CSE 3002- PROGRAMMING IN JAVA

UNIT III
GENERICS AND COLLECTIONS

DR. R. MANIKANDAN
ASSISTANT PROFESSOR (SENIOR GRADE)
VIT BHOPAL UNIVERSITY

Generics: Introduction

- **Generics** are a facility of generic programming that were added to the Java programming language in 2004 within version J2SE 5.0.
- They were designed to extend Java's type system to allow “a type or method to operate on objects of **various types** while providing **compile-time type safety**”.
- **Generics** in Java is similar to **templates in C++**. The idea is to allow type (Integer, String, … etc and user defined types) to be a parameter to methods, classes and interfaces.

- **There are mainly 3 advantages of generics.**

  1) Provide **flexible type safety** to your code
  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.

Generic Class: Introduction

- We can define our own classes with generics type.
- A generic type is a class or interface that is parameterized over types. We use angle brackets (<>) to specify the type parameter.
- To create objects of generic class, we use following syntax.
  - **BaseType <Type> obj = new BaseType <Type>()**
- **Note:** In Parameter type we can not use primitives like 'int','char' or 'double'.

The most commonly used type parameter names are:

- E – Element (used extensively by the Java Collections Framework, for example ArrayList, Set etc.)
- K – Key (Used in Map)
- N – Number
- T – Type
- V – Value (Used in Map)
- S,U,V etc. – 2nd, 3rd, 4th types

- **Example:** Generic1.java, Generic2.java

Generic Method: Introduction

- Generic functions that can be called with different types of arguments based on the type of arguments passed to generic method, the compiler handles each method.
- **Example:** Generic3.java, Generic4.java

Bounded Type Parameters

- There may be times when you want to restrict the types that can be used as type arguments in a parameterized type.
- 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.

Syntax

- <T extends **superClassName**>
  - List the type parameter's name
  - Along by the extends keyword
  - And by its upper bound

Example: BoundClass.java, BoundedClass1.java

Collection Framework: Introduction

- A **Java collection framework** provides an **architecture** to store and manipulate a **group of objects**.
- Java Collections can achieve all the operations that you perform on a data such as **searching, sorting, insertion, manipulation, and deletion.**

Java arrays have limitations

- They cannot dynamically shrink and grow.
- They have limited type safety.
- Implementing efficient, complex data structures from scratch would be difficult.

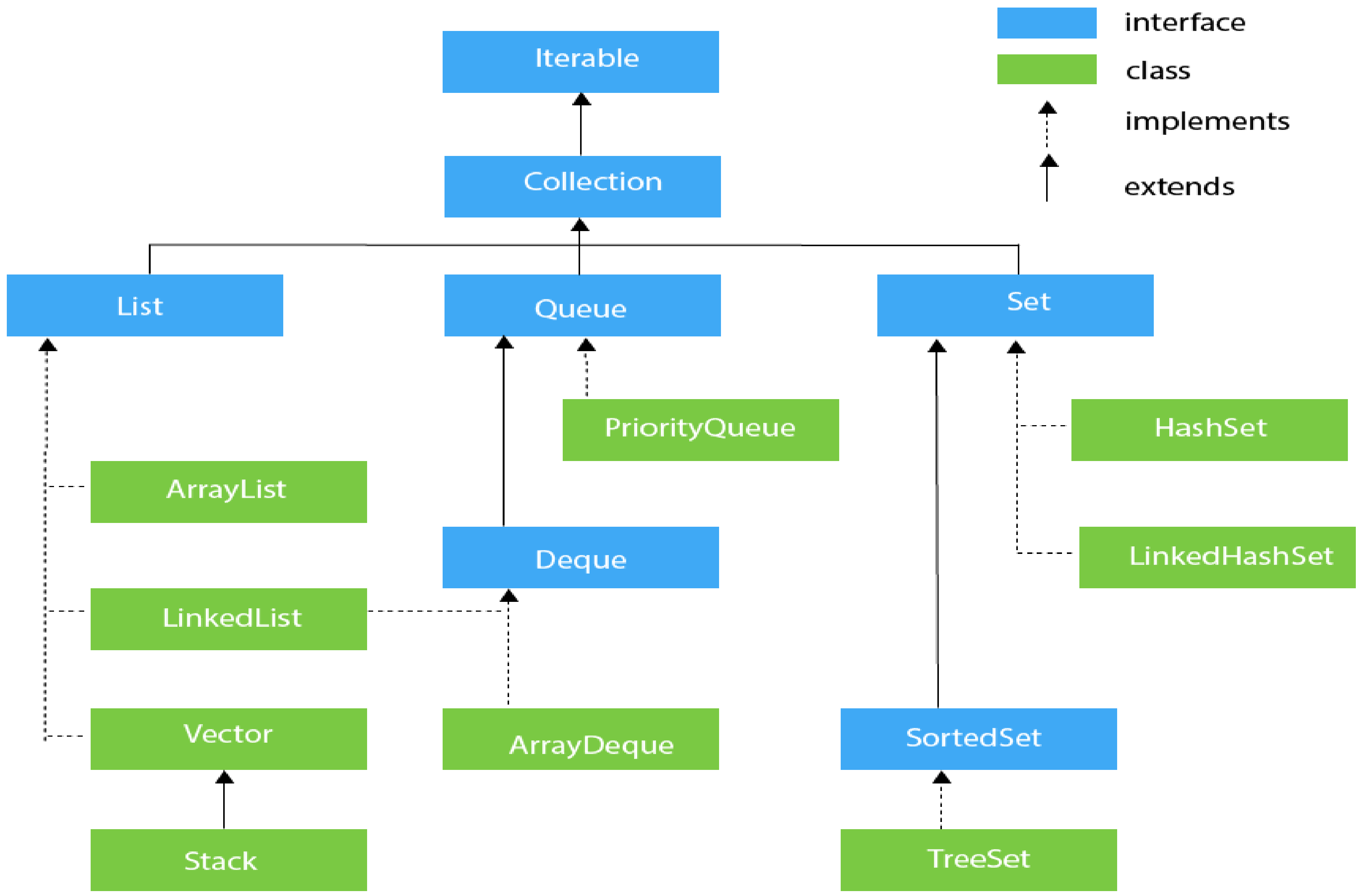
- **Benefits of Java Collections Framework**
  - **Reduced Development Effort** – It comes with almost all common types of collections and useful methods to iterate and manipulate the data. So we can concentrate more on business logic rather than designing our collection APIs.

Collection Framework: Introduction

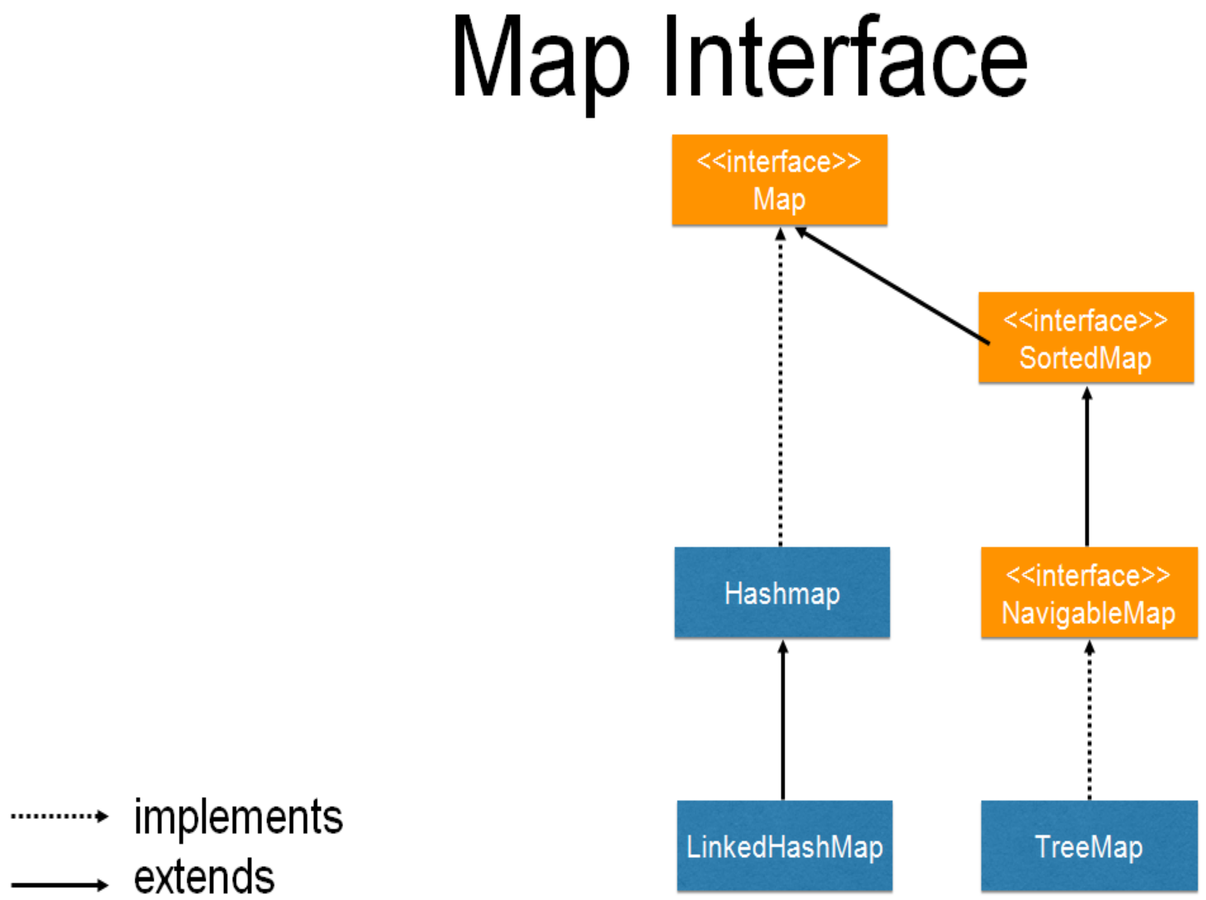
- **Reusability and Interoperability**
- The framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability.
- The framework had to extend and/or adapt a collection easily.
- **Increased Quality**
- Using core collection classes that are well tested increases our program quality rather than using any home developed data structure

- A Java collection framework includes the following:
  - **Interfaces**: Interface in Java refers to the abstract data types. They allow Java collections to be manipulated independently from the details of their representation. Also, they form a hierarchy in **object-oriented programming languages.**
  - **Classes:** Classes in Java are the implementation of the collection interface. It basically refers to the data structures that are used again and again.
  - **Algorithm:** Algorithm refers to the methods which are used to perform operations such as searching and sorting, on objects that implement collection interfaces. Algorithms are **polymorphic in nature** as the same method can be used to take many forms or you can say perform different implementations of the Java collection interface.

Collection Framework Hierarchy



Collection Framework Hierarchy



All the collection classes are present in java.util and java.util.concurrent package.

Collection Interface

- **Iterable Interface**
  - The Iterable interface is the root interface for all the collection classes. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.
  - It contains only one abstract method. i.e.,
  - Iterator<T> iterator()
  - It returns the iterator over the elements of type T.
- **Iterator interface**
  - Iterator is an interface that iterates the elements. It is used to traverse the list and modify the elements. Iterator interface has three methods which are mentioned below:
  - **public boolean hasNext()** – This method returns true if the iterator has more elements.
  - **public object next()** – It returns the element and moves the cursor pointer to the next element.
  - **public void remove()** – This method removes the last elements returned by the iterator.
  - **Note:** If any class implements the Iterable interface, it gains the ability to iterate over an object of that class using an iterator.

Collection Interface: List

- **List interface** is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can **have duplicate values**.
- List interface is implemented by the classes **ArrayList, LinkedList, Vector, and Stack.**
- **ArrayList**
  - The ArrayList class implements the List interface. It uses a dynamic array to store the duplicate element of different data types.
  - The ArrayList class maintains the **insertion order** and is **non-synchronized**. The elements stored in the ArrayList class can be **randomly accessed**.
- **Example:** ArrayListExample, 1, 2.java, CustomArrayList .java

Collection Interface: List

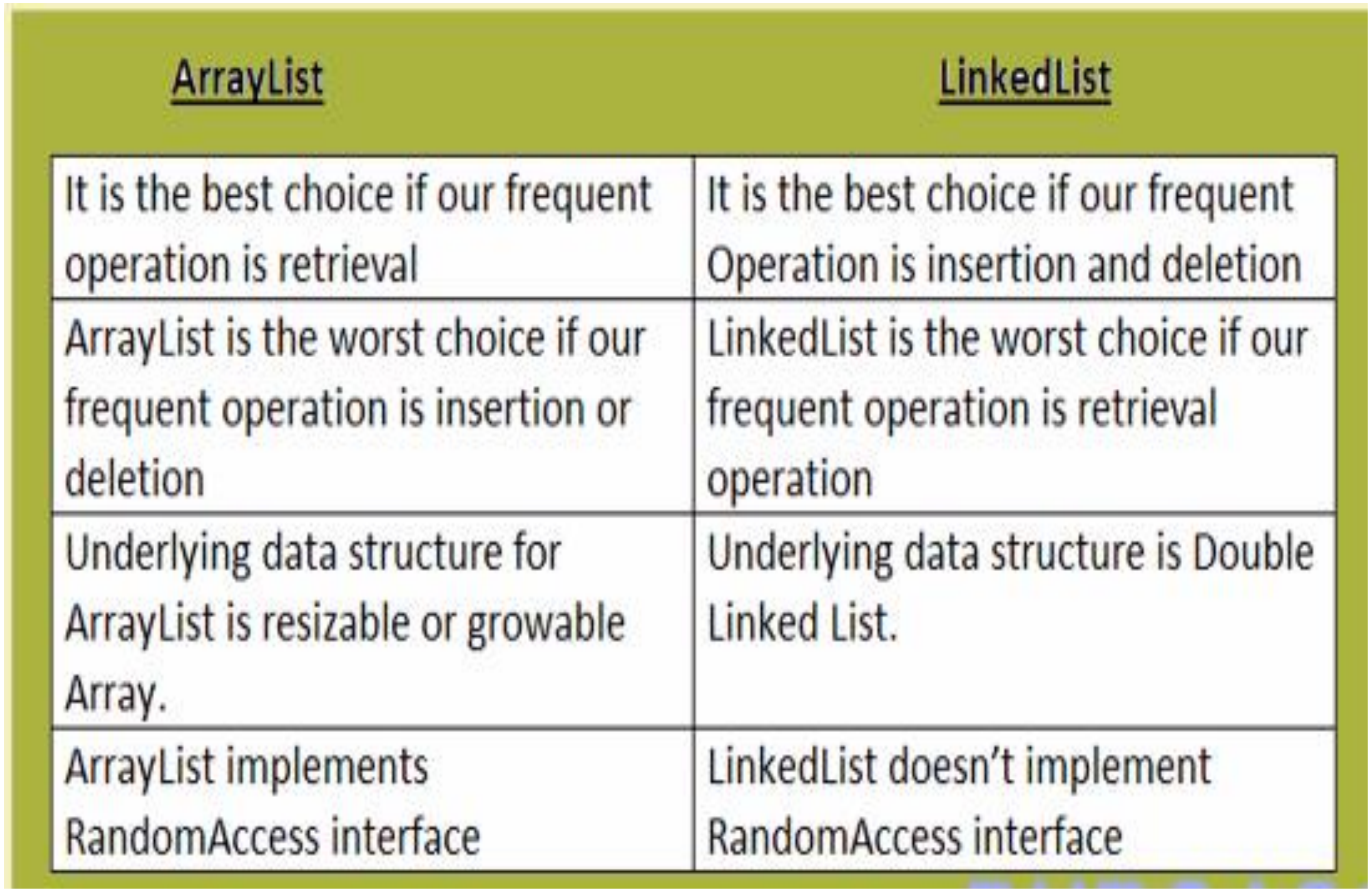
- **Why ArrayList is better than Array?**
  - The limitation with array is that it has a fixed length so if it is full you cannot add any more elements to it, likewise if there are number of elements gets removed from it the memory consumption would be the same as it doesn't shrink.
  - On the other ArrayList can dynamically grow and shrink after addition and removal of elements. Apart from these benefits ArrayList class enables us to use predefined methods of it which makes our task easy.
- **Why ArrayList is not synchronized?**
  - Array list is not synchronized means that object is mutable that means once creating the Array list object that object is calling two threads at a time but one thread is changing the value of object that can be effected by another object so **ArrayList is not thread safe** so Array list is not Synchronized by default.
- **More Methods & Examples**
  - https://beginnersbook.com/java-collections-tutorials/

Collection Interface: List

- **LinkedList**
  - LinkedList is a **doubly-linkedlist** implementation of the **List and Deque** interfaces.
  - It uses a doubly linked list internally to store the elements. It can store the **duplicate elements**. It maintains the insertion order and is **not synchronized**. In LinkedList, the **manipulation is fast** because no shifting is required.
  - LinkedList allows for constant-time insertions or removals using iterators, but only sequential access of elements.
  - In other words, LinkedList can be searched forward and backward but the time it takes to traverse the list is directly proportional to the size of the list. **Example:** LinkedList1.java

Collection Interface: List

- **ArrayList Vs LinkedList**



ArrayList	LinkedList
It is the best choice if our frequent operation is retrieval	It is the best choice if our frequent Operation is insertion and deletion
ArrayList is the worst choice if our frequent operation is insertion or deletion	LinkedList is the worst choice if our frequent operation is retrieval operation
Underlying data structure for ArrayList is resizable or growable Array.	Underlying data structure is Double Linked List.
ArrayList implements RandomAccess interface	LinkedList doesn't implement RandomAccess interface

Example: LinkedList2.java

More Methods & Examples:
https://beginnersbook.com/2013/12/linkedlist-in-java-with-example/

Collection Interface: Vector

- **Vector**
  - The vector class implements a **growable array of objects**. Like an array, it contains the component that can be accessed using an integer index.
  - Vector is very useful if we don't know the size of an array in advance or we need one that can change the size over the lifetime of a program.
  - Vector implements a **dynamic array** that means it can grow or shrink as required.
  - Vectors basically falls in **legacy classes** but now it is fully compatible with collections.
    - Early version of java did not include the Collections framework. It only defined several classes and interfaces that provide methods for storing objects.
    - When Collections framework were added in J2SE 1.2, the original classes were reengineered to support the collection interface. These classes are also known as **Legacy classes**.

Collection Interface: Vector

  - All legacy classes and interface were redesign by JDK 5 to support Generics. In general, the legacy classes are supported because there is still some code that uses them.

- It is similar to the ArrayList, but with two differences-
  - Vector is synchronized.
  - The vector contains many legacy methods that are not the part of a collections framework
- **Example:** VectorExample1.java
- **More Methods & Examples:**

https://beginnersbook.com/2013/12/vector-in-java/

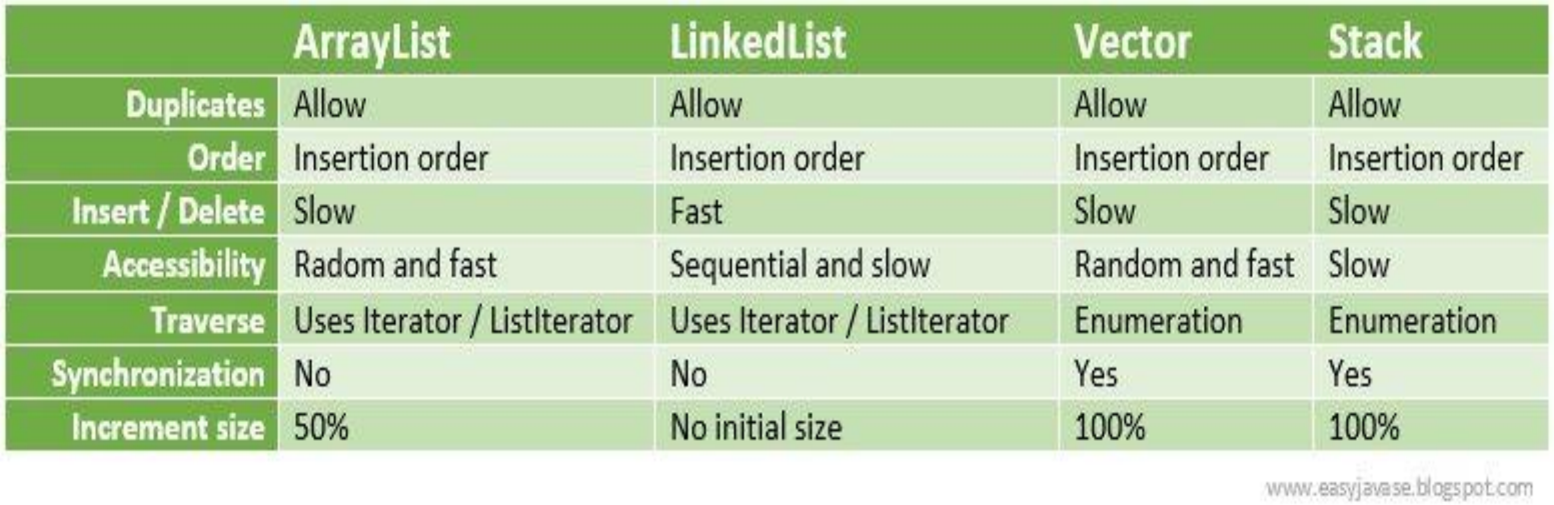
https://www.geeksforgeeks.org/java-util-vector-class-java/

Collection Interface: Stack

- **Stack**
  - Java Collection framework provides a Stack class which models and implements Stack data structure. The class is based on the basic principle of **Last-In-First-Out (LIFO)**.
  - In addition to the basic push and pop operations, the class provides three more functions of empty, search and peek.
    - **Object push(Object element)** : Pushes an element on the top of the stack.
    - **Object pop()** : Removes and returns the top element of the stack. An 'EmptyStackException' exception is thrown if we call pop() when the invoking stack is empty.
    - **Object peek()** : Returns the element on the top of the stack, but does not remove it.
    - **boolean empty()** : It returns true if nothing is on the top of the stack. Else, returns false.
    - **int search(Object element)** : It determines whether an object exists in the stack. If the element is found, it returns the position of the element from the top of the stack. Else, it returns -1.

Collection Interface: Stack

- **Stack**
  - The class can also be said to **extend Vector** and treats the class as a stack with the five mentioned functions. The class can also be referred to as the **subclass of Vector.**
  - **Example:** StackExample.java



	ArrayList	LinkedList	Vector	Stack
Duplicates	Allow	Allow	Allow	Allow
Order	Insertion order	Insertion order	Insertion order	Insertion order
Insert / Delete	Slow	Fast	Slow	Slow
Accessibility	Radom and fast	Sequential and slow	Random and fast	Slow
Traverse	Uses Iterator / ListIterator	Uses Iterator / ListIterator	Enumeration	Enumeration
Synchronization	No	No	Yes	Yes
Increment size	50%	No initial size	100%	100%

www.easyjavase.blogspot.com

Collection Interface: General

What are the main differences between Collection and Collections in Java?

- Major difference between Collection and Collections is Collection is an interface and Collections is a class. Both are belongs to java.util package. Collection is base interface for list set and queue. Collections is a class and it is called utility class.
- Collections **utility class** contains some predefined methods so that we can use while working with Collection type of classes(treeset, arraylist, linkedlist etc.) Collection is base interface for List , Set and Queue.

What is Utility Class?

- Utility Class, also known as Helper class, is a class, which contains just static methods, it is stateless and cannot be instantiated. It contains a bunch of related methods, so they can be **reused across the application**. As an example consider Apache StringUtils, CollectionUtils or java.lang.Math.
- **Example:** Utility1.java, Utility2.java
- **More Details**: https://www.programcreek.com/2015/12/top-10-java-utility-classes/

Comparable and Comparator

- **Comparable and Comparator** both are interfaces and can be used **to sort collection elements**.

Comparable Interface

- A comparable object is capable of comparing itself with another object. The class itself must implements the **java.lang.Comparable** interface to compare its instances.
- **Example:**
  - Consider a Movie class that has members like, rating, name, year. Suppose we wish to sort a list of Movies based on year of release. We can implement the Comparable interface with the Movie class, and we override the method **compareTo()** of Comparable interface.
  - **Example:** ComparableExample.java
  - **Note:**
  - Now, suppose we want sort movies by their rating and names also. When we make a collection element comparable(by having it implement Comparable), we get only one chance to implement the compareTo() method. The solution is using **Comparator**.

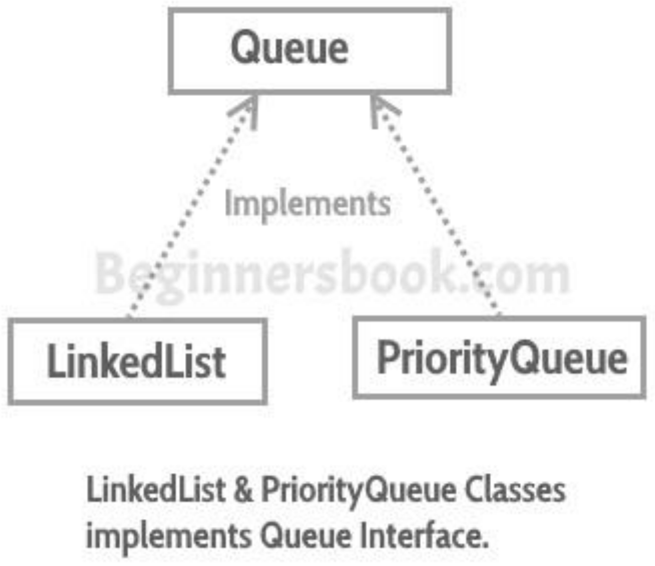
Comparable and Comparator

Comparator Interface

- Unlike Comparable, Comparator is external to the element type we are comparing. It's a separate class.
- We create multiple separate classes (that implement Comparator) to **compare by different members**.
- A Comparator is present in the **java.util** package.
- Collections class has a second sort() method and it takes Comparator. The sort() method invokes the compare() to sort objects.
  - **Example:** Comparator Example.java

Collection Interface: Queue

- **Queue interface**
  - It orders the element in **FIFO**(First In First Out) manner. In FIFO, first element is removed first and last element is removed at last.


Queue
Implements
Beginnersbook.com
LinkedList
PriorityQueue
LinkedList & PriorityQueue Classes implements Queue Interface.

  - **Example**: QueueExample.java

Collection Interface: Queue

- **PriorityQueue class**
  - **What if we want to serve the request based on the priority rather than FIFO?**
  - A PriorityQueue is used when the objects are supposed to be processed based on the priority. It is known that a queue follows First-In-First-Out algorithm, but sometimes the elements of the queue are needed to be processed according to the **priority,** that's when the PriorityQueue comes into play.
  - The PriorityQueue is based on the priority heap. The elements of the priority queue are **ordered according to the natural ordering, or by a Comparator provided at queue construction time**, depending on which constructor is used.
  - **Example:** QueueExample1, 2.java

Collection Interface: Queue

- **Queue interface**
  - Java Deque Interface is a linear collection that supports element insertion and removal at both ends. Deque is an acronym for **"double ended queue".**
  - Deque is an interface and has two implementations: LinkedList and ArrayDeque.
  - **Example**: ArrayDequeExample.java

Collection Interface: Set

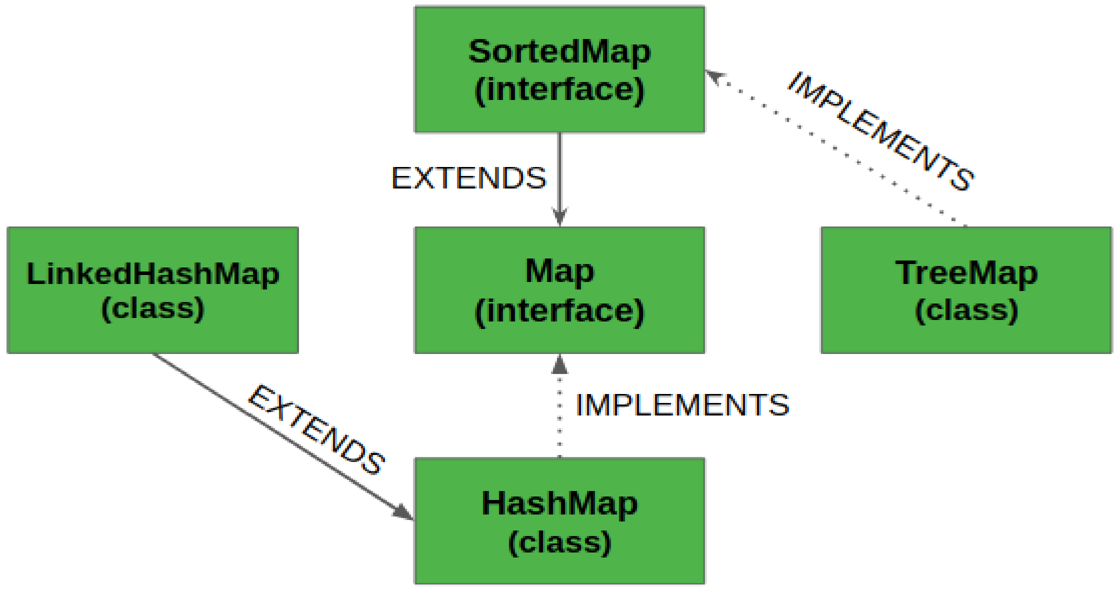
- **Set interface**
  - Set is an interface which extends Collection. It is an **unordered collection of objects** in which **duplicate values cannot be stored.**
  - Basically, Set is implemented by HashSet, LinkedHashSet or TreeSet (sorted representation).
  - We can store at most one null value in Set.
- **HashSet**
  - HashSet class implements Set Interface. It represents the collection that uses a hash table for storage. Hashing is used to store the elements in the HashSet. It contains unique items.
- **LinkedHashSet**
  - LinkedHashSet class represents the LinkedList implementation of Set Interface. It extends the HashSet class and implements Set interface. Like HashSet, It also contains unique elements. It maintains the insertion order and permits null elements.

Collection Interface: Set

- **SortedSet Interface**
  - SortedSet is the alternate of Set interface that provides a total ordering on its elements. The elements of the SortedSet are arranged in the increasing (ascending) order.
  - The SortedSet provides the additional methods that inhibit the natural ordering of the elements.
- **TreeSet**
  - Java TreeSet class implements the Set interface that uses a tree for storage. Like HashSet, TreeSet also contains unique elements. However, the access and retrieval time of TreeSet is **quite fast.** The elements in TreeSet stored in **ascending** order.
  - **Example:** SetExample 1,2,3,4.java

Map Interface

- A map contains values on the **basis of key**, i.e. **key and value pair**. Each key and value pair is known as an entry. A Map **cannot** contain **duplicate keys** and each key can map to at most one value.
- A **Map** is useful if you have to **search, update or delete** elements on the **basis of a key**.
- The **Map interface** is **not** a subtype of the **Collection interface**. Therefore it behaves a bit different from the rest of the collection types.



MAP Hierarchy in Java

Map Interface

Why and When to use Maps?

- Maps are perfect to use for **key-value association** mapping such as **dictionaries**. The maps are used to perform lookups by keys or when someone wants to retrieve and update elements by keys.
- A collection that stores **multiple key-value pairs**
  - **Key**: Unique identifier for each element in a collection
  - **Value**: A value stored in the element associated with the key
- **Some examples are**:
  - A map of error codes and their descriptions.
  - A map of zip codes and cities.
  - A map of managers and employees. Each manager (key) is associated with a list of employees (value) he manages.
  - A map of classes and students. Each class (key) is associated with a list of students (value).

Map Interface: HashMap

- **HashMap** is a Key-Value Pair implementation that implements Map interface and It works similar to Hash Table.
- HashMap **cannot have duplicate keys** but it can have**duplicates as values**.
- Hash Map **can have only one NULL key** and **multiple NULL Values**.
- The Order in which, We add the values to HashMap is not guaranteed, hence the **Insertion order is not Preserved**.
- **Searching the Object is fast** since the Java HashMap has the Keys.
- HashMap is not good for Multi-Threading because it does not support Synchronization.
- **Example:** MapExample1.java, ShoppingCart_HashMap.java, Iterating_ShoppingCart_HashMap.java, SearchingKeys_ShoppingCart_HashMap.java, SearchingValues_ShoppingCart_HashMap.java

Map Interface: HashMap

- **HashMap** is a Key-Value Pair implementation that implements Map interface and It works similar to Hash Table.
- HashMap **cannot have duplicate keys** but it can have**duplicates as values**.
- Hash Map **can have only one NULL key** and **multiple NULL Values**.
- The Order in which, We add the values to HashMap is not guaranteed, hence the **Insertion order is not Preserved**.
- **Searching the Object is fast** since the Java HashMap has the Keys.
- HashMap is not good for Multi-Threading because it does not support Synchronization.
- **Example:** MapExample1.java, ShoppingCart_HashMap.java, Iterating_ShoppingCart_HashMap.java, SearchingKeys_ShoppingCart_HashMap.java, SearchingValues_ShoppingCart_HashMap.java

Map Interface: LinkedHashMap

- Java LinkedHashMap class is Hashtable and Linked list implementation of the Map interface, with predictable iteration order. It inherits HashMap class and implements the Map interface.

Points to remember

- Java LinkedHashMap contains values based on the key.
- Java LinkedHashMap contains unique elements.
- Java LinkedHashMap may have one null key and multiple null values.
- Java LinkedHashMap is non synchronized.
- Java LinkedHashMap maintains insertion order.

- **Example:** LinkedHashMapExample.java

Map Interface: TreeMap

- Java TreeMap class is a red-black tree based implementation. It provides an efficient means of storing key-value pairs in sorted order.

Points to remember

- Java TreeMap contains values based on the key. It implements the NavigableMap interface and extends AbstractMap class.
- Java TreeMap contains only unique elements.
- Java TreeMap cannot have a null key but can have multiple null values.
- Java TreeMap is non synchronized.
- Java TreeMap maintains ascending order.

Example: TreeMapExample.java