLast], ...  [offerFirst], [offerLast], ... |
|  | **MAP** | | **HashMap** | It uses a hash table as the underlying data structure. It implements all of the Map operations and allows null values and one null key. This class is roughly equivalent to Hashtable - a legacy data structure before Java Collections Framework, but it is not synchronized and permits nulls. HashMap does not guarantee the order of its key-value elements. Therefore, consider to use a HashMap when order does not matter and nulls are acceptable. |
|  | **LinkedHashMap** | this implementation uses a hash table and a linked list as the underlying data structures, thus the order of a LinkedHashMap is predictable, with insertion-order as the default order. This implementation also allows nulls like HashMap. So consider using a LinkedHashMap when you want a Map with its key-value pairs are sorted by their insertion order. |
|  | **TreeMap** | this implementation uses a red-black tree as the underlying data structure. A TreeMap is sorted according to the natural ordering of its keys, or by a Comparator provided at creation time. This implementation does not allow nulls. So consider using a TreeMap when you want a Map sorts its key-value pairs by the natural order of the keys (e.g. alphabetic order or numeric order), or by a custom |

# • HashMap:

# • LinkedHashMap

# • TreeMap: order you specify.ARRAYS

**Array** – is object that stores collection of values

Array can store 2 types of data:

* **A collection of primitive values** –
* **A collection of objects** – in fact it’s heap memory

Creating array involves 3 steps:

* **declaration** –

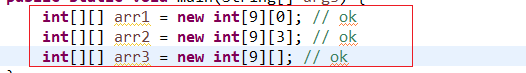
int[] arr1; - one dimensional array

int[][] arr2; - multidimensional array

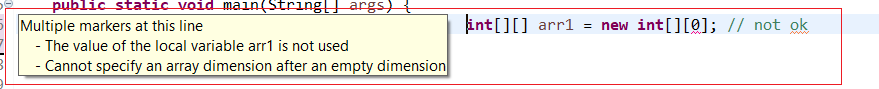
int[] arr2[]; - multidimensional array (another form)

* **allocation**

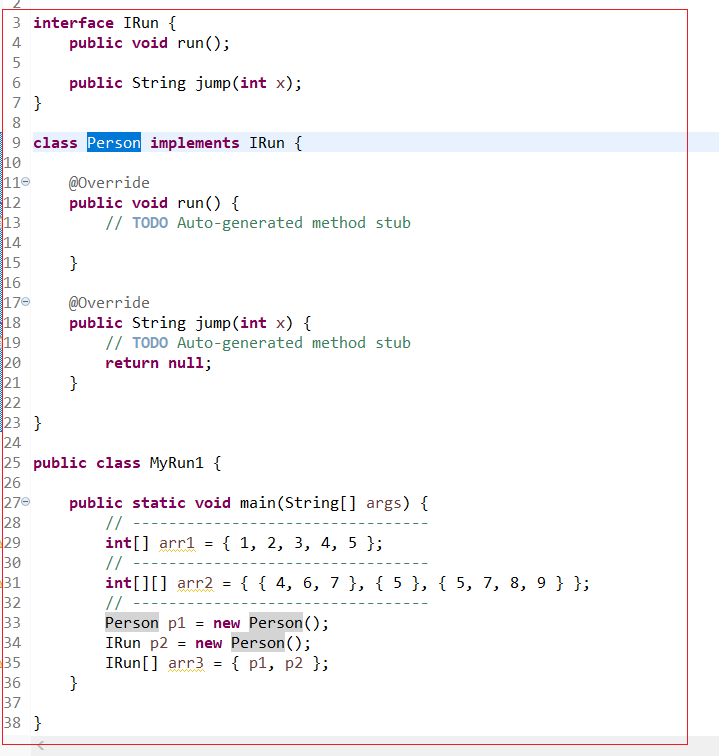
*example of right declaration:*



*example of not right declaration:*



* **initialization**



# ARRAYLIST

Array can’t change size once created. ArrayList does it automatically

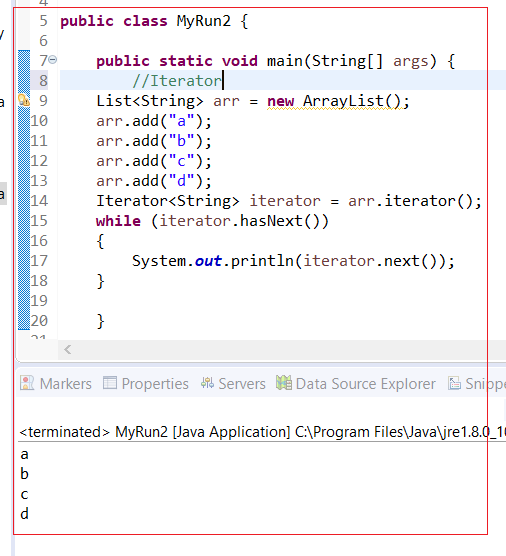
* Arraylist cannot contain primitives



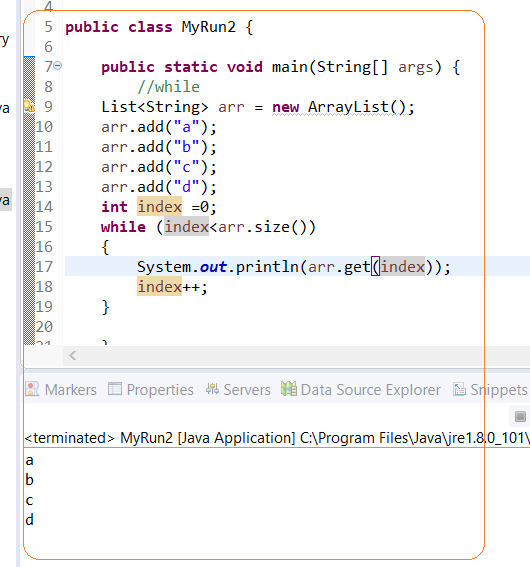
There are 5 ways to iterate through loop:

* **For Loop** -
* **ForEach** -
* **Iterator** -
* **While Loop** -
* **Collections’s stream() util (Java8)** –

**ITERATOR**

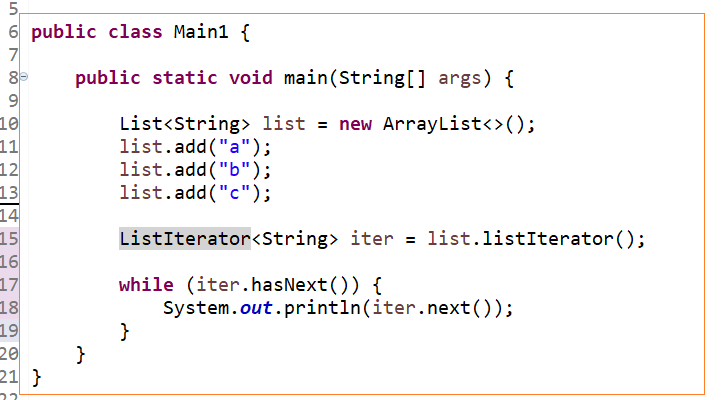


**WHILE**



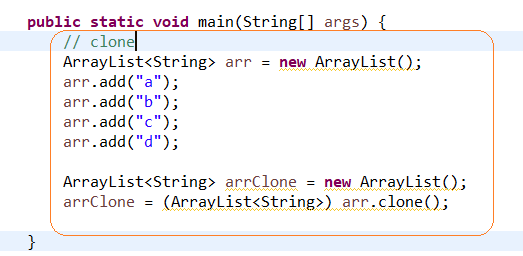
### **ITERATOR vs FOR LOOP**

In for-each loop, we can’t modify collection, it will throw a ConcurrentModificationException on the other hand with iterator we can modify collection.



**CLONE**

**Cloning –** it does not clone Object. It clones only reference of elements of object



# GENERICS

## OIVERVIEW

* Before java 5 you had to write code like this



and hope that programmers remember that you wanted only String

* In Java 5 you could parametrized types



* when Java 7 came you could use shorten form



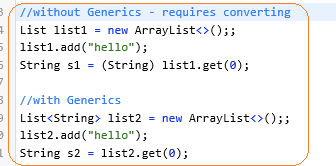
**Diamond operator** - The shorten form <> is called diamond (=the pair of angle brackets)

## MAIN

**Generics** are features of the Java programming language that allow programmers to write parameterized code.

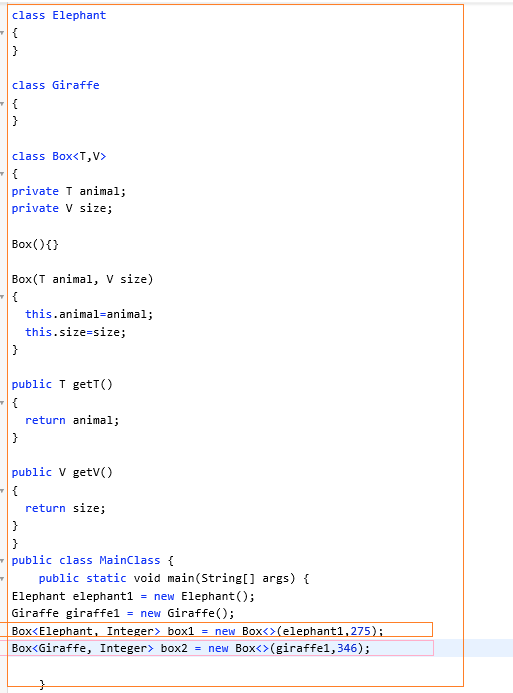
**Generics** – is similar like a template. It allows type to be a parameter to method, class or interface. Generic provides compile time checking and removing risk of [**ClassCastException**].

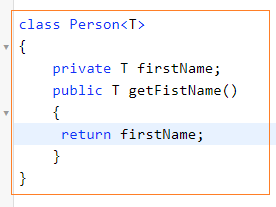
* *It allows to avoid run time erro*r **[ClassCastException**] *and if there mistake throws error at compile step*
* *We don’t need to make additional casting*

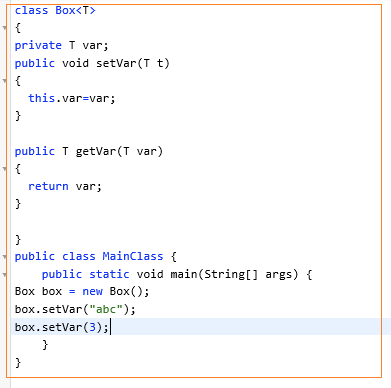


* *Help to reuse code*

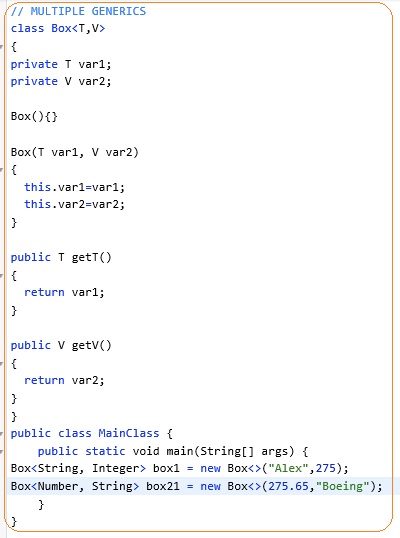
note: in example below you will not need to create Box for each animal. You can reuse the same Box – just parametrize







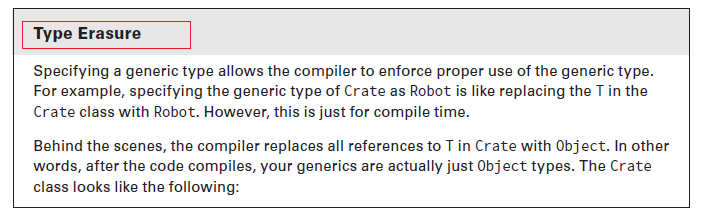
**Multiple generics types**

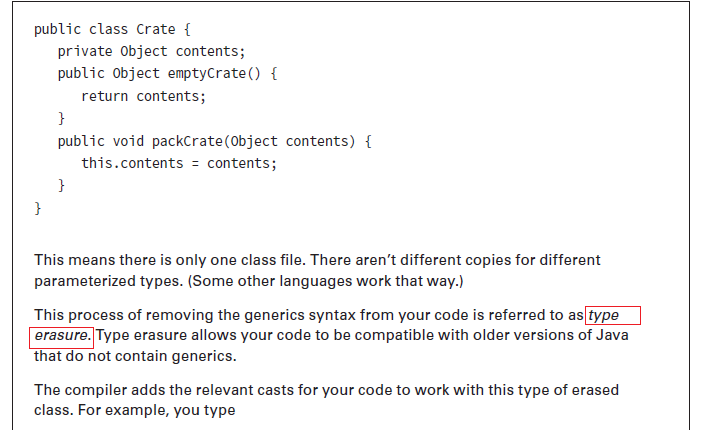
****

## ERAUSERE

**Type Erasure** – when compiler compiles your code it replaces all references (like <T>) to Object class with necessary casting. It’s done to be compatible with early versions of Java when there were not generics.

Type erasure is a process in which compiler replaces a generic parameter with actual class or bridge method. Type erasure is a process to these types and map it to raw type in byte code and it is done during compilation by the Java compiler





## GENERIC BOUNDS

|  |  |  |
| --- | --- | --- |
| TYPE OF BOUND | SYNTAX | EXAMPLE |
| **UNBOUNDED** | **<?>** | List<?> list = new ArrayList<String>(); |
| **UPPER BOUNDED** | **<? extends type>** | List<? extends Number> list = new ArrayList<Integer>(); |
| **LOWER BOUNDED** | **<? Super type>** | List<? super Exception> list = new ArrayList<Object>(); |

*There may be times when you'll want to restrict the kinds of types that are allowed to be passed to a type parameter. For example, a method that operates on numbers might only want to accept instances of Number or its subclasses. This is what bounded type parameters are for. To declare a bounded type parameter, list the type parameter's name, followed by the extends keyword, followed by its upper bound.*

UPPER BOUNDED:



LOWER BOUNDED:



It will work for Integer->Number-\_Object

# COLLECTIONS

**Java Collection Frameworks** provides common data structures implementations which are enough for general-purpose such as list, set, map, queue, tree, etc. These collections are high-performance, high-quality, and easy to use with very good documentation. The Java Collections Framework is a collection of classes and interfaces that helps in storing and processing data efficiently.

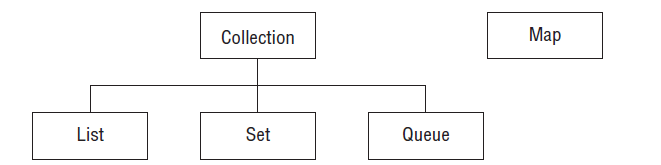
**Collections interfaces:**

* List – ordered collection of elements that allows duplicates. Its’s oreder in that way that they are added
* Set- collection of unique elements. Allows one null (except TreeSet implementation. HashSet and LinkedHaashSet allow null)

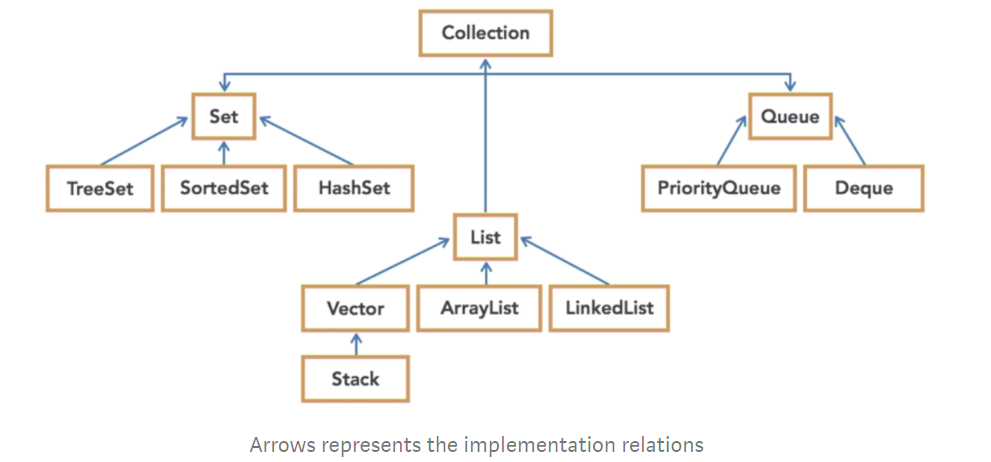
Can loop through [iterate] or [forEach] or [stream]. Just [for loop] is not supported

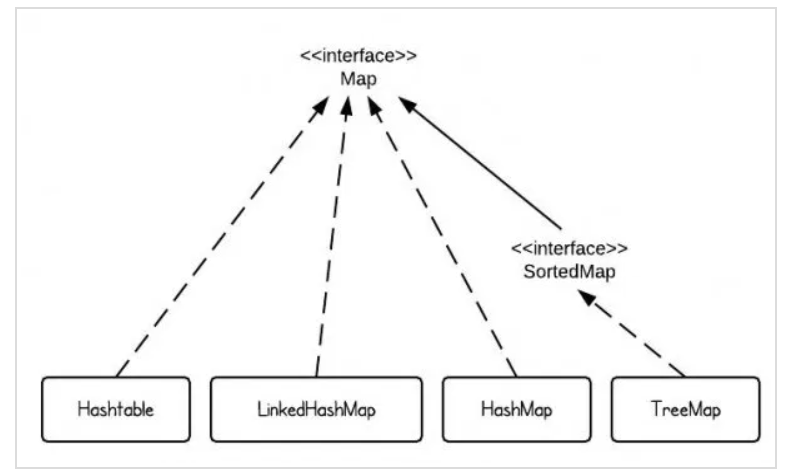
* Queue – is a collection that process elements in a specific order. Typical order is FIFO
* Map – set of key/values pairs. Key has to be unique

##### **Collection interfaces**



##### **Collection interfaces and implementations**





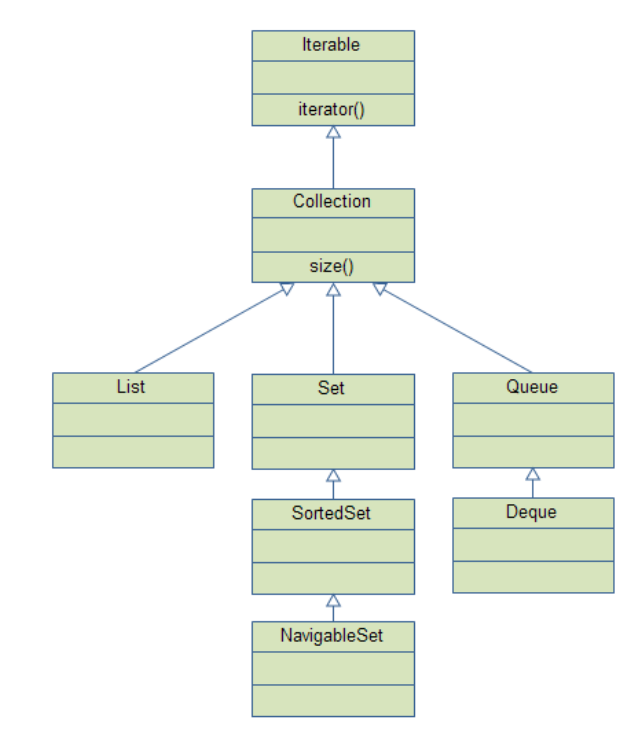
#### **What is the difference between Collection and Collections in java?**

* **Collection** is a root level interface of the Java Collection Framework. Most of the classes in Java Collection Framework inherit from this interface. List, Set and Queue are main sub interfaces of this interface.
* **Collections** is an utility class in java.util package. It consists of only static methods which are used to operate on objects of type Collection: Collections.max(), Collections.min(), Collections.shuffle()

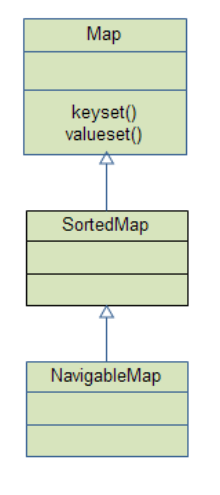
There are 2 group of interfaces

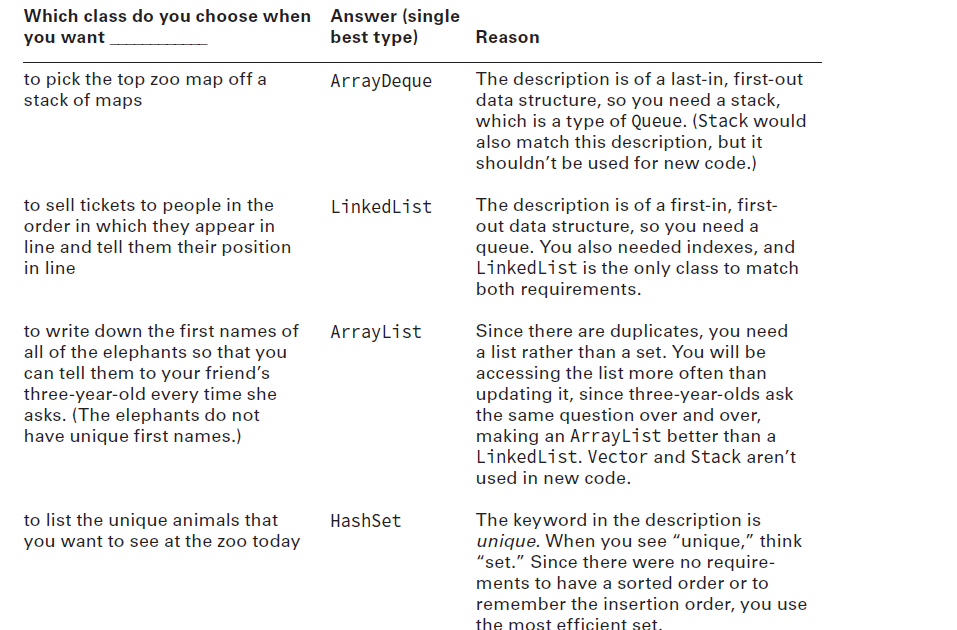
* Collection
* Map

COLLECTION interface



MAP interface





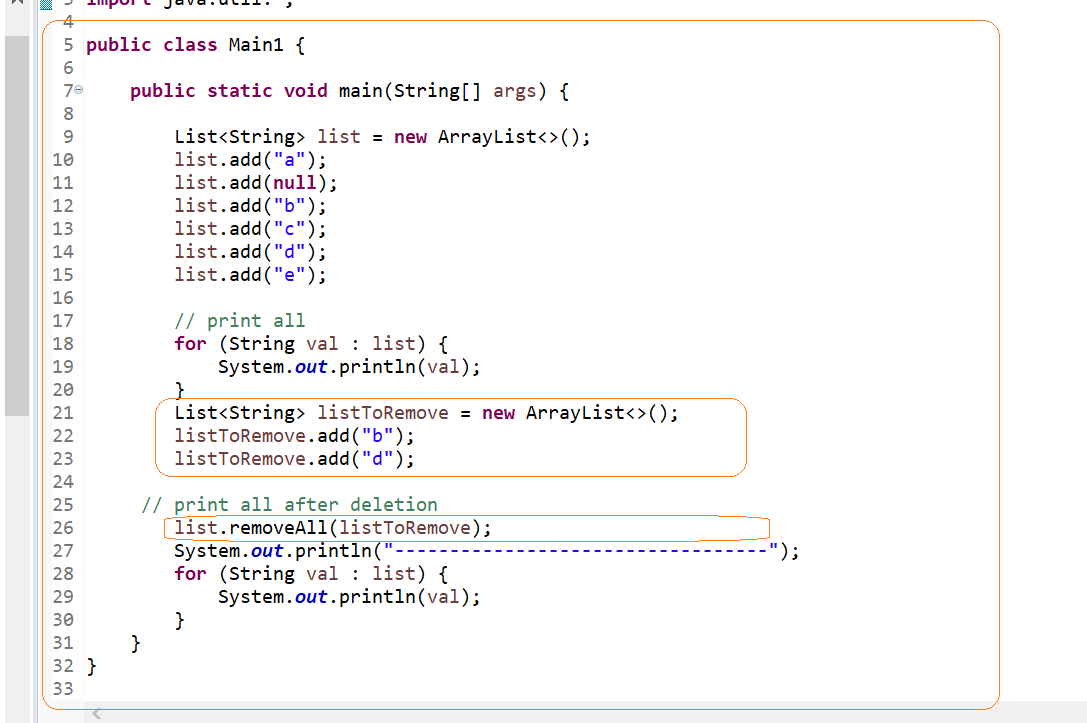
**Collection methods**

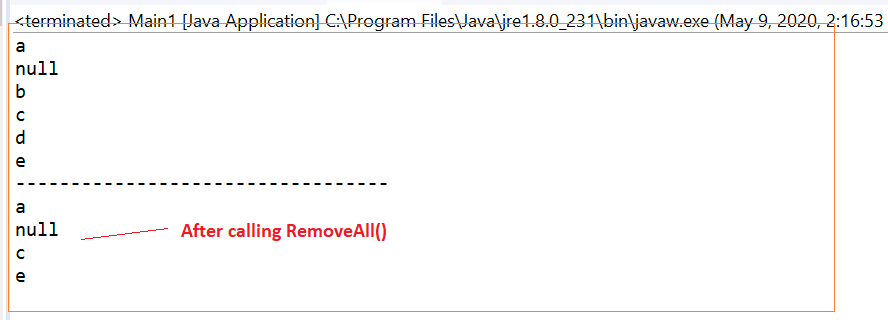
|  |  |
| --- | --- |
| Method | Description |
| add() | It adds elements and returns boolean |
| remove() | It removes elements and returns Boolean: true if elements have been removed |
| removeAll | Remove all elements from a defined list |
| isEmpty() |  |
| size() |  |
| contains() |  |
| indexOf | Find index by element |
| clear() | Remove all elements |

##### **CLEAR vs REMOVEALL()**

Clear() – remove all

RemoveAll() – remove elements from an argument list





**Map methods**

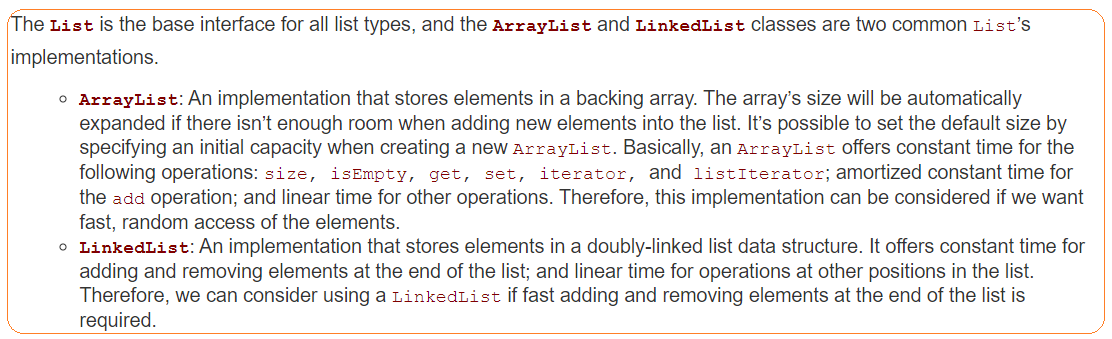
|  |  |
| --- | --- |
| Method | Description |
| put() | Add pair. For example, map1.put(”one”,”Alex”); |
|  | It removes elements and returns Boolean: true if elements have been removed |
|  |  |
|  |  |
|  |  |
|  |  |
|  | Find index by element |
|  | Remove all elements |

## LIST INTERFACE

* List – ordered collection of elements(*in insertion order of elements*) that allows duplicates and nulls . Its’s ordered in that way that they are added

Implementations

* ArrayList
* LinkedList



## ArrayList

* You can look up any element in constant time
* It allows dups
* It allows nulls
* Arraylist cannot contain primitives
* Adding and removing are slower than accessing elements



## LinkedList

LinkedList –is double linked list. It implements both List and Queue

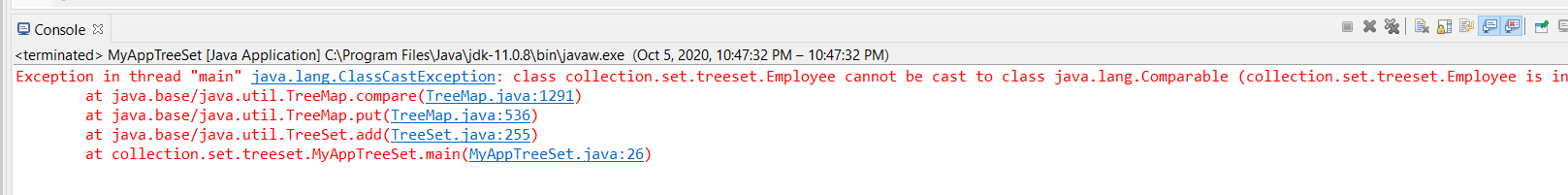
The main benefit of [LinkedList] is that you can [add, [delete, [remove] elements in the beginning and the end in the same time

## Vector

Vector is legacy

## SET INTERFACE

* Set –collection of elements that does not have duplicates
* Can loop through [iterate] or [forEach]. Just [for loop] is not supported
* You can’t sort TreeSet or LinkedHashSet. You need to convert it to List interface
* Does not guarantee order for HashSet (most common case)
* Guarantee order for [LinkedHashSet] in order of inserting values and [TreeSet] natural order
* **(!) for custom classes** you need to implement <Comparable> interface or implement custom <Comparator>. It needs in order to implement [compare] method



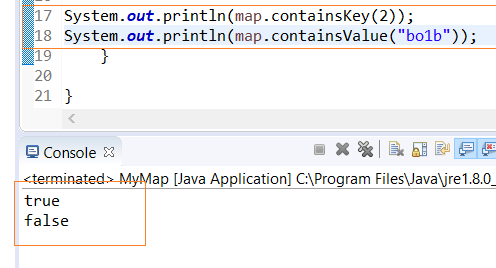
Note:

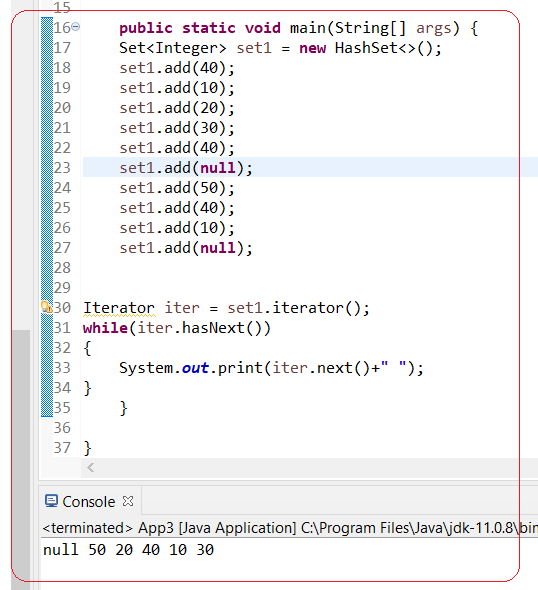
* if you put dups it will return unique
* If you put null it returns error

Implementations

* HashSet
* LinkedHashSet
* TreeSet

Useful methods:





## HashSet

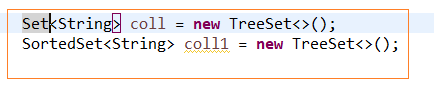
* **HashSet stores elements in hash table. It means it uses hashcode()**
* No guarantee on order
* Allows one null
* HashSet is the most common

## LinkedHashSet

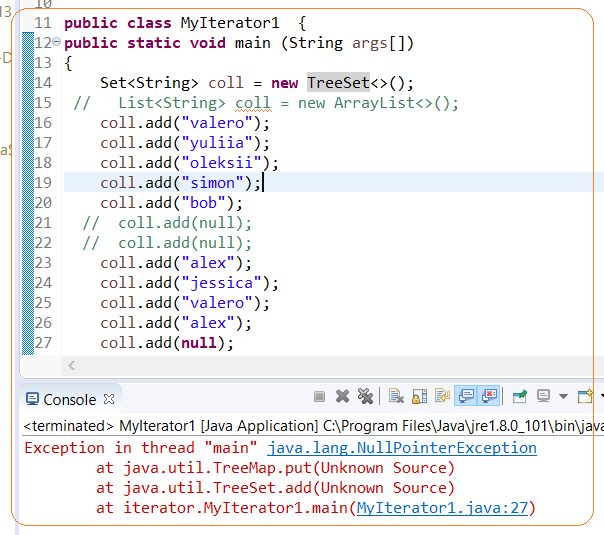
* Elements ordered in that order that inserted in collection
* Allows one null

## TreeSet

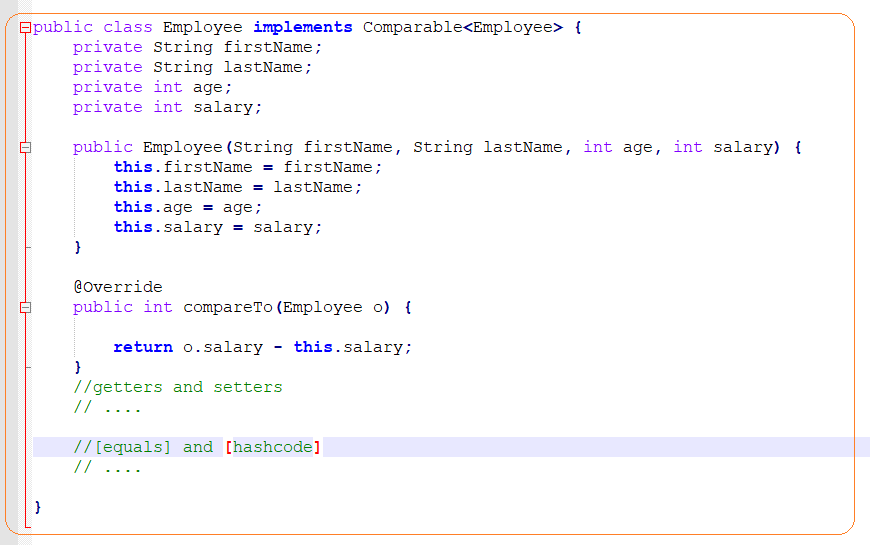
* TreeSet is implemennted from interface [SortedSet]

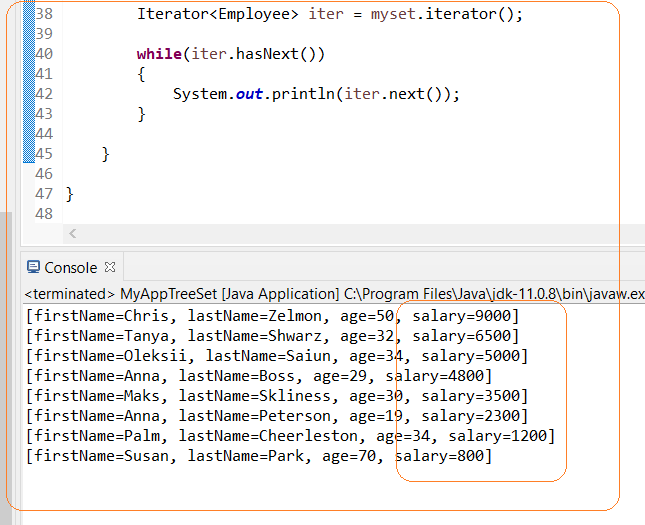
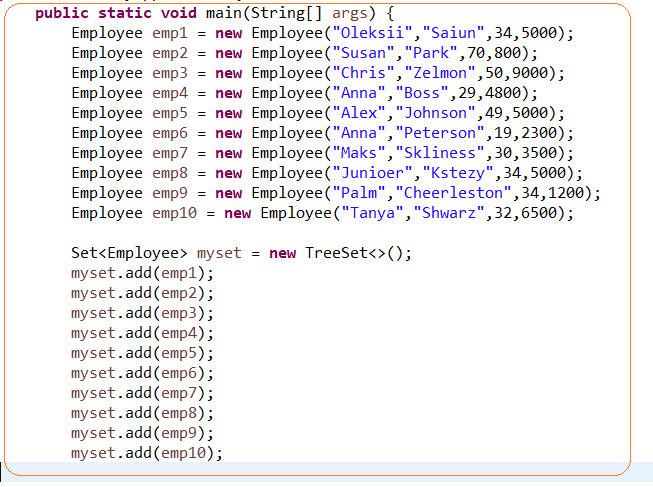


* Ordered by ascending (natural sorting order)
* **Does not allow nulls**.



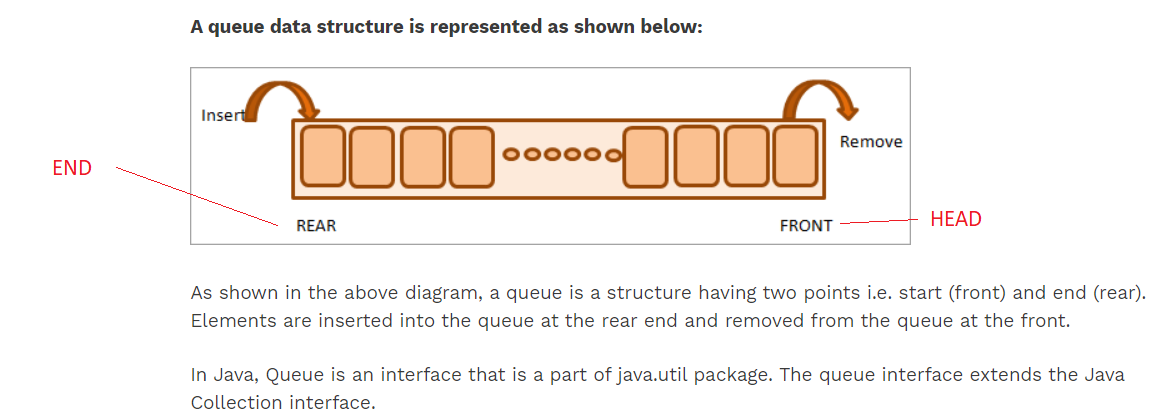
## CUSTOM CLASS FOR [SET] INTERFACE

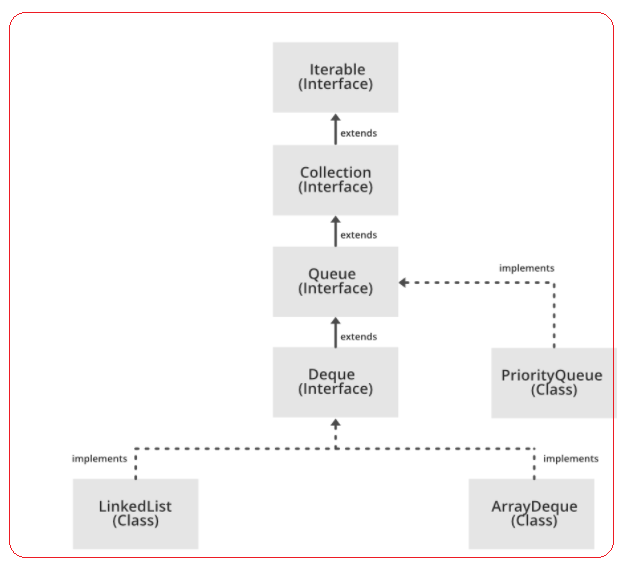




## QUEUE INTERFACE

**Queue** –is infrastructure designed to have elements inserted at the end and removed from the beginning of the queue



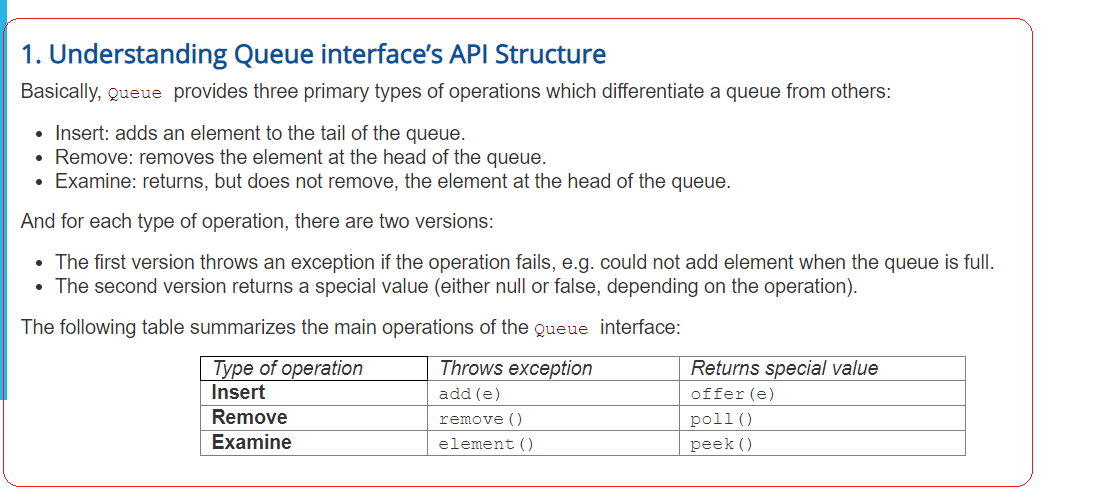


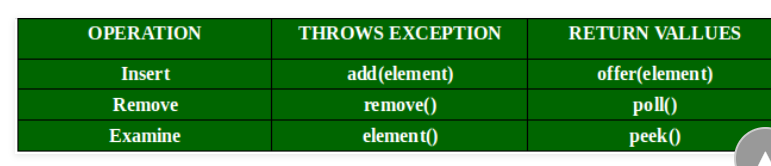
Note: Queue is a super

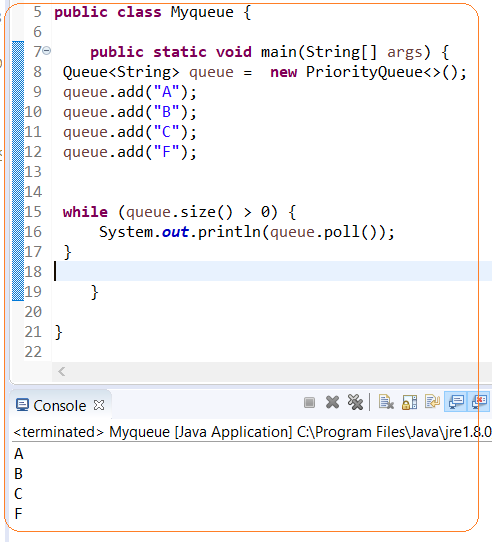
Implementation

* LinkedList - this class implements both List and Deque interface, thus having hybrid characteristics and behaviors of list and queue. Consider using a LinkedList when you want fast adding and fast removing elements at both ends, plus accessing elements by index
* PriorityQueue - this queue orders elements according to their natural ordering, or by a Comparator provided at construction time.
* ArrayDeque –is more efficient than LinkedList. It is much better in term of performance

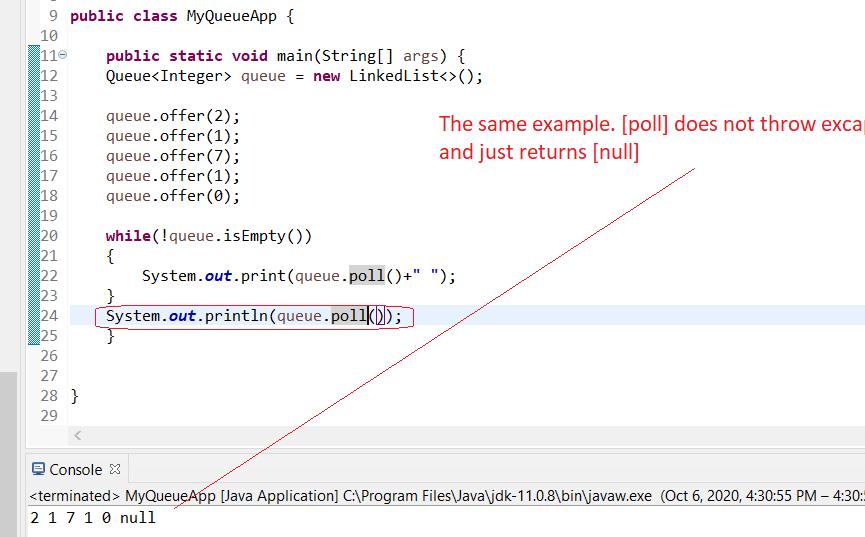
METHODS

****

****

****

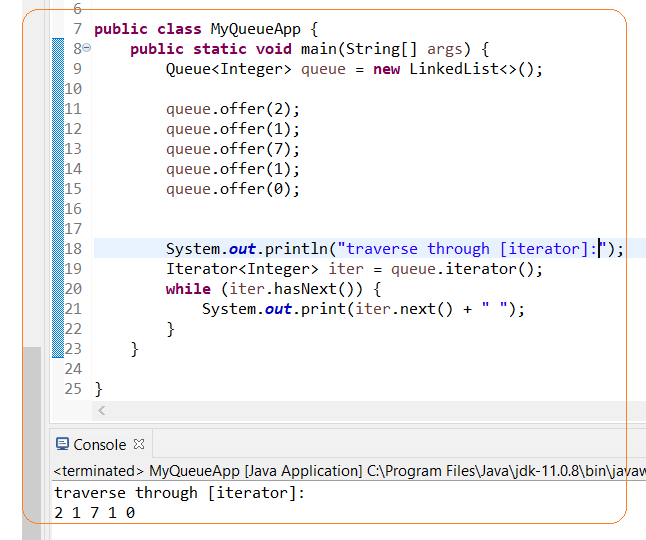
****

****

## TRAVERSE

#### TRAVERSE - [FOR LOOP]

#### TRAVERSE - [ITERATOR]



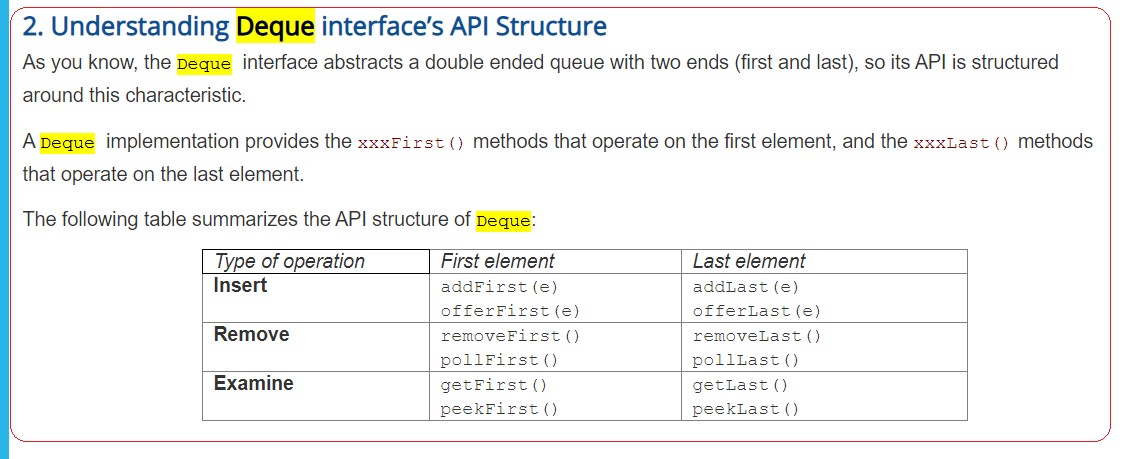
#### TRAVERSE – REMOVE ELEMENTS EITHER CALLING [POLL] OR [REMOVE] METHOD

## DEQUEUE INTERFACE

**Dequeue** –is Double ended queue. It implements FIFO and LIFO. So it has additional methods like [addLast],[addFirst] (instead of just one [add]) and so on…

Implementation

* LinkedList
* ArrayDequeue



**NOTE**

**WHEN FIFO: offer/poll/peek (queue)**

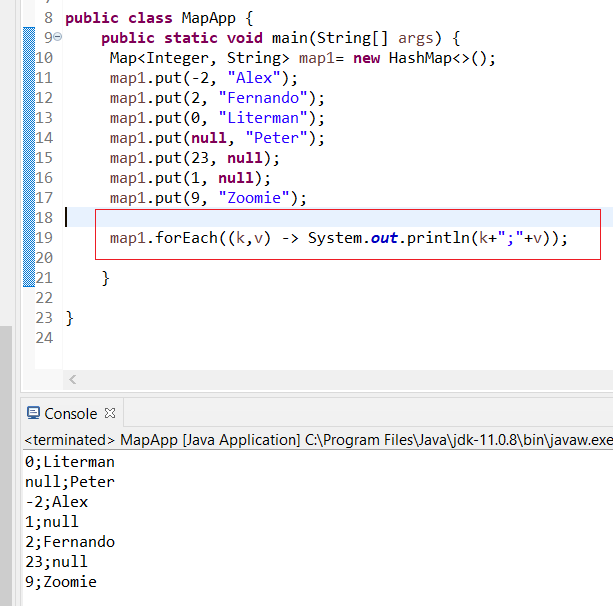
**WHEN LIFO: push/poll/peek (stack)**

## **MAP INTERFACE**

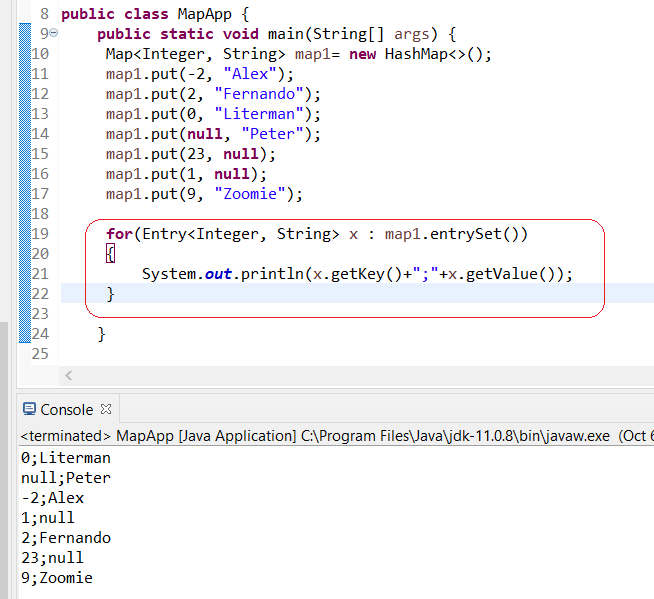
Map is not considered to be a true collection, as the Map interface does not extend the Collection interface. Instead, it starts an independent branch in the Java Collections Framework

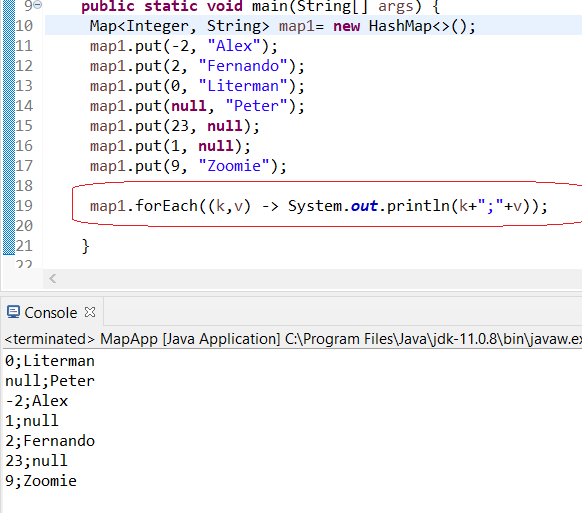
* EntrySet Api
* Iterator
* Lambda
* Stream API

**Example1. Traversal for each**



**Traversal EntrySet API**

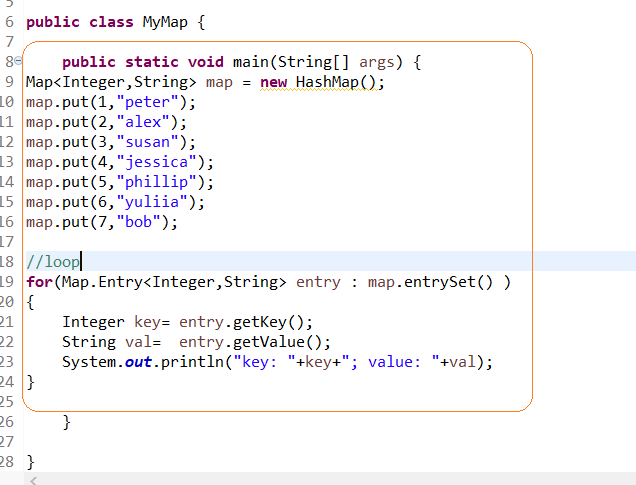
****

****

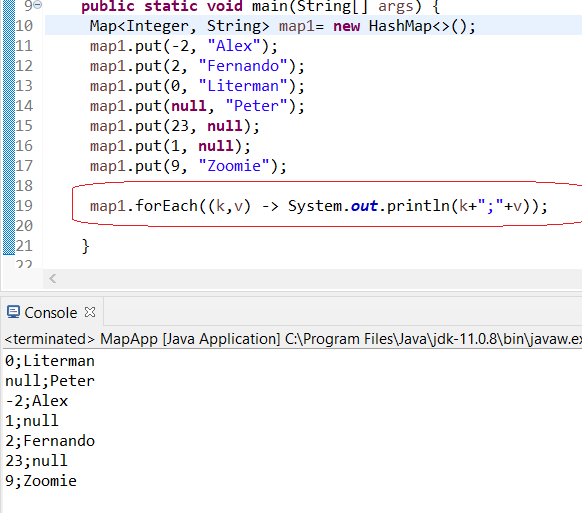
Map implementations

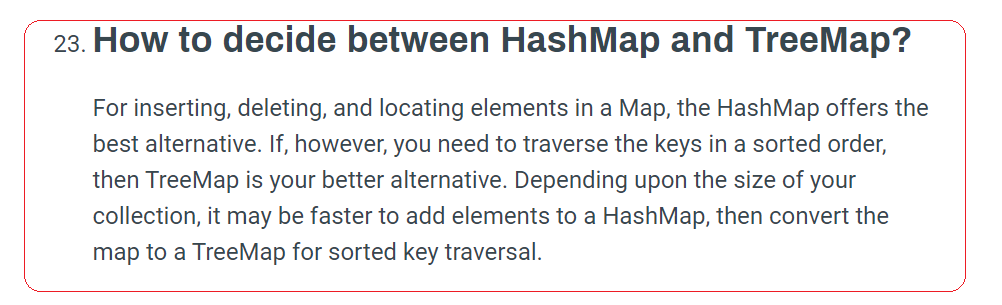
* **HashMap**: this implementation uses a hash table as the underlying data structure. It implements all of the Map operations and allows null values and one null key. This class is roughly equivalent to Hashtable - a legacy data structure before Java Collections Framework, but it is not synchronized and permits nulls. HashMap does not guarantee the order of its key-value elements. Therefore, consider to use a HashMap when order does not matter and nulls are acceptable.
* **LinkedHashMap**: this implementation uses a hash table and a linked list as the underlying data structures, thus the order of a LinkedHashMap is predictable, with insertion-order as the default order. This implementation also allows nulls like HashMap. So consider using a LinkedHashMap when you want a Map with its key-value pairs are sorted by their insertion order.
* **TreeMap**: this implementation uses a red-black tree as the underlying data structure. A TreeMap is sorted according to the natural ordering of its keys, or by a Comparator provided at creation time. This implementation does not allow nulls. So consider using a TreeMap when you want a Map sorts its key-value pairs by the natural order of the keys (e.g. alphabetic order or numeric order), or by a custom order you specify.

**Loop Map through Iterator:**



**Loop Map through Lambda:**

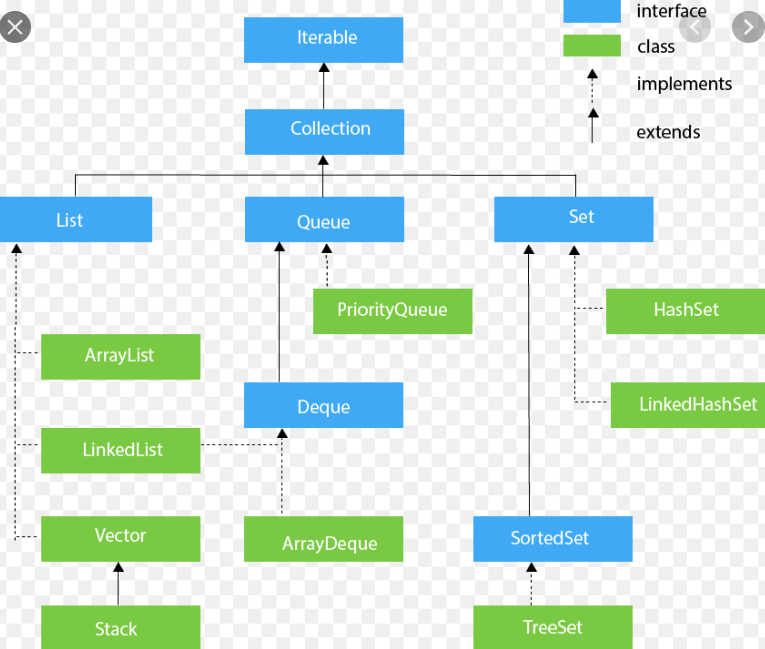




## **JAVA ITERABLE**

**Iterable interface** – is top of Collection hierarchy. Collection extends [Iterable] interface

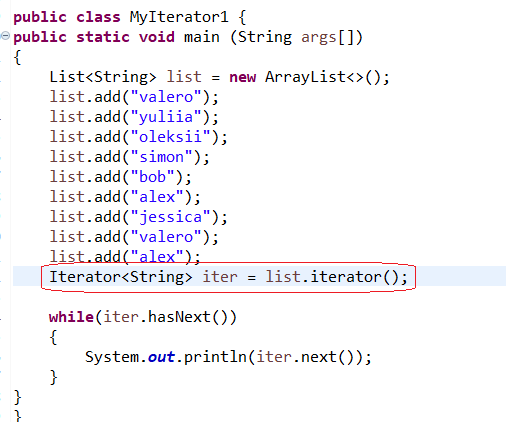
You don’t need explicitly to implements this interface. Java takes care of it behind the scene



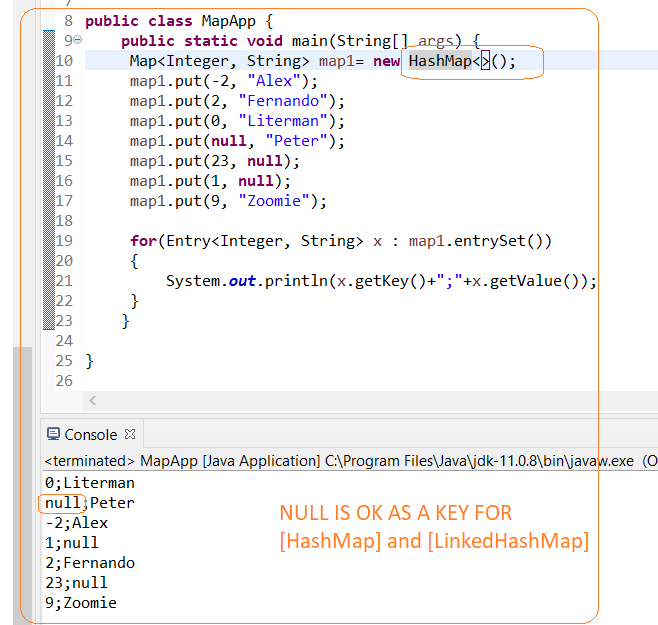
## JAVA ITERATOR

**Iterator** – is one of the oldest mechanism in Java to iterate collections

Standard Java Collection contains a method called [iterator()].



### **NULLS. IMPLEMENTATIONS [HASHMAP] AND [LINKEDHASHMAP] ALLOWS NULLS AS A KEY (ONLY ONE), BUT [TEEMAP] DOES NOT ALLOW**





# COMPARABLE

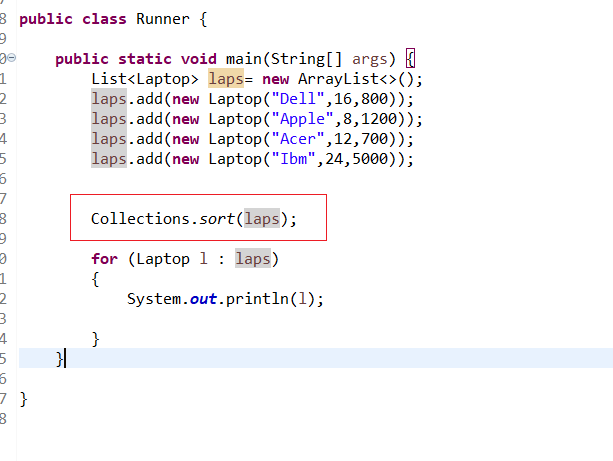
You need to use the interface [Comparable] that implements method [compareTo()]

Cons:

* You need to implement in interface. It’s not convenient, especially if you need to change order
* You can implement only one way of sorting

# 

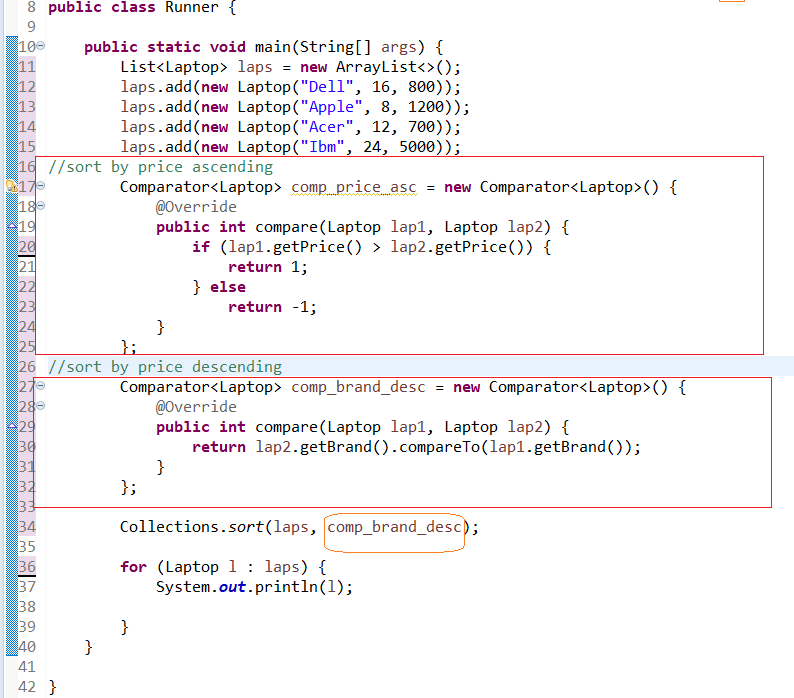


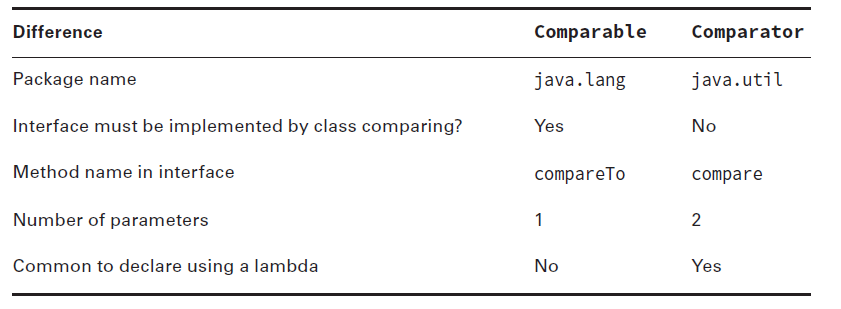


# COMPARATOR

Allows to define a few sorting and use what you need

Collections.sort(coll, sort\_comparator);



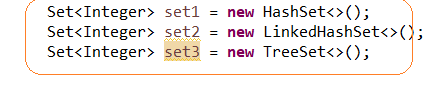


# COLELCTION BEST PRACTICS

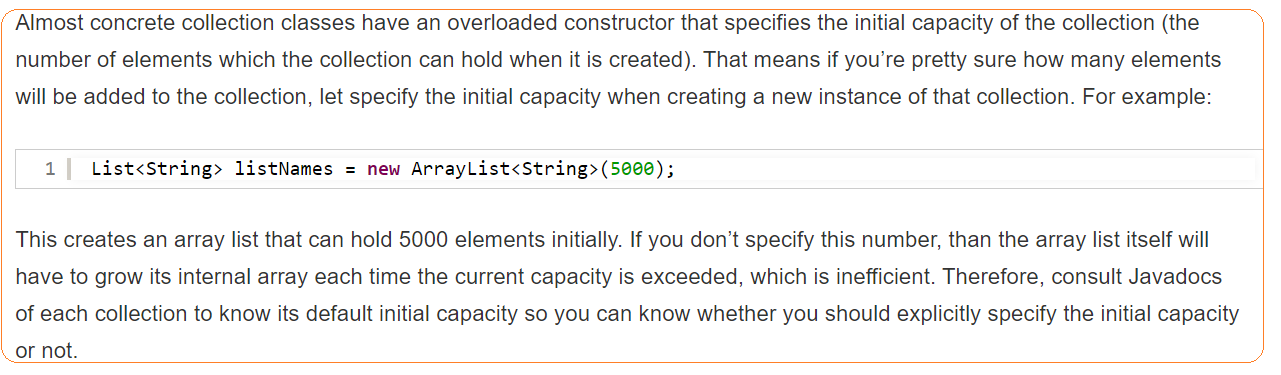
1.**Always using interface type when declaring a collection**.

By declaring a collection using an interface type,

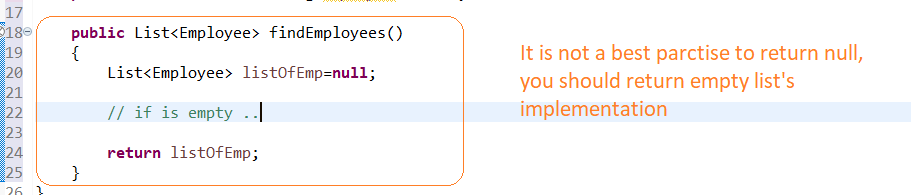
the code would be more flexible as you can change the concrete implementation easily when needed

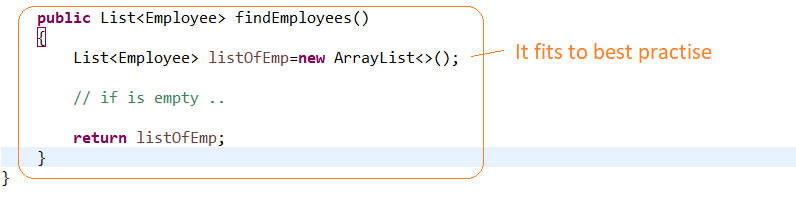


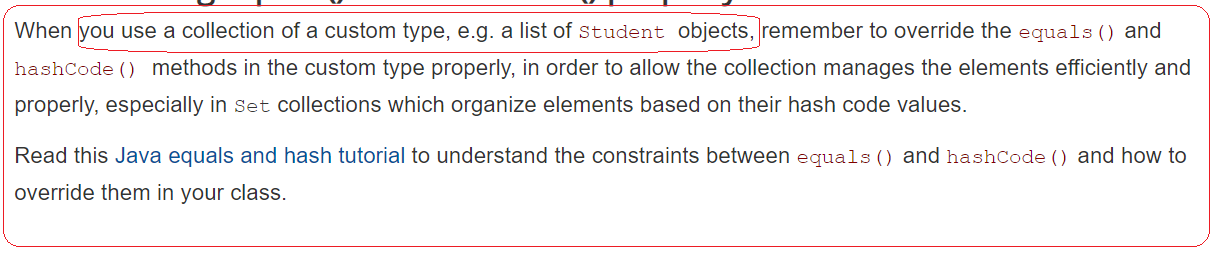
2. **Specify initial capacity of a collection if possible**.



3. **Do not return null in a method that returns a collection**.



4. **Overriding equals() and hashCode() MUST** .



# LAMBDA VS STREAM API

**Lambda expressions** are much like anonymous functions.

**Streams API** are sequences or elements that support sequential or parallel operations on their items (in a very different way from classic collections).