# COLLECTIONS

# 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** –

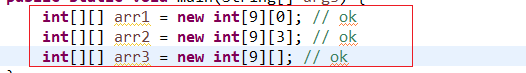
int[] arr1; - one dimensional array

int[][] arr2; - multidimensional array

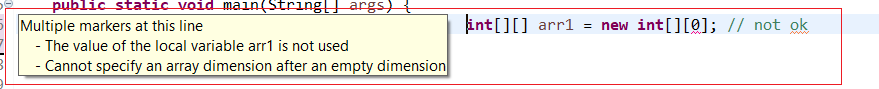
int[] arr2[]; - multidimensional array (another form)

* **allocation**

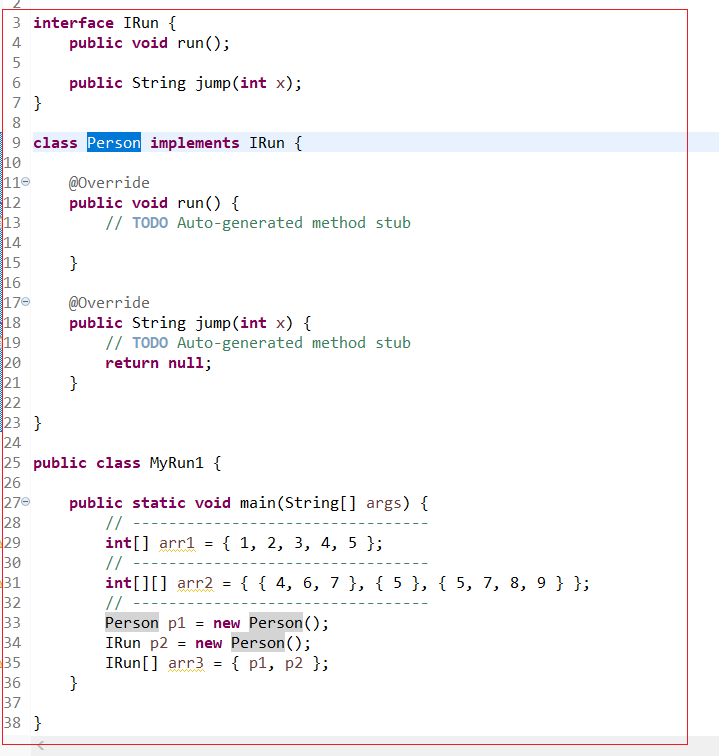
*example of right declaration:*



*example of not right declaration:*



* **initialization**



# ARRAYLIST

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

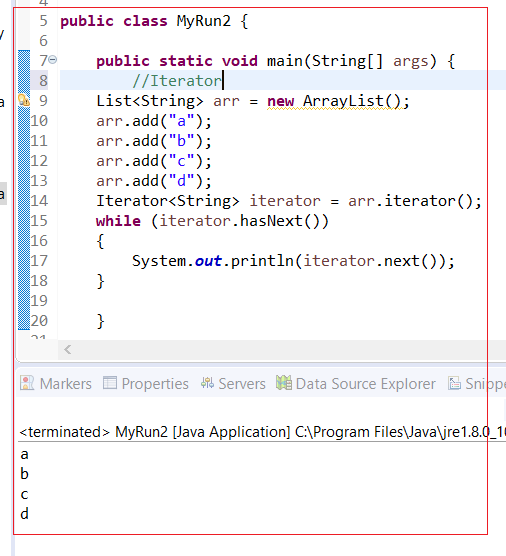
* Arraylist cannot contain primitives



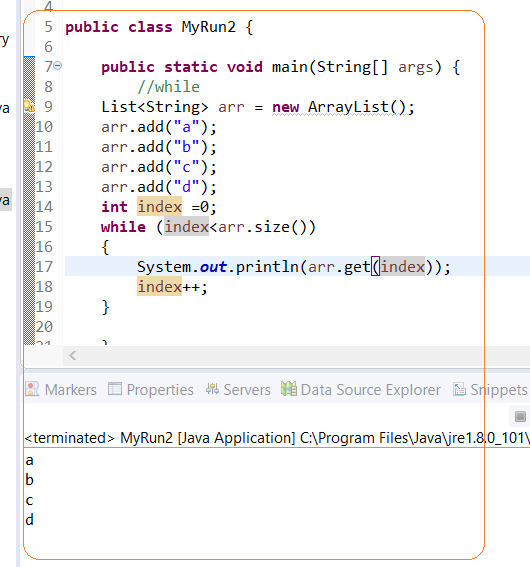
There are 5 ways to iterate through loop:

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

**ITERATOR**

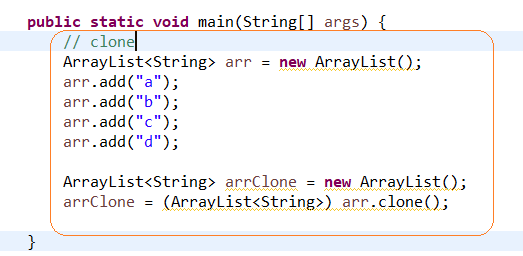


**WHILE**



**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

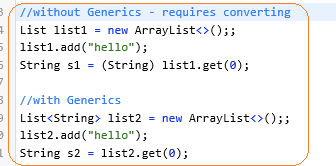


**Diamond operator** - The shorten form <> is called diamond (=the pair of angle brackets)

## MAIN

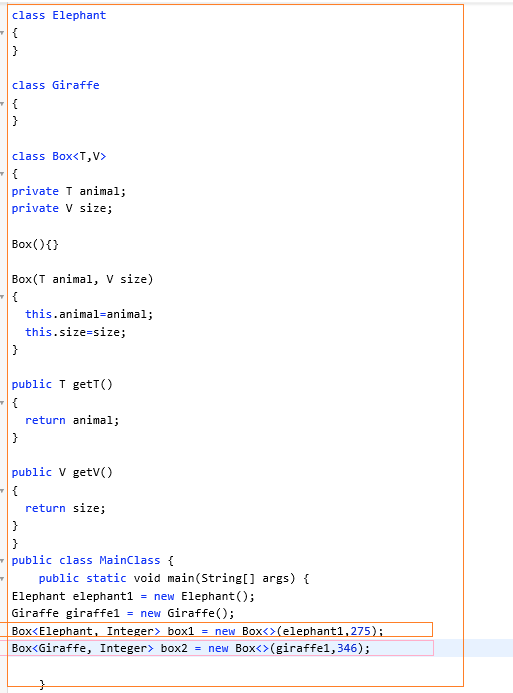
**Generics** – is similar like a template. It allows type to be a parameter to method, class or interface. Generic provides compile time checking and removing risk of [**ClassCastException**].

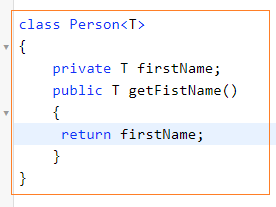
* *It allows to avoid run time erro*r **[ClassCastException**] *and if there mistake throws error at compile step*
* *We don’t need to make additional casting*

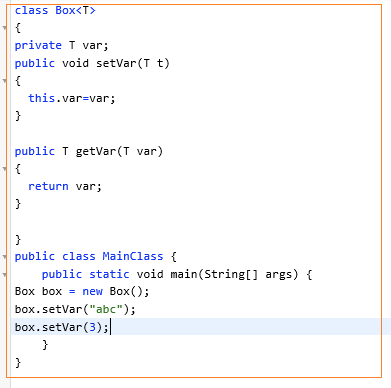


* *Help to reuse code*

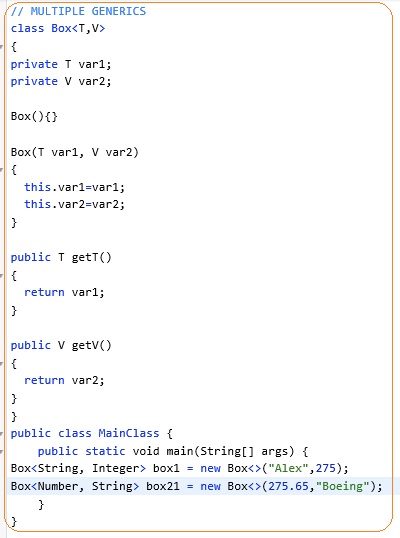
note: in example below you will not need to create Box for each animal. You can reuse the same Box – just parametrize





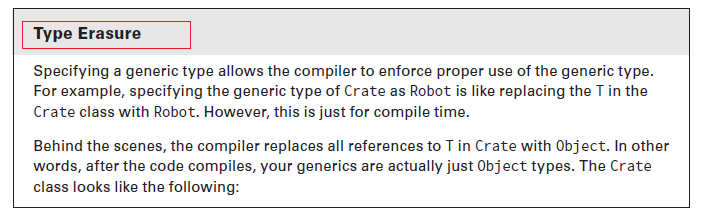


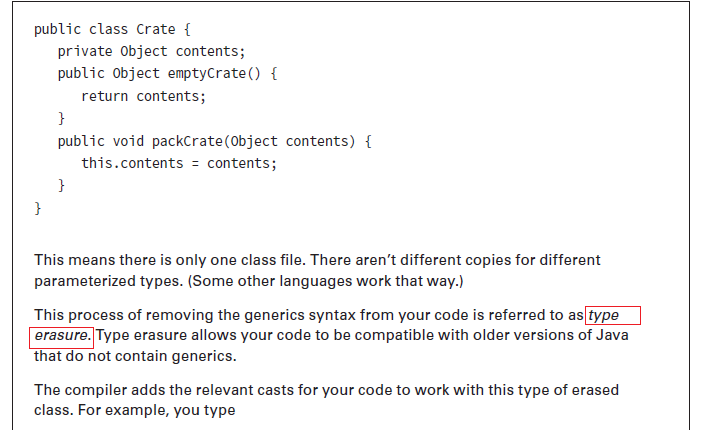
**Multiple generics types**

****

## 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





## 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>(); |

UPPER BOUNDED:



LOWER BOUNDED:



It will work for Integer->Number-\_Object

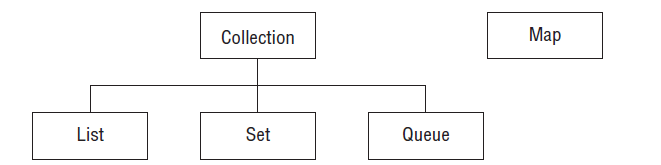
# COLLECTIONS

**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

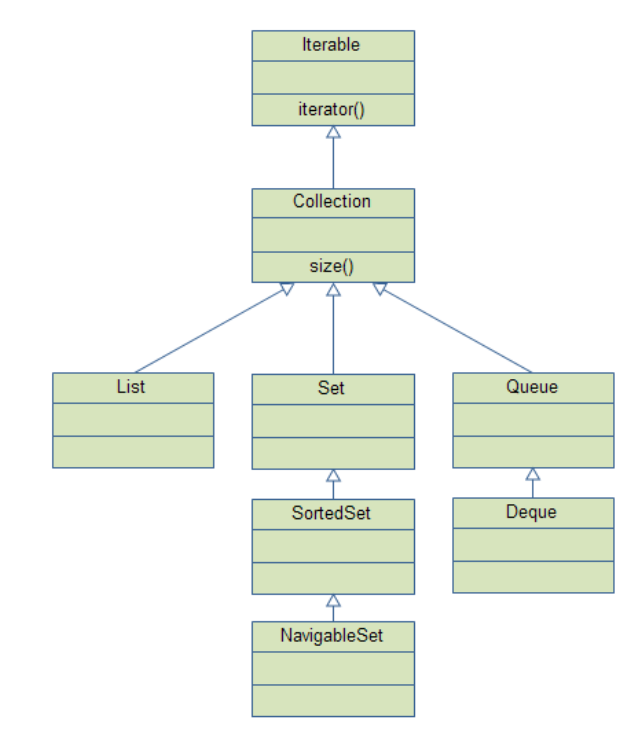
* Queue – is a collection that process elements in a specific order. Typical order is FIFO
* Map – set of key/values pairs. Key has to be unique



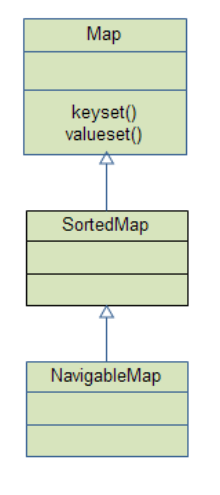
There are 2 group of interfaces

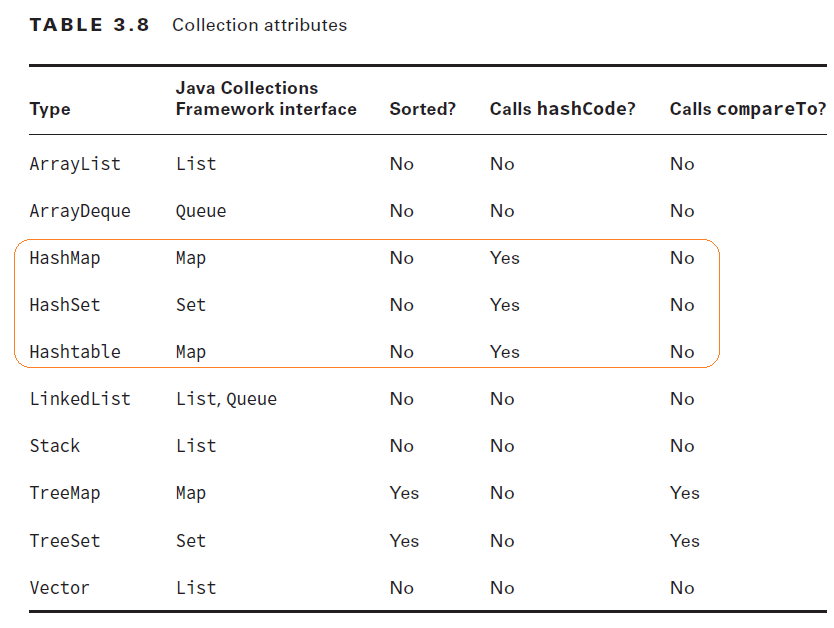
* Collection
* Map

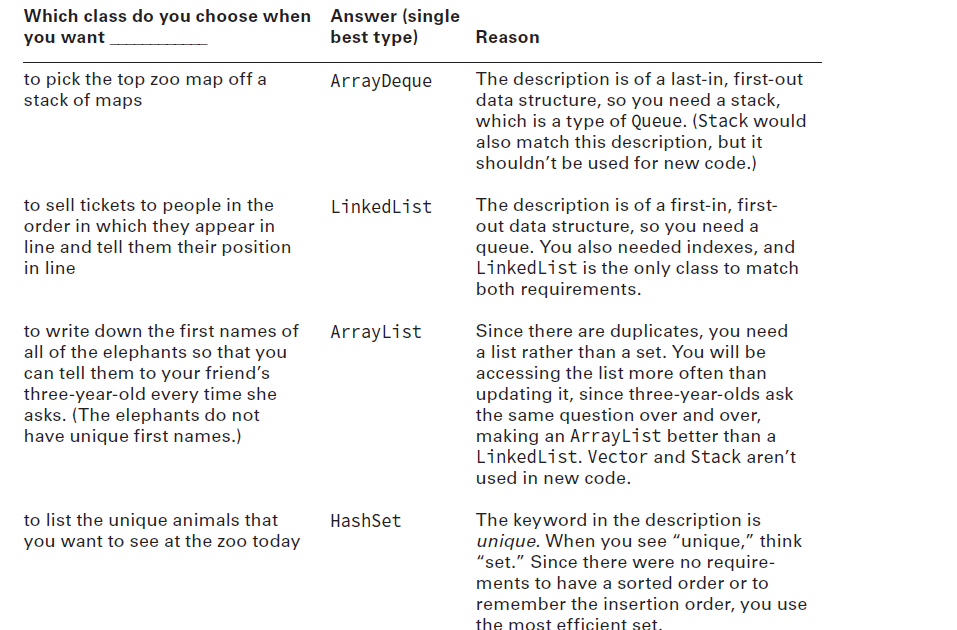
COLLECTION interface



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 |  |
| isEmpty() |  |
| size() |  |
| contains() |  |
| indexOf | Find index by element |
| clear() | Remove all elements |

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 that allows duplicates. 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

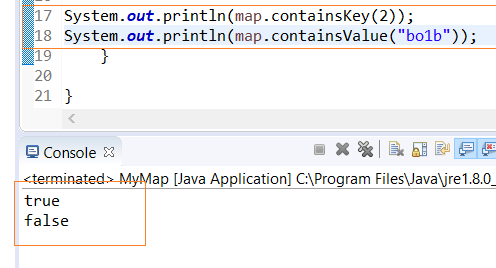
## SET INTERFACE

* Set –collection of elements that does not allow duplicates
* Can loop through [iterate] or [forEach]. Just [for loop] is not supported

Implementations

* HashSet
* LinkedHashSet
* TreeSet

Useful methods:



## HashSet

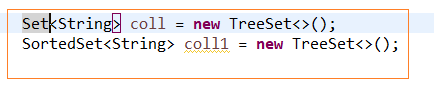
* **HashSet stores elements in hash table. It means it uses hashcode()**
* No guarantee on order
* Allows one null
* HashSet is the most common

## LinkedHashSet

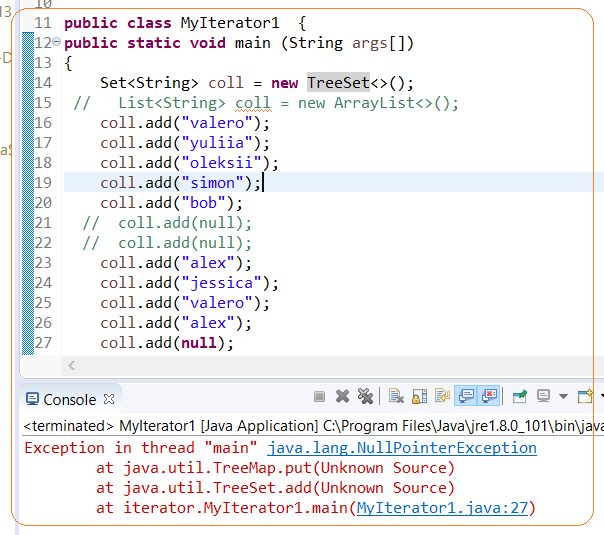
* Elements ordered in that order that inserted in collection
* Allows one null

## TreeSet

* TreeSet is implemennted from interface [SortedSet]



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



## QUEUE INTERFACE

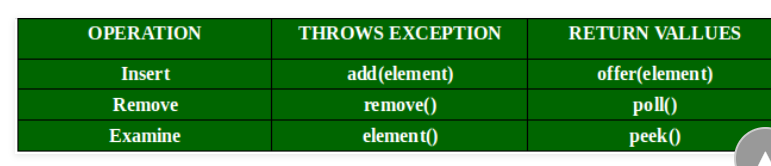
**Queue** –is infrastructure designed to have elements inserted at the end and removed from the beginning of the queue

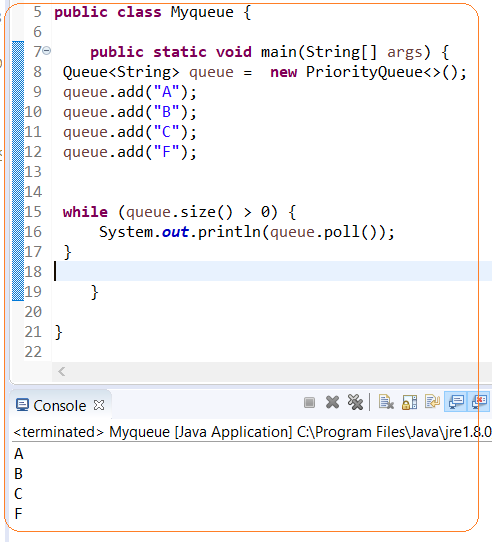
Implementation

* LinkedList
* PriorityQueue
* ArrayDeque –is more efficient than LinkedList

METHODS

|  |  |  |
| --- | --- | --- |
| METHOD | DESC | EXCEPTION/NULL |
| element() | Take the first element of Queue  without removing | Throws exception [NoSuchElementException] if the Queue is empty |
| peek() | Take the first element of Queue without removing | It returns null if the Queue is empty |
| remove() | Remove element from the head of Queue | Throws exception if the Queue is empty |
| poll() | Remove element from the head of Queue | It returns null if the Queue is empty |

****

****

## DEQUEUE INTERFACE

**Dequeue** –is Double ended queue. It implements FIFO and LIFO

Implementation

* LinkedList
* ArrayDequeue

**NOTE**

**WHEN FIFO: offer/poll/peek**

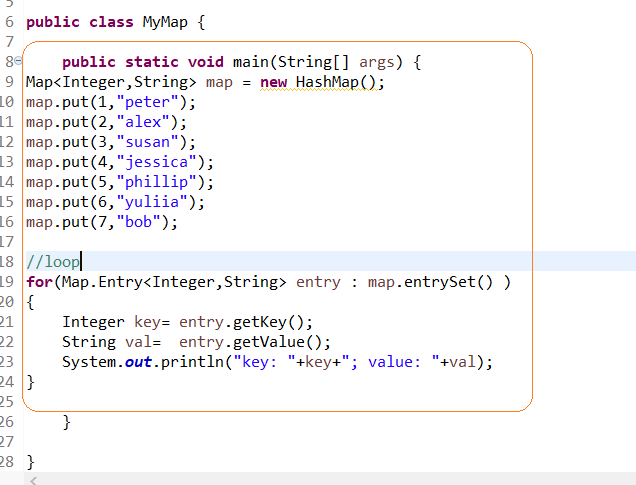
**WHEN LIFO: push/poll/peek**

## MAP INTERFACE

Map implementations

* HashMap-most common
* TreeMap-natural ordering
* LinkedHashMap

**Loop Map:**



## **JAVA ITERABLE**

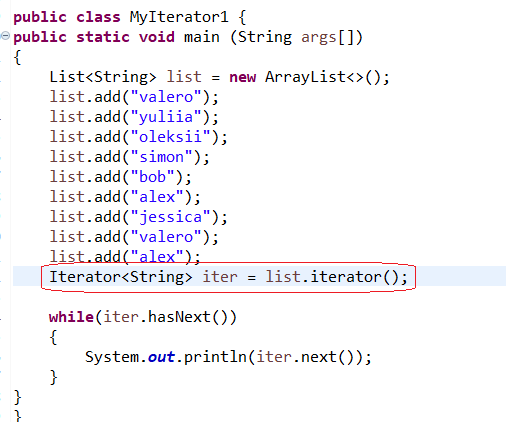
**Iterable interface** – is top of Collection hierarchy. Collection extends [Iterable] interface

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()].



# COMPARATOR

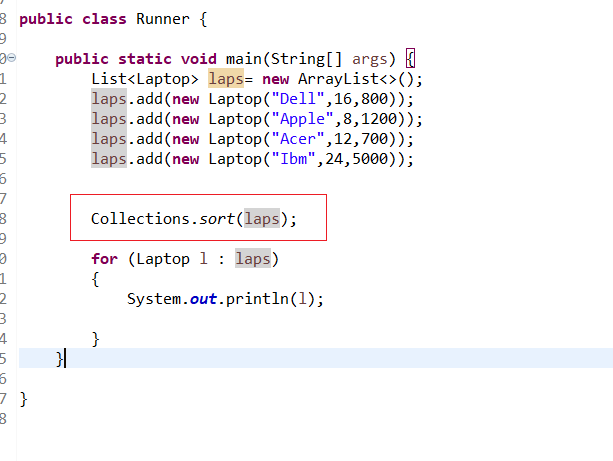
You need to use the interface [Comparator] that implements method [compareTo()]

Cons:

You need to implement in interface. It’s not convenient, especially if you need to change order

# 





# COMPARABLE

Allows to define a few sorting and use what you need

Collections.sort(coll, sort\_comparator);

