**COLLECTIONS**

Collections in java is a framework that provides an architecture to store and manipulate the group of objects.

All the operations that you perform on a data such as searching, sorting, insertion, manipulation, deletion etc. can be performed by Java Collections.

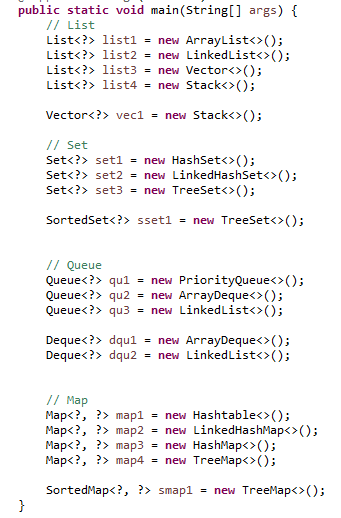
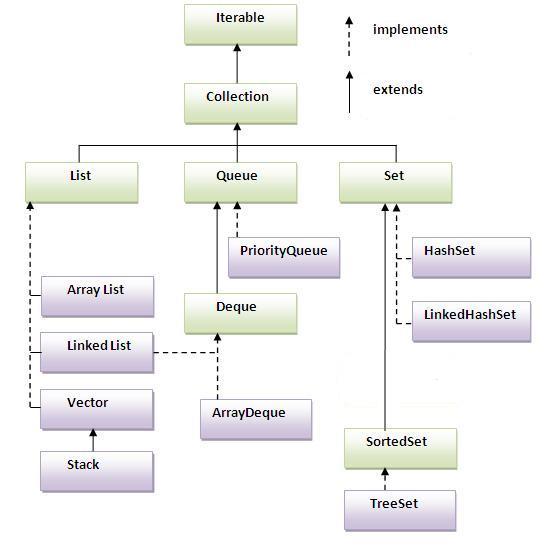
**Collection -** Collection represents a single unit of objects i.e. a group.

**Framework -** provides readymade architecture, represents set of classes and interface, and is optional.

**Collection Framework -** Collection framework represents a unified architecture for storing and manipulating group of objects. It has Interfaces and its implementations i.e. classes, and Algorithm

Hierarchy of Collection Framework

Let us see the hierarchy of collection framework. The java.util package contains all the classes and interfaces for Collection framework.



**Iterable**

The Iterable interface (java.lang.Iterable) is one of the root interfaces of the Java collection classes. The Collection interface extends Iterable, so all subtypes of Collection also implement the Iterable interface.

A class that implements the Iterable can be used with the new for-loop.

**Collection**

The Collection interface (java.util.Collection) is one of the root interfaces of the Java collection classes. Though you do not instantiate a Collection directly, but rather a subtype of Collection, you may often treat these subtypes uniformly as a Collection.

**List**

-It represents an ordered list of objects, meaning you can access the elements of a List in a specific order, and by an index too. You can also add the same element more than once to a List.

-Elements can be inserted or accessed by their position in the list, using a zero-based index.

-A list may contain duplicate elements.

**Array List**

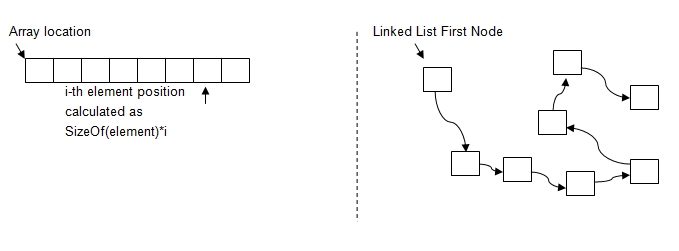
ArrayList is the resizable array implementation of list interface

ArrayList is preferred when there are more get(int) or search operations need to be performed as every search operation runtime is O(1).

**Linked List**

LinkedList is the Doubly-linked list implementation of the list interface

If application requires more insert(int) , delete(int) operations than the get(int) operations then LinkedList is preferred as they do not need to maintain back and forth like arraylist  to preserve continues indices.



**Vector**

Vector is similar with ArrayList, but it is synchronized. It also contains many legacy methods that are not part of the collections framework.

**Stack**

A Stack is a data structure where you add elements to the "top" of the stack, and also remove elements from the top again. This is also referred to as the "Last In First Out (LIFO)" principle. In contrast, a Queue uses a "First In First Out (FIFO)" principle.

**Queue**

-It represents an ordered list of objects just like a List, but its intended use is slightly different. A queue is designed to have elements inserted at the end of the queue, and elements removed from the beginning of the queue.

-Queues typically, but do not necessarily, order elements in a FIFO (first-in-first-out) manner. Among the exceptions are priority queues, which order elements according to a supplied comparator, or the elements' natural ordering, and LIFO queues (or stacks) which order the elements LIFO (last-in-first-out).

**Priority Queue**

The PriorityQueue class provides the facility of using queue. But it does not orders the elements in FIFO manner.

A priory queue is a queue in which each element has an associated priority. The element with the highest priority is removed next from the queue.

**Deque**

-A linear collection that supports element insertion and removal at both ends. The name deque is short for "double ended queue" and is usually pronounced "deck". This interface defines methods to access the elements at both ends of the deque.

**Array Deque**

Resizable-array implementation of the [Deque](https://docs.oracle.com/javase/7/docs/api/java/util/Deque.html" \o "interface in java.util) interface. Array deques have no capacity restrictions; they grow as necessary to support usage. They are not thread-safe; in the absence of external synchronization, they do not support concurrent access by multiple threads. Null elements are prohibited. This class is likely to be faster than [Stack](https://docs.oracle.com/javase/7/docs/api/java/util/Stack.html) when used as a stack, and faster than [LinkedList](https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html" \o "class in java.util)when used as a queue.

**Set**

A Set is a Collection that cannot contain duplicate elements.

**Hash Set**

HashSet is Implemented using a hash table. Elements are not ordered. The add, remove, and contains methods have constant time complexity O(1).

**Linked Hash Set**

LinkedHashSet is between HashSet and TreeSet. It is implemented as a hash table with a linked list running through it, so it provides the order of insertion. The time complexity of basic methods is O(1).

**Sorted Set**

It behaves like a normal set with the exception that the elements are sorted internally. This means that when you iterate the elements of a SortedSet the elements are returned in the sorted order.

**Tree Set**

TreeSet is implemented using a tree structure(red-black tree in algorithm book). The elements in a set are sorted, but the add, remove, and contains methods has time complexity of O(log (n)). It offers several methods to deal with the ordered set like first(), last(), headSet(), tailSet(), etc.

**Map**

-The java.util.Map interface represents a mapping between a key and a value. The Map interface is not a subtype of the Collection interface. Therefore it behaves a bit different from the rest of the collection types.

-Given a key and a value, you can store the value in a Map object. After the value is stored, you can retrieve it by using its key.

**Hash Table**

Hashtable is synchronized, in contrast to HashMap. It has an overhead for synchronization.

**Linked Hash Map**

LinkedHashMap preserves the insertion order

**Hash Map**

HashMap is implemented as a hash table, and there is no ordering on keys or values.

**Tree Map**

TreeMap is implemented based on red-black tree structure, and it is ordered by the key.

**Sorted Map**

The SortedMap interface extends Map, with the addition that the elements stored in the map are sorted internally.

**Benefits of Java Collections Framework**

Java Collections framework have following benefits:

* **Reduced Development Effort –** It comes with almost all common types of collections and useful methods to iterate and manipulate the data. So we can concentrate more on business logic rather than designing our collection APIs.
* **Increased Quality –** Using core collection classes that are well tested increases our program quality rather than using any home developed data structure.
* **Reusability and Interoperability**
* **Reduce effort –** to learn any new API if we use core collection API classes.

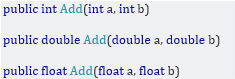
**GENERICS**

- The Java Generics programming is introduced in J2SE 5 to deal with type-safe objects.

Before generics, we can store any type of objects in collection i.e. non-generic. Now generics, forces the java programmer to store specific type of objects.

Generic Method

- Generics allow you to customize a "**generic**" method or class to whatever type you're working with, this method can accept any type of argument. For example, suppose you have a method that adds two numbers together. In order to work with the types themselves, you might have to create multiple versions of this method. For instance:



Generics allow you to create a single method that is customized for the type that invokes it.



T is substituted for whatever type you use.

Advantage of Java Generics

**1) Type-safety:** We can hold only a single type of objects in generics. It doesn’t allow to store other objects.

**2) Type casting is not required:** There is no need to typecast the object.

**3) Compile-Time Checking:** It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

Wildcard

The ? (Question mark) symbol represents wildcard element. It means any type. If we write <? extends Number>, it means any child class of Number e.g. Integer, Float, double etc. Now we can call the method of Number class through any child class object.

Type Parameters

It's more convention than anything else.

* T is meant to be a Type
* E is meant to be an Element (List<E>: a list of Elements)
* K is Key (in a Map<K,V>)
* V is Value (as a return value or mapped value)

But these are just conventions. It doesn't even have to be a single, upper-case letter; you can use any name that you like, just like you can give classes, variables etc. any name you like

List, List<>, List<?>, List<Object>

- I'd put it this way: While List, List<>, and List<Object> can contain any type of objects, List<?> contains elements of an unknown type, but once that type is captured, it can only contain elements of that type. Which is why it is the only type safe variant of those three, and therefore generally preferable.