# Collections in Java

## Arrays

- a collection of values of similar data types `int[] nums = {10, 20, 33, 40}; String[] names = {"Vinod", "Shyam"}; Person[] people = {new Person("Vinod", "vinod@vinod.co"), new Person("Shyam", "shyam@xmpl.com")};`
- dynamically created, during the runtime.
- memory required for an array can be determined at the runtime, and the allocation of the required memory is done in the heap.
- once the array is crated, it cannot grow or shrink in size
- an array occupies continuous bytes in the heap. For eaxmple, if you create an array of a million double values, then the required memory space is 8 million bytes
- insertion and deletion of values at random indexes are difficult operations

## Collection framework

- introduced in the very first release of Java, to overcome the problems related to arrays, especially the redundant functionalities (like insert, remove, set, get, etc)
- in version 1.2, they did refactor and introduced a lot of new interfaces and implementations for the same. The old classes of version 1.0, went through changes too.
- in version 1.4, they introduced new features like `RandomAccess`
- in version 1.5 they introduced the concept of **generics**, which allows us to create collections of homogeneous types as against the heterfogeneous types.
    - Up to version 1.4, a Vector is a collection of objects (that may include numbers, strings, custom objects etc.)
    - Since 1.5, a Vector is a collection of either Integers, Strings or Custom objects
- in version 1.8, Java introduced a new set of APIs to work with huge amount of data in collections, called 'Streams' API.

```
// 1.0
public class Vector implements Cloneable, Serializable {

}

// 1.2
public class Vector extends AbstractList implements List, Cloneable,
Serializable {

}

// 1.4
public class Vector extends AbstractList implements List, Cloneable,
Serializable, RandomAccess {

}
```

```
// 1.4
public class Vector<T> extends AbstractList<T> implements List<T>,
Cloneable, Serializable, RandomAccess {

}
```

High level illustration of various classes and interfaces that makes the collection framework:

```
@startuml
interface Iterator<T> {
    + next(): T
    + hasNext(): boolean
    + remove(): void
}

interface Iterable<T> {
    + iterator(): Iterator<T>
}

interface Collection<T> {
    + add(t: T): boolean
    + addAll(coll: Collection<T>): boolean
    + clear(): void
    + remove(elem: T): boolean
    + removeAll(coll: Collection<T>): boolean
    + retainAll(coll: Collection<T>): boolean
    + isEmpty(): boolean
    + contains(elem: T): boolean
    + containsAll(coll: Collection<T>): boolean
    + size(): int
}

interface List{
    + add(index: int, t: T): void
    + addAll(index: int, coll: Collection<T>): void
    + remove(index: int): T
    + get(index: int): T
    + set(index: int, t: T): void
    + sublist(start: int, end: int): List<T>
}
note bottom of List: Provides index based operations
interface Set{}
interface Queue{}

Iterable <|-- Collection
Collection <|-- List
Collection <|-- Set
Collection <|-- Queue

class ArrayList<T>{}
class LinkedList<T>{}
class Vector<T>{}
```

```
class Stack<T>{}

List <|.. ArrayList
List <|.. LinkedList
List <|.. Vector
Vector <|-- Stack

note right of Iterable: an Iterable produces an Iterator

class HashSet<T>{}
class TreeSet<T>{}
class LinkedHashSet<T>{}
class Hashtable<T>{}
interface NavigatableSet<T>{}
interface SortedSet<T>{}

Set <|.. HashSet
Set <|.. LinkedHashSet
Set <|.. TreeSet
Set <|.. Hashtable
Set <|-- SortedSet
NavigatableSet <|.. TreeSet
SortedSet <|-- NavigatableSet
Hashtable <|-- Properties
HashSet <|-- LinkedHashSet


@enduml
```

1. ArrayList

    - since 1.2
    - uses an array of objects to hold the data
    - by default the initial capacity is 10
    - when the number of elements exceeds the capacity, new array is allocated with double the current capacity, and old values are copied to the new array, and allows more addition of elements
    - accessing an element from an array is always faster than other data structures like linked lists.
    - **this should be the default choice for a List**
    - insertion and deletion at random indexes are time consuming

2. LinkedList

    - since 1.2
    - internal storage is a doubly linked list
    - each node in the list are separate objects, and do not necessarily occupy continuous memory (as against an array)
    - insertion and deletion operations at radom index is faster than that of an array (especially insert and delete at index 0)
    - accessing an element at random index may be slower than that of an array

- use this implementation only when **insert and delete at 0 index** is more frequent than accessing elements at random indexes

3. Vector
    - since 1.0
    - a.k.a legacy collection
    - internal mechanism is same as **ArrayList**
    - some of methods are marked as *synchronized*, which has some overheads, but they are needed if the collection is shared among multiple threads
    - use this implementation only if the list is shared by multiple threads that may mutate the data
4. Stack
    - since 1.0
    - provides the standard *STACK* operations such as *push()* or *pop()* etc.

Set is an interface that does not provide any additional methods (until Java 1.8). A set object does not allow duplicate entries. Following are the choices of implementation we have:

1. HashSet
    - does not guarantee the order of retrieval to be same as order of insertion
    - internally uses another collection type called `HashMap`, which internally uses an array of object references to store the data
    - this must be the preferred Set implementation, since arrays are faster than other data structures, during retrieval.
    - in order to prevent the insertion of duplicate values, this class depends on the element's `hashCode()` and `equals()` methods.
2. LinkedHashSet
    - uses a linked list to store the data
    - uses `hashCode` and `equals` for duplicate check
    - assures the order of retrieval to be same as order of insertion
3. TreeSet
    - assures that the order retrieval is ascending order of element's natural ordering
    - uses a red-black-tree for store data
    - **DOES NOT REQUIRE `hashCode` and `equals` methods** in the element's type
4. Hashtable
    - legacy (since 1.0)
    - has methods marked as *synchronized* for thread safety
    - requires `hashCode` and `equals` methods in the element

It's a good practice to use the itnerface names for function's parameters and return types.

For example,

```
    public List<Employee> findByCity(String city) {
        // logic here
        return ...
    }
```

The above method may return (is allowed to return) an object of:

1. ArrayList
2. Vector
3. LinkedList
4. Stack
5. or any other class that implements `java.util.List` interface

However,

```
public ArrayList<Employee> findByCity(String city) {
    // logic here
    return ...
}
```

the above function can only return an object of

1. ArrayList
2. or any subclass of ArrayList