Basic data structures

1. Array
2. Graph
3. Hash
4. Linked List
5. Pointers
6. Recursion
7. Stack
8. Queues
9. Structure
10. Tree

Refer MS team recording

1. Array

2. Graph

Refer MS team recording

3. Hash/Hashing/Hash function

Refer MS team recording

4. Linked List

Refer MS team recording

5. Pointers

Refer MS team recording

6. Recursion

Refer MS team recording

7. Stack

Refer MS team recording

8. Queue

Refer MS team recording

Refer MS team recording

9. Structure

Structs are value types while classes are reference types. Structs can be instantiated without using a new operator. A struct cannot inherit from another struct or class, and it cannot be the base of a class.

A structure doesn't support inheritance, and polymorphism, but a class supports both.

A Structure is not secure and cannot hide its implementation details from the end-user while a class is secure and can hide its programming and designing details

Class	Structure
Members of a class are private by default.	Members of a structure are public by default.
Memory allocation happens on the heap.	Memory allocation happens on a stack.
It is a reference type data type.	It is a value type data type.
It is declared using the **class** keyword.	It is declared using the **struct** keyword.

10. Tree

Refer MS team recording

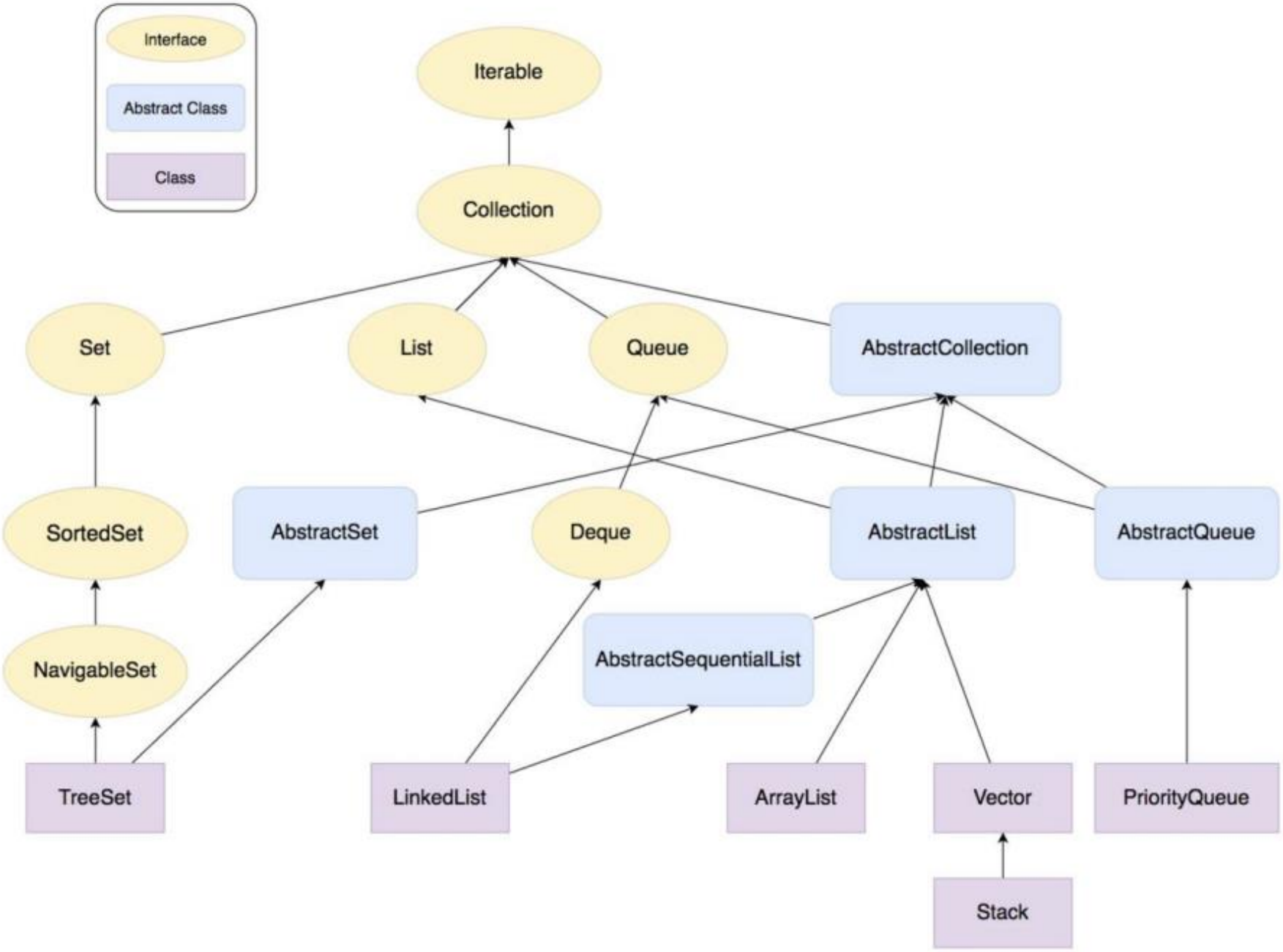
The Java Collections Framework is like the C++ Standard Template Library: "a unified architecture for representing and manipulating collections (objects that group multiple elements into a single unit)."

All of the core collections featured in the Java Collections Framework are already present in core Python.

As it turns out, the equivalent to the Java Collections Framework in Python is... Python. All of the core collections featured in the Java Collections Framework are already present in core Python.
Give it a try! Sequences provide lists, queues, stacks, etc. Dictionaries are your hash-tables and maps. Sets are present, etc.
One might consider Python a "higher" language than Java, because it natively provides all of these higher order abstract data types intrinsically. (It also supports Object Oriented, procedural, and functional programming methodologies.)

NOTE If you are familiar with C++, then you will find it helpful to know that the Java collections technology is similar in spirit to the Standard Template Library (STL) defined by C++. What C++ calls a container, Java calls a collection. However, there are significant differences between the Collections Framework and the STL. It is important to not jump to conclusions.

The Collections Framework hierarchy is as follows:



CSE2005 (Object Oriented Programming Systems)

Module 4
The Collections Framework and Generic Programming

Dr. Arundhati Das

Java's most powerful subsystems: the *Collections Framework*

- "The Collections Framework is a sophisticated hierarchy of interfaces and classes that provide state-of-the-art technology for managing groups of objects."-Java complete reference book
- Why Collection Framework? What is the advantage?
  - Java provides many data structures implemented in pre-defined interfaces and classes, so that you don't have to write them from scratch. This will help in solving and focusing on a bigger problem.
    - Already implemented in an optimized way
    - Less computing time

NOTE If you are familiar with C++, then you will find it helpful to know that the Java collections technology is similar in spirit to the Standard Template Library (STL) defined by C++. What C++ calls a container, Java calls a collection. However, there are significant differences between the Collections Framework and the STL. It is important to not jump to conclusions.

All of the core collections featured in the Java Collections Framework are already present in core Python.

Collections framework contd.

- The data structures are:
  - It defines a collections framework as “a unified architecture for representing and manipulating collections, allowing them to be manipulated independent of the details of their representation.”
  - It defines a “collection” as “an object that represents a group of objects”.

  - **The java.util package contains all the classes and interfaces for the Collection framework,** Java’s most powerful subsystems**.**

Collections framework contd.

- In JDK 1.2 onwards, Java developers introduced the Collections Framework, which is an important framework to help you achieve all of your data operations.

- Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

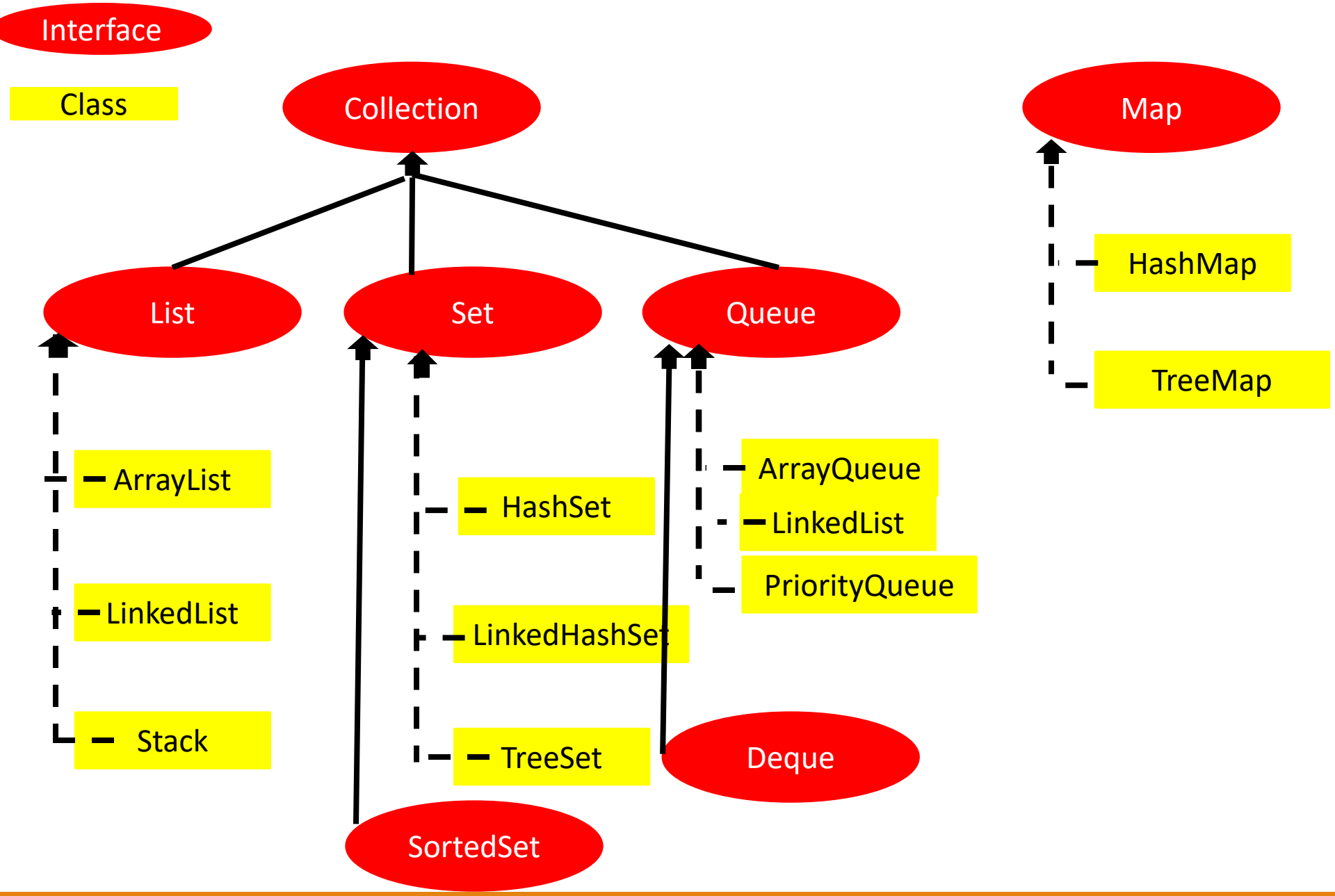
- Java Collection framework provides framework provides many interfaces (Set, List, Queue, Deque)

- classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

[Try to understand the collection framework and collection are two things, collection is an interface under the framework of collection. Collection framework contain other interfaces such as map and iterator]

Collections framework contd.

Collection Interfaces - Represent different types of collections, such as sets, lists, queue and maps. These interfaces form the basis of the framework.
General-purpose Implementations - Primary implementations of the collection interfaces.
Legacy Implementations - The collection classes from earlier releases, Vector and Hashtable, have been retrofitted to implement the collection interfaces.
Wrapper Implementations - Add functionality, such as synchronization, to other implementations.
Convenience Implementations - High-performance "mini-implementations" of the collection interfaces.
Abstract Implementations - Partial implementations of the collection interfaces to facilitate custom implementations.
Algorithms - Static methods that perform useful functions on collections, such as sorting a list.
Infrastructure - Interfaces that provide essential support for the collection interfaces.
Array Utilities - Utility functions for arrays of primitives and reference objects. Not, strictly speaking, a part of the Collections Framework, this functionality is being added to the Java platform at the same time and relies on some of the same infrastructure.

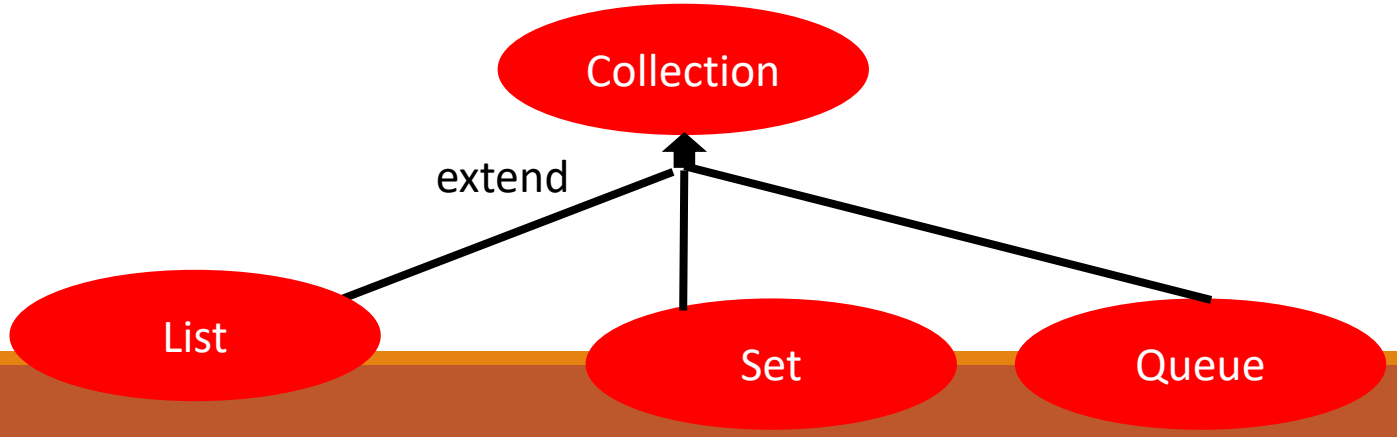

Interface
Class
Collection
Map
List
Set
Queue
HashMap
TreeMap
ArrayList
LinkedList
Stack
HashSet
LinkedHashSet
TreeSet
ArrayQueue
LinkedList
PriorityQueue
Deque
SortedSet

Collection interfaces

The core collection interfaces encapsulate different types of collections. They are interfaces so they do not provide an implementation!

The Collection interface is the interface which is implemented by all the classes in the collection framework.

Some types of collections allow duplicate elements, and others do not. Some are ordered and others are unordered.



Methods of Collection interface

Method	Description
boolean add(E *obj*)	Adds *obj* to the invoking collection. Returns **true** if *obj* was added to the collection. Returns **false** if *obj* is already a member of the collection and the collection does not allow duplicates.
boolean addAll(Collection<? extends E> *c*)	Adds all the elements of *c* to the invoking collection. Returns **true** if the collection changed (i.e., the elements were added). Otherwise, returns **false**.
void clear( )	Removes all elements from the invoking collection.
boolean contains(Object *obj*)	Returns **true** if *obj* is an element of the invoking collection. Otherwise, returns **false**.
boolean containsAll(Collection<?> *c*)	Returns **true** if the invoking collection contains all elements of *c*. Otherwise, returns **false**.
boolean equals(Object *obj*)	Returns **true** if the invoking collection and *obj* are equal. Otherwise, returns **false**.
int hashCode( )	Returns the hash code for the invoking collection.
boolean isEmpty( )	Returns **true** if the invoking collection is empty. Otherwise, returns **false**.
Iterator<E> iterator( )	Returns an iterator for the invoking collection.
default Stream<E> parallelStream( )	Returns a stream that uses the invoking collection as its source for elements. If possible, the stream supports parallel operations. (Added by JDK 8.)
boolean remove(Object *obj*)	Removes one instance of *obj* from the invoking collection. Returns **true** if the element was removed. Otherwise, returns **false**.
boolean removeAll(Collection<?> *c*)	Removes all elements of *c* from the invoking collection. Returns **true** if the collection changed (i.e., elements were removed). Otherwise, returns **false**.
default boolean removeIf(Predicate<? super E> *predicate*)	Removes from the invoking collection those elements that satisfy the condition specified by *predicate*. (Added by JDK 8.)

Table 18-1 The Methods Declared by **Collection**

public interface Collection<E> extends Iterable<E>

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 iterator().

Collection — the root of the collection hierarchy. A collection represents a group of objects known as its *elements*. The Collection interface is the least common denominator that all collections implement and is used to pass collections around and to manipulate them when maximum generality is desired. Some types of collections allow duplicate elements, and others do not. Some are ordered and others are unordered. The Java platform doesn't provide any direct implementations of this interface but provides implementations of more specific subinterfaces, such as Set and List.

public interface Collection<E> extends Iterable<E>

```
public interface Collection<E> extends Iterable<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);          //optional
    boolean remove(Object element); //optional
    Iterator<E> iterator();
    // Bulk operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);
//optional
    boolean removeAll(Collection<?> c);
//optional
    boolean retainAll(Collection<?> c);
//optional
    void clear();
//optional

    // Array operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

A note on iterators

- for loop
- For each loop

  Iterator interface provides the facility of iterating the elements in a forward direction only.

- An `Iterator` is an object that enables you to traverse through a collection and to remove elements from the collection selectively, if desired. You get an `Iterator` for a collection by calling its `iterator()` method. The following is the `Iterator` interface.

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); //optional
}
```

public interface **List<E>** extends Collection<E>

```
public interface List<E> extends Collection<E> {
    // Positional access
    E get(int index);
    E set(int index, E element);    //optional
    boolean add(E element);         //optional
    void add(int index, E element); //optional
    E remove(int index);            //optional
    boolean addAll(int index,
        Collection<? extends E> c); //optional

    // Search
    int indexOf(Object o);
    int lastIndexOf(Object o);

    // Iteration
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);

    // Range-view
    List<E> subList(int from, int to);
}
```

public interface **List<E>** extends Collection<E>

List

- an ordered collection (sometimes called a *sequence*).
- Lists can contain duplicate elements.
- The user of a List generally has precise control over where in the list each element is inserted and can access elements by their integer index (position).
- Example implementations are LinkedList (linked list based) and ArrayList (dynamic array based)

A note on ListIterators

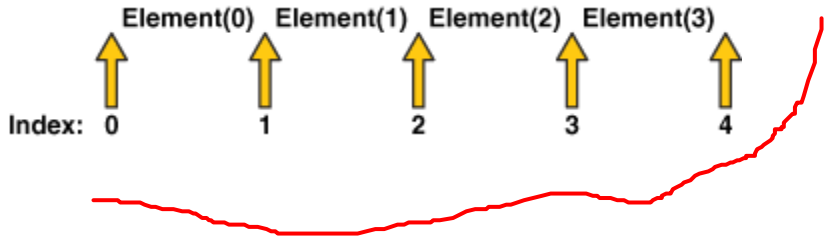
The three methods that ListIterator inherits from Iterator (hasNext, next, and remove) do exactly the same thing in both interfaces. The hasPrevious and the previous operations are exact analogues of hasNext and next. The former operations refer to the element before the (implicit) cursor, whereas the latter refer to the element after the cursor. The previous operation moves the cursor backward, whereas next moves it forward.

The nextIndex method returns the index of the element that would be returned by a subsequent call to next, and previousIndex returns the index of the element that would be returned by a subsequent call to previous

The set method overwrites the last element returned by next or previous with the specified element.

The add method inserts a new element into the list immediately before the current cursor position.

```
public interface ListIterator<E> extends Iterator<E> {
    boolean hasNext();
    E next();
    boolean hasPrevious();
    E previous();
    int nextIndex();
    int previousIndex();
    void remove(); //optional
    void set(E e); //optional
    void add(E e); //optional
}
```



here is **no current element** in ListIterator. Its cursor always lies between the previous and next elements. The **previous()** will return to the previous elements and the **next()** will return to the next element. Therefore, for a list of n length, there are n+1 possible cursors.

public interface **Set<E>** extends Collection<E>

Set

- a collection that cannot contain duplicate elements.

- This interface models the mathematical set abstraction and is used to represent sets, the courses making up a student's schedule, or the processes running on a machine.

- Example implementations of Set interface are HashSet (Hashing based) and TreeSet (balanced BST based).

public interface **Set<E>** extends Collection<E>

```
public interface Set<E> extends Collection<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);          //optional
    boolean remove(Object element); //optional
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c); //optional
    boolean removeAll(Collection<?> c);        //optional
    boolean retainAll(Collection<?> c);        //optional
    void clear();                              //optional

    // Array Operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

public interface **SortedSet<E>** extends Set<E>

SortedSet — a Set that maintains its elements in ascending order. Several additional operations are provided to take advantage of the ordering. Sorted sets are used for naturally ordered sets, such as word lists and membership rolls.

```
public interface SortedSet<E> extends Set<E> {
    // Range-view
    SortedSet<E> subSet(E fromElement, E toElement);
    SortedSet<E> headSet(E toElement);
    SortedSet<E> tailSet(E fromElement);

    // Endpoints
    E first();
    E last();

    // Comparator access
    Comparator<? super E> comparator();
}
```

public interface **Queue<E>** extends Collection<E>

Queue

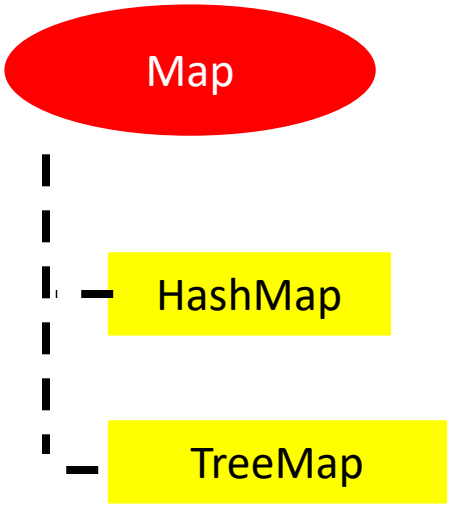
- Queue typically order elements in FIFO order except exceptions like PriorityQueue and Deque.
- In PriorityQueue, elements are deleted based on the priority.
- In Deque, Elements can be inserted and removed at both ends. Allows both LIFO and FIFO.
- a collection of Queue is used to hold multiple elements prior to processing.
- Besides basic Collection operations, a Queue provides additional insertion, extraction, and inspection operations.

```
public interface Queue<E> extends Collection<E> {
    E element();//throws exception if queue is empty
    E peek();                       //null
    boolean offer(E e); //add - bool
    E remove();                     //throws
    E poll();                       //null
}
```

public interface **Map<K,V>**

Map

- an object that maps keys to values.
- A Map cannot contain duplicate keys; each key can map to at most one value.
- If you've used Hashtable, you're already familiar with the basics of Map.
- Example implementation are HashMap and TreeMap. TreeMap implements SortedMap.



The difference between Set and Map interface is that in Set we have only keys, whereas in Map, we have key, value pairs.

public interface Map<K,V>

```
public interface Map<K,V> {

    // Basic operations
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();

    // Bulk operations
    void putAll(Map<? extends K, ? extends V> m);
    void clear();

    // Collection Views
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K,V>> entrySet();

    // Interface for entrySet elements
    public interface Entry {
        K getKey();
        V getValue();
        V setValue(V value);
    }
}
```

public interface **SortedMap<K,V>** extends Map<K,V>

SortedMap — a Map that maintains its mappings in ascending key order. This is the Map analog of SortedSet. Sorted maps are used for naturally ordered collections of key/value pairs, such as dictionaries and telephone directories.

```
public interface SortedMap<K, V> extends Map<K, V>{

    SortedMap<K, V> subMap(K fromKey, K toKey);
    SortedMap<K, V> headMap(K toKey);
    SortedMap<K, V> tailMap(K fromKey);
    K firstKey();
    K lastKey();

    Comparator<? super K> comparator();
}
```

Note on Comparator interface

Comparator is another interface (in addition to Comparable) provided by the Java API which can be used to order objects.

You can use this interface to define an order that is different from the Comparable (natural) order.

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

However, there are many differences between Comparable and Comparator interfaces that are given below.

Comparable and Comparator

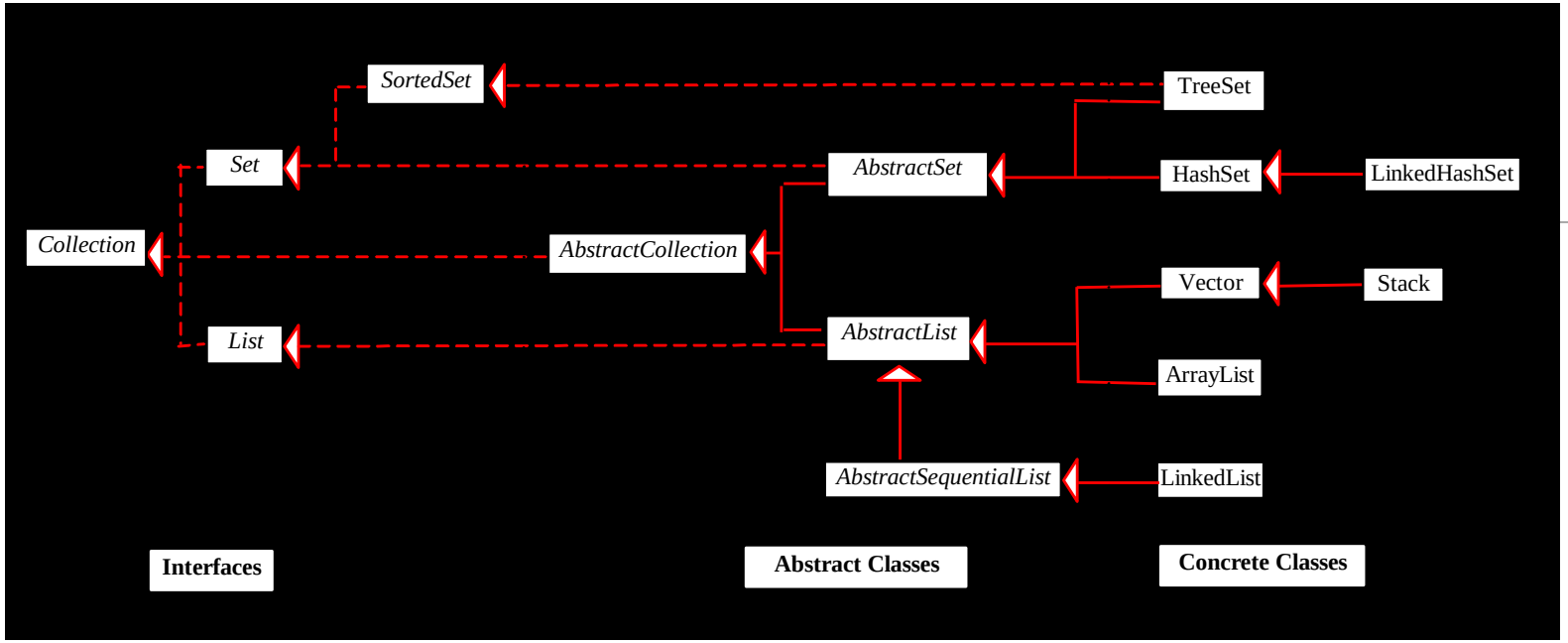
Comparable	Comparator
1) Comparable provides a **single sorting sequence**. In other words, we can sort the collection on the basis of a single element such as id, name, and price.	The Comparator provides **multiple sorting sequences**. In other words, we can sort the collection on the basis of multiple elements such as id, name, and price etc.
2) Comparable **affects the original class**, i.e., the actual class is modified.	Comparator **doesn't affect the original class**, i.e., the actual class is not modified.
3) Comparable provides **compareTo() method** to sort elements.	Comparator provides **compare() method** to sort elements.
4) Comparable is present in **java.lang** package.	A Comparator is present in the **java.util** package.
5) We can sort the list elements of Comparable type by **Collections.sort(List)** method.	We can sort the list elements of Comparator type by **Collections.sort(List, Comparator)** method.

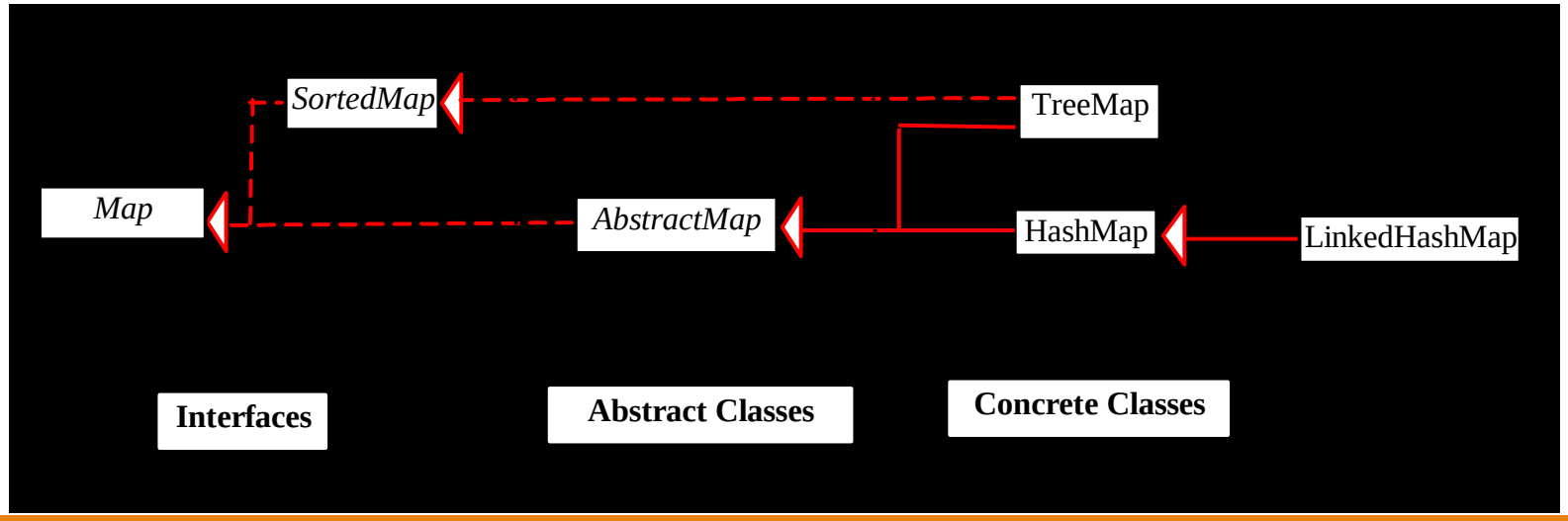
General-purpose Implementations

Interfaces	Implementations				
	Hash table	**Resizable array**	**Tree (sorted)**	**Linked list**	**Hash table + Linked list**
Set	**HashSet**		TreeSet **(sorted)**		LinkedHashSet
List		**ArrayList**		LinkedList	
Queue					
Map	**HashMap**		TreeMap **(sorted)**		LinkedHashMap

Note the naming convention

LinkedList also implements queue and there is a PriorityQueue implementation (implemented with heap)





Making your own implementations

Most of the time you can use the implementations provided for you in the Java API.

In case the existing implementations do not satisfy your needs, you can write your own by extending the abstract classes provided in the collections framework.

implementations

Each of the implementations offers the strengths and weaknesses of the underlying data structure.

What does that mean for:

- Hashtable
- Resizable array
- Tree
- LinkedList
- Hashtable plus LinkedList

Think about these tradeoffs when selecting the implementation!

Lists

- **List** in Java provides the facility to maintain the *ordered collection*.
- It contains the index-based methods to insert, update, delete and search the elements.
- It can have the duplicate elements also. We can also store the null elements in the list.
- The List interface is found in the java.util package and inherits the Collection interface.
- It is a factory of ListIterator interface. Through the ListIterator, we can iterate the list in forward and backward directions.
- The implementation classes of List interface are ArrayList, LinkedList, Stack and Vector.
- The ArrayList and LinkedList are widely used in Java programming. The Vector class is deprecated since Java 5.

```
/Creating a List of type String using ArrayList
List<String> list=new ArrayList<String>();
 //Creating a List of type Integer using ArrayList
List<Integer> list=new ArrayList<Integer>();
 //Creating a List of type Book using ArrayList
List<Book> list=new ArrayList<Book>();
 //Creating a List of type String using LinkedList
List<String> list=new LinkedList<String>();
```

List Methods

Method	Description
void add(int index, E element)	It is used to insert the specified element at the specified position in a list.
boolean add(E e)	It is used to append the specified element at the end of a list.
boolean addAll(Collection<? extends E> c)	It is used to append all of the elements in the specified collection to the end of a list.
boolean addAll(int index, Collection<? extends E> c)	It is used to append all the elements in the specified collection, starting at the specified position of the list.
void clear()	It is used to remove all of the elements from this list.
boolean equals(Object o)	It is used to compare the specified object with the elements of a list.
int hashcode()	It is used to return the hash code value for a list.
E get(int index)	It is used to fetch the element from the particular position of the list.
boolean isEmpty()	It returns true if the list is empty, otherwise false.
int lastIndexOf(Object o)	It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
Object[] toArray()	It is used to return an array containing all of the elements in this list in the correct order.
<T> T[] toArray(T[] a)	It is used to return an array containing all of the elements in this list in the correct order.
boolean contains(Object o)	It returns true if the list contains the specified element
boolean containsAll(Collection<?> c)	It returns true if the list contains all the specified element
int indexOf(Object o)	It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.
E remove(int index)	It is used to remove the element present at the specified position in the list.
boolean remove(Object o)	It is used to remove the first occurrence of the specified element.
boolean removeAll(Collection<?> c)	It is used to remove all the elements from the list.
void replaceAll(UnaryOperator<E> operator)	It is used to replace all the elements from the list with the specified element.
void retainAll(Collection<?> c)	It is used to retain all the elements in the list that are present in the specified collection.
E set(int index, E element)	It is used to replace the specified element in the list, present at the specified position.
void sort(Comparator<? super E> c)	It is used to sort the elements of the list on the basis of specified comparator.
Spliterator<E> spliterator()	It is used to create spliterator over the elements in a list.
List<E> subList(int fromIndex, int toIndex)	It is used to fetch all the elements lies within the given range.
int size()	It is used to return the number of elements present in the list.

Collection: LinkedList

LinkedList implements the Collection interface.

It uses a doubly linked list internally to store the elements.

It can store the duplicate elements.

It maintains the insertion order

Ravi
Vijay
Ravi
Ajay

```
import java.util.*;
public class TestJavaCollection2{
public static void main(String args[]){
LinkedList<String> al=new LinkedList<String
al.add("Ravi");
al.add("Vijay");
al.add("Ravi");
al.add("Ajay");
Iterator<String> itr=al.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Difference between ArrayList and LinkedList

- ArrayList and LinkedList both implements List interface and maintains insertion order.

ArrayList	LinkedList
1) ArrayList internally uses a dynamic array to store the elements.	LinkedList internally uses a doubly linked list to store the elements.
2) Manipulation with ArrayList is slow because it internally uses an array. If any element is removed from the array, all the bits are shifted in memory.	Manipulation with LinkedList is faster than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory.
3) An ArrayList class can act as a list only because it implements List only.	LinkedList class can act as a list and queue both because it implements List and Deque interfaces.
4) ArrayList is better for storing and accessing data.	LinkedList is better for manipulating data.

The eight classes of the *java.lang* package are known as wrapper classes in Java. The list of eight wrapper classes are given below:

Wrapper class

Primitive Data types and their Corresponding Wrapper class

Primitive Data Type	Wrapper Class
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean

A Wrapper class is **a class whose object wraps or contains primitive data types**

When we create an object to a wrapper class, it contains a field and in this field, we can store primitive data types.

In other words, we can wrap a primitive value into a wrapper class object.

Wrapper classes in Java

- The **wrapper class in Java** provides the mechanism *to convert primitive into object and object into primitive*.
- Since J2SE 5.0, **autoboxing** and **unboxing** feature convert primitives into objects and objects into primitives automatically. The automatic conversion of primitive into an object is known as autoboxing and vice-versa unboxing.

Use of Wrapper classes in Java

- Java is an object-oriented programming language, so we need to deal with objects many times like in Collections, Serialization, Synchronization, etc. Let us see the different scenarios, where we need to use the wrapper classes.
- **Change the value in Method:** Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
- **Serialization:** We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
- **Synchronization:** Java synchronization works with objects in Multithreading.
- **java.util package:** The java.util package provides the utility classes to deal with objects.
- **Collection Framework:** Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

Autoboxing

- The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing, for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.
- Since Java 5, we do not need to use the valueOf() method of wrapper classes to convert the primitive into objects.

Before J2SE 5, explicit conversion of primitive types to wrapper objects:

```
public class WrapperExample1{
public static void main(String args[]){
//Converting int into Integer
int a=20;
Integer i=Integer.valueOf(a);//converting int into Integer explicitly
Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally

System.out.println(a+" "+i+" "+j);
}}
```

Unboxing

- The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing. It is the reverse process of autoboxing. Since Java 5, we do not need to use the intValue() method of wrapper classes to convert the wrapper type into primitives.

Vector in Java

ArrayList is not synchronized. Its equivalent synchronized class in Java is Vector.

- The **Vector** class implements a growable array of objects. Vectors basically fall in legacy classes but now it is fully compatible with collections. It is found in the **java.util package** and implements the List interface, so we can use all the methods of List interface here.
- ector implements a dynamic array that means it can grow or shrink as required. Like an array, it contains components that can be accessed using an integer index
- They are very similar to ArrayList but Vector is synchronized and has some legacy method that the collection framework does not contain.
- It also maintains an insertion order like an ArrayList but it is rarely used in a non-thread environment as it is **synchronized** and due to which it gives a poor performance in adding, searching, delete and update of its elements.
- The Iterators returned by the Vector class are fail-fast. In the case of concurrent modification, it fails and throws the **ConcurrentModificationException.**

```
public class WrapperExample {
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		byte b=10;
		short s=20;
		int i=30;
		long l=40;
		float f=50.0F;
		double d=60.0D;
		char c='a';
		boolean b2=true;
		//wrapper classes
		//autoboxing
			Byte byteobj=b;
			Short shortobj=s;
			Integer intobj=i;
			Long longobj=l;
			Float floatobj=f;
			Double doubleobj=d;
			Character charobj=c;
			Boolean boolobj=b2;
		System.out.println(byteobj);
		System.out.println(shortobj);
		System.out.println(intobj);
		//unboxing
		byte b1=byteobj;
		short s1=shortobj;
		int i1=(int)longobj;
		System.out.println(b1);
		System.out.println(i1);
	}}
```

```
mport java.io.*;
import java.util.*;

class VectorExample {

    public static void main(String[] args)
    {
        // Size of the
        // Vector
        int n = 5;

        // Declaring the Vector with
        // initial size n
        Vector<Integer> v = new Vector<Integer>(n);

        // Appending new elements at
        // the end of the vector
        for (int i = 1; i <= n; i++)
            v.add(i);

        // Printing elements
        System.out.println(v);

        // Remove element at index 3
        v.remove(3);

        // Displaying the vector
        // after deletion
        System.out.println(v);

        // Printing elements one by one
        for (int i = 0; i < v.size(); i++)
            System.out.print(v.get(i) + " ");
    }
}
```

Remove first character from a string in JavaScript

1. Using substring() method

The `substring()` method returns the part of the string between the specified indexes or to the end of the string.

```
let str = 'Hello';

str = str.substring(1);
console.log(str);

/*
    Output: ello
*/
```

[List]

Q. Enter 10 names into a list. Write a program to delete the first two characters of the names and arrange the resulting names in:

a) alphabetical ascending order and print them out.

b) alphabetical descending order and print them out.

```
List<String> l1=new ArrayList<>();
l1.add("Hritik");
l1.add("Praise");
l1.add("Aniket");
l1.add(2,"Manasa");
System.out.println(l1);
List<String> l2=new ArrayList<>();
for(String s:l1)
{
s=s.substring(2);
System.out.println(s);
l2.add(s);
}
Collections.sort(l2);
System.out.println("ascending"+l2);
Collections.sort(l2, Collections.reverseOrder());
System.out.println("descending"+l2);
```

[Set]

Q. A set of 10 names are given. Write a program to delete the first three characters of the names and arrange the resulting names in alphabetical order and print them out.

a) alphabetical ascending order and print them out.

b) alphabetical descending order and print them out.

```
Set<String> s1=new HashSet<>();
s1.add("Hritik");
s1.add("Praise");
s1.add("Manasa");
s1.add("Aniket");
System.out.println(s1);
Set<String> s2=new TreeSet<>();
for(String s:s1)
{
s=s.substring(3);
System.out.println(s);
s2.add(s);

}
System.out.println(“ascending
"+s2);
```

```
Set<String> s2=new
TreeSet<>(Comparator.reverseOrder());
for(String s:s1)
{
s=s.substring(3);
System.out.println(s);
s2.add(s);

}
System.out.println(“descending "+s2);
```

2. Using `slice()` method

The `slice()` method extracts the text from a string and returns a new string.

```
let str = 'Hello';

str = str.slice(1);
console.log(str);

/*
    Output: ello
*/
```

3. Using substr() method

The `substr()` method returns a portion of the string, starting at the specified index and extending for a given number of characters or until the string's end.

```
let str = 'Hello';

str = str.substr(1);
console.log(str);

/*
    Output: ello
*/
```

[Stack]

Q. Take 5 user input for name and age and push it into a stack. Now write a method to check if the age of the person on the top of the stack is less than equal to 18. If yes, pop it and add it into one list named “Children”. Else add it into another list “Adult”. Display both the lists. Create an user-defined class Citizen where you can define the parameterized constructor with name and age. (LIFO)

```
Stack<Citizen> ci=new Stack<Citizen>();

ci.push(new Citizen("Amar",20));
ci.push(new Citizen("Akbar",19));
ci.push(new Citizen("Antony",14));
ci.push(new Citizen("Joy", 13));
ci.push(new Citizen("Happy",21));
//System.out.println("stack pop ci "+ci.pop());
System.out.println("stack peek ci "+ci.peek().age);

System.out.println("stack ci "+ci.size());
ArrayList<Citizen> children=new ArrayList();
ArrayList<Citizen> adult=new ArrayList();
if(ci.peek().age>=18)
{
System.out.println("hello from adult");
adult.add(new Citizen(ci.peek().name,ci.peek().age));
ci.pop();
}
else
{
System.out.println("hello from clideren");
children.add(new Citizen(ci.peek().name,ci.peek().age));
ci.pop();
}
```

```
class Citizen
{
String name;
int age;
Citizen(String name, int age)
{
this.name=name;
this.age=age;

System.out.println("name "+name+" age "+age);
}

}
```

[Queue] offer(), poll()

Q. Take 5 user input for name and age and push it into a queue. Now write a method to check if the age of the person on the queue's other end is less than equal to 18. If yes, pop it and add it into one list named "Child". Else add it into another list "Adult". Display both the lists. Create an user-defined class Citizen where you can define the parameterized constructor. (FIFO)

[Exception Handling] WORDLE game

Q. Generate a random 3 digit number using random() method. Ask user to guess 1 digit present in the 3 digit random number. If the digit is present, show its position. If it is not present, throw an user-defined exception showing message “invalid input”. Repeat this step for all 3 digits. If all 3 digits are guessed correctly, display a message “You Win!!!”. If all are not guessed until 5 wrong attempts, display message “You lose”.

```
//how to generate random number and convert it into array

int rand=(int)Math.floor(Math.random()*(1000-100+1)+100);
System.out.println(rand);
 int number = rand;
 String tempString = Integer.toString(number);
 int numbers[] = new int[tempString.length()];
 for (int i = 0; i < tempString.length(); i++) {
    numbers[i] = tempString.charAt(i) - '0';
 }


 System.out.println("digits are      "+numbers[0]+"
"+numbers[1]+"  "+numbers[2]);
```

LinkedList, ListIterator Example

```
package oop;
import java.util.*;
public class LinkedListExample {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
LinkedList<Integer> ll=new LinkedList<Integer>();
ll.add(2);
ll.addFirst(4);
ll.addLast(6);
ll.add(8);
System.out.println(ll);
//int x=ll.remove();
ListIterator<Integer> li=ll.listIterator();
while(li.hasNext())
{
	System.out.println(li.nextIndex());
	System.out.println(li.next());

}
while(li.hasPrevious())
{
	System.out.println(li.previous());
}
	}

}
```

ListIterator is one of the Java cursors. It is a java iterator which is used to traverse all types of lists including ArrayList, Vector, LinkedList, Stack etc. It is available since Java 1.2. It extends the iterator interface.

Methods of ListIterator

Method	Description
void add(E e)	This method inserts the specified element in the list.
boolean hasNext(),	This returns true if the list has more elements to traverse.
boolean hasPrevious()	This returns true if the list iterator has more elements while traversing the list in the backward direction.
E next()	This method returns the next element and increases the cursor by one position.
int nextIndex()	This method returns the index of the element which would be returned on calling the next() method.
E previous()	This method returns the previous element of the list and shifts the cursor one position backwards.
int previousIndex()	This method returns the index of the element which would be returned on calling the previous() method.
void remove()	This method removes the last element from the list that was returned on calling next() or previous() method element from.
void set(E e)	This method replaces the last element that was returned on calling next() or previous() method with the specified element.

```
package oop;
import java.util.*;
public class ListExample {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
List<String> l=new ArrayList<String>();
l.add(“Thor");
l.add(“Ironman");
l.add(“Shaktiman");
l.add(3,“Loki");
ListIterator<String> list=l.listIterator();
while(list.hasNext())
{
	System.out.println(list.nextIndex());
	System.out.println(list.next());
}
while(list.hasPrevious())
{
	System.out.println(list.previousIndex());
	System.out.println(list.previous());
}
//for(String names:l)
//System.out.println(names);
}
}
```

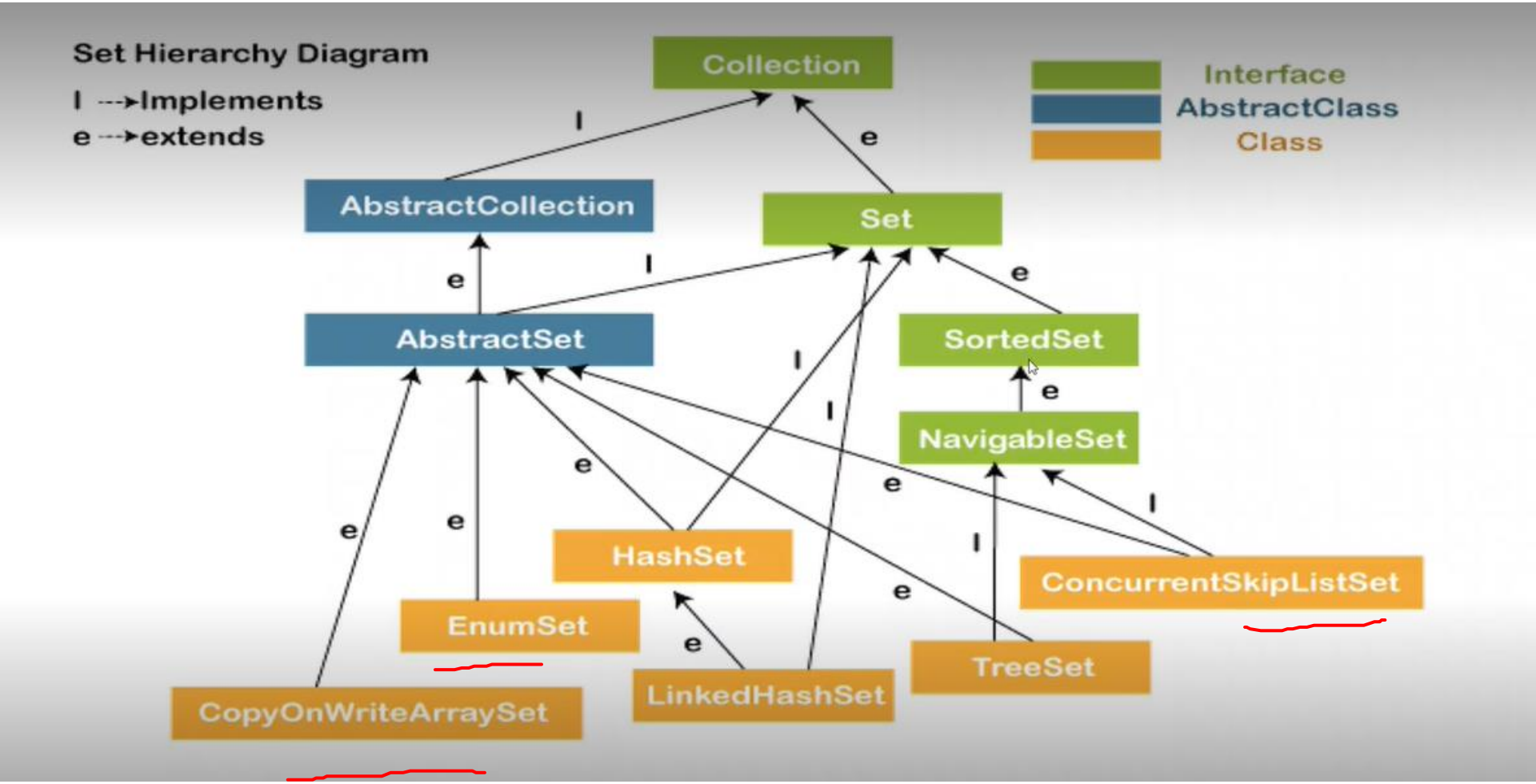
```
package oop;
import java.util.*;
public class VectorEx {

	public static void main(String[] args) {
		// TODO Auto-generated method
stub
Vector<Integer> v=new Vector<Integer>(5);
for(int i=0;i<=5;i++)
	v.add(i);
//System.out.println(v);
//v.remove(3);
//System.out.println(v);
ListIterator<Integer> list=v.listIterator();
while(list.hasNext())
{
	System.out.println(list.next());
}
	}

}
```

```
package oop;
import java.util.*;
public class StackExample {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
Stack<String> st=new Stack<String>();
st.push("deepa");
st.push("surya");
st.push("raghu");
System.out.println(st);
//System.out.println("top of the stack="+st.peek());
//System.out.println("the poped element
is"+st.pop());
//System.out.println("the poped element
is"+st.pop());
ListIterator<String> s=st.listIterator();
while(s.hasNext())
{
	System.out.println(s.next());
}
	}

}
```

Set in Java

- The set interface is present in java.util package and extends the Collection interface is an unordered collection of objects in which duplicate values cannot be stored.
- It is an interface that implements the mathematical set.
- This interface contains the methods inherited from the Collection interface and adds a feature that restricts the insertion of the duplicate elements.
- There are two interfaces that extend the set implementation namely SortedSet and NavigableSet.

In the above image, the navigable set extends the sorted set interface.
Since a set doesn't retain the insertion order, the navigable set interface provides the implementation to navigate through the Set.
The class which implements the navigable set is a TreeSet which is an implementation of a self-balancing tree.
Therefore, this interface provides us with a way to navigate through this tree.


Set Hierarchy Diagram
I --->Implements
e --->extends
Collection
Interface
AbstractClass
Class
AbstractCollection
Set
AbstractSet
SortedSet
NavigableSet
HashSet
ConcurrentSkipListSet
EnumSet
TreeSet
LinkedHashSet
CopyOnWriteArraySet

Operations on the Set Interface

- On the Set, we can perform all the basic mathematical operations like intersection, union and difference.
- Suppose, we have two sets, i.e., set1 = [22, 45, 33, 66, 55, 34, 77] and set2 = [33, 2, 83, 45, 3, 12, 55].
- We can perform the following operation on the Set:
- **Intersection:** The intersection operation returns all those elements which are present in both the set. The intersection of set1 and set2 will be [33, 45, 55].
- **Union:** The union operation returns all the elements of set1 and set2 in a single set, and that set can either be set1 or set2. The union of set1 and set2 will be [2, 3, 12, 22, 33, 34, 45, 55, 66, 77, 83].
- **Difference:** The difference operation deletes the values from the set which are present in another set. The difference of the set1 and set2 will be [66, 34, 22, 77].
- In set, **addAll()** method is used to perform the union, **retainAll()** method is used to perform the intersection and **removeAll()** method is used to perform difference.

```
package oop;
import java.util.*;
public class SetOperationExampke {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
Integer[] x= {34,23,67,12,87,16,45};
Integer[] y= {5,35,34,87,16,79,62,90};
Set<Integer> s1=new HashSet<Integer>();
s1.addAll(Arrays.asList(x));
Set<Integer> s2=new HashSet<Integer>();
s2.addAll(Arrays.asList(y));
//union of two sets
Set<Integer> set_union=new HashSet<Integer>(s1);
set_union.addAll(s2);
System.out.println("union of two sets="+set_union);
//intersection
Set<Integer> set_intersect=new HashSet<Integer>(s1);
set_intersect.retainAll(s2);
System.out.println("intersection values="+set_intersect);
//difference
Set<Integer> set_diff=new
HashSet<Integer>(s1);
set_diff.removeAll(s2);
System.out.println("difference of set1 with
set2="+set_diff);
	}

}
```

```
package oop;
import java.util.*;
public class SetExample {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
Set<Integer> s=new HashSet<Integer>();
s.add(1);
s.add(4);
s.add(6);
s.add(2);

Iterator<Integer> i=s.iterator();
while(i.hasNext())
{
	System.out.println(i.next());
}



	}

}
```

```
package oop;
import java.util.*;
public class SetExample1 {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
TreeSet<String> s=new TreeSet<String>();
s.add("sneha");
s.add("Jathin");
s.add("Jathin");
s.add("sriya");
s.add("ankur");
s.add("harish");
Iterator<String> i=s.iterator();
while(i.hasNext())
{
	System.out.println(i.next());
}
System.out.println(s.hashCode());
s.clear();
System.out.println(s);
System.out.println(s.isEmpty());
	}}
```

Map: more examples

```
package oop;
import java.util.*;
public class HashMapEx {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
HashMap<String,Integer> s=new HashMap<String,Integer>();
s.put("a",100);
s.put("b",101);
s.put("c",200);
s.put("a",102);
//System.out.println(s);
for(Map.Entry e:s.entrySet())
{
	System.out.println(e.getKey()+" "+e.getValue());
}
	}

}
```

Map: more examples

```
package oop;
import java.util.*;
public class HashsetExample {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
TreeSet<String> s=new TreeSet<String>();
s.add("Ganesh");

s.add("Surya");
s.add("Ganesh");
s.add("pranita");
Iterator<String> i=s.iterator();
while(i.hasNext())
{
	System.out.println(i.next());
}
System.out.println(s.contains("Surya"));
System.out.println(s.hashCode());
s.clear();
System.out.println(s);
	}

}
```

Generics in Java

- The **Java Generics** programming is introduced in J2SE 5 to deal with type-safe objects. It makes the code stable by detecting the bugs at compile time.
- Before generics, we can store any type of objects in the collection, i.e., non-generic. Now generics force the java programmer to store a specific type of objects

A generic type can be defined for a class or interface.

A concrete type must be specified when using the class to create an object or using the class or interface to declare a reference variable.

Advantage of Java Generics

- There are mainly 3 advantages of generics. They are as follows:

1) **Type-safety:** We can hold only a single type of objects in generics. It doesnt allow to store other objects.

```
//without generics
List list = new ArrayList();
list.add(10);
list.add("10");
//With Generics, it is required to specify the type of object we need to store.

List<Integer> list = new ArrayList<Integer>();
list.add(10);
list.add("10");// compile-time error
```

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

```
//without generics
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);//typecasting
//After Generics, we don't need to typecast the object.
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);
```

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.

```
//with generics
List<String> list = new ArrayList<String>();
list.add("hello");
list.add(32);//Compile Time Error
//with generics
ArrayList<String>
```

Generic class

- A class that can refer to any type is known as a generic class. Here, we are using the T type parameter to create the generic class of specific type.
- Let's see a simple example to create and use the generic class.
- Creating a generic class:

```
class MyGen<T>{
T obj;
void add(T obj){this.obj=obj;}
T get(){return obj;}
}
```

```
class TestGenerics3{
public static void main(String args[]){
MyGen<Integer> m=new MyGen<Integer>();
m.add(2);
//m.add("vivek");//Compile time error
System.out.println(m.get());
}}
```

Type Parameters

1.T - Type

2.E - Element

3.K - Key

4.N - Number

5.V - Value

Generic Method

- Like the generic class, we can create a generic method that can accept any type of arguments. Here, the scope of arguments is limited to the method where it is declared. It allows static as well as non-static methods.
- Let's see a simple example of java generic method to print array elements. We are using here **E** to denote the element.

A generic type can be defined for a static method.

You can also use generic types to define generic methods.

To declare a generic method, you place the generic type <E> immediately after the keyword static in the method header. For ex:

```
public static <E> void print(E[] list)
```

To invoke a generic method, prefix the method name with the actual type in angle brackets. For ex:

```
GenericMethodDemo.<Integer>print(integers);

GenericMethodDemo.<String>print(strings);
```

or simply invoke it as follows:

```
print(integers);

print(strings);
```

```
package oop;


public class TestGeneric {
public static < E > void printArray(E[] ele)
{
	for(E element:ele)
	{
		System.out.println(element);
	}
}

	public static void main(String[] args) {
		// TODO Auto-generated method stub
Integer[] iArray= {1,2,3,4,5,6};
Character[] cArray= {'V','I','T','A','P'};
System.out.println("integer array");
printArray(iArray);
System.out.println("character array");
printArray(cArray);
	}

}
```

Bounded types with generics in Java

- 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.

In the preceding examples, the type parameters could be replaced by any class type.

This is fine for many purposes, but sometimes it is useful to limit the types that can be passed to a type parameter.

A generic type can be specified as a subtype of another type. Such a generic type is called **bounded**.

When specifying a type parameter, you can create an upper bound that declares the superclass from which all type arguments must be derived.

This is accomplished through the use of an extends clause when specifying the type parameter, as shown here:

<T extends superclass>

This specifies that T can only be replaced by superclass, or subclasses of superclass. Thus, superclass defines an inclusive, upper limit.

Defining bounded-types for class

You can declare a bound parameter just by extending the required class with the type-parameter, within the angular braces as −

```
class Sample <T extends Number>
```

```
class Sample <T extends Number>
{
T data;
Sample(T data){
this.data = data;     }
public void display() {
System.out.println("Data value is: "+this.data);     } }
public class BoundsExample {
public static void main(String args[]) {
Sample<Integer> obj1 = new Sample<Integer>(20);
obj1.display();
Sample<Double> obj2 = new Sample<Double>(20.22d);
obj2.display();
Sample<Float> obj3 = new Sample<Float>(125.332f);
obj3.display();     } }
```

```
public      class Sample <T
extends Number>
            {
            T data;
            Sample(T data)
            {
            this.data= data;
            }
            public void display()
{

      System.out.println("Data
value is: "+this.data);
            }
            }
```

```
public class BoundsExample {

            public static void main(String args[]) {
            Sample<Integer> obj1 = new Sample<Integer>(20);
            obj1.display();
            Sample<Double> obj2 = new
Sample<Double>(20.22d);
            obj2.display();
            Sample<Float> obj3 = new
Sample<Float>(125.332f);
            obj3.display();
      //    Sample<String> obj4=new Sample<String>("vitap");
            }



}
```

Defining bounded-types for methods

- Just like with classes to define bounded-type parameters for generic methods, specify them after the extend keyword.
- If you pass a type which is not sub class of the specified bounded-type an error will be generated.

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;
import java.util.Iterator;
public class GenericMethod {
public static <T extends Collection<Integer>>
void sampleMethod(T ele){
Iterator<Integer> it = ele.iterator();
while (it.hasNext()) {
System.out.println(it.next());        }     }
public static void main(String args[]) {
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(24);
list.add(56);
list.add(89);
list.add(75);
list.add(36);
sampleMethod(list);     } }
```

```
import java.util.*;
public class GenericMethod {

        public static <T extends Collection<Integer>> void
sampleMethod(T ele){
        Iterator<Integer> it = ele.iterator();
        while (it.hasNext()) {
        System.out.println(it.next());
        }}
        public static void main(String args[]) {
        ArrayList<Integer> list = new ArrayList<Integer>();
        list.add(24);
        list.add(56);
        list.add(89);
        list.add(75);
        list.add(36);
        sampleMethod(list);
        Integer[] a= {1,2,3,4};
        sampleMethod(a); //error
}}
```

Wildcard in Java Generics

- The ? (question mark) symbol represents the wildcard element. It means any type. If we write <? extends Number>, it means any child class of Number, e.g., Integer, Float, and double. Now we can call the method of Number class through any child class object.
- We can use a wildcard as a **type of a parameter, field, return type, or local variable. However, it is not allowed to use a wildcard as a type argument for a generic method invocation, a generic class instance creation, or a supertype**.

```
import java.util.*;
abstract class Shape{
abstract void draw();
}
class Rectangle extends Shape
{
        void draw() {System.out.println("drawing rectangle");}}
class Circle extends Shape{
void draw(){System.out.println("drawing circle");}}
class GenericTest{
//creating a method that accepts only child class of Shape
public static void drawShapes(List<? extends Shape> lists){
for(Shape s:lists){
s.draw();//calling method of Shape class by child class instance }}
public static void main(String args[])
{
List<Rectangle> list1=new ArrayList<Rectangle>();
list1.add(new Rectangle());
List<Circle> list2=new ArrayList<Circle>();
list2.add(new Circle());
list2.add(new Circle());
drawShapes(list1);
drawShapes(list2);
}}
```

A wildcard generic type has three forms: **?** and **? extends T**, as well as **? super T**, where **T** is a generic type.

The first form, **?** , called an **unbounded wildcard**, is the same as **? Extends Object**.

The second form, **? extends T**, called a **bounded wildcard**, represents **T** or a **subtype of T**.

The third form, **? super T**, called a **lower-bound wildcard**, denotes **T** or a **supertype of T**.

Upper Bounded Wildcards

- The purpose of upper bounded wildcards is to decrease the restrictions on a variable.
- It restricts the unknown type to be a specific type or a subtype of that type. It is used by declaring wildcard character ("?") followed by the extends (in case of, class) or implements (in case of, interface) keyword, followed by its upper bound.
- **?** is a wildcard character.
- **extends**, is a keyword.
- **Number**, is a class present in java.lang package
- Suppose, we want to write the method for the list of Number and its subtypes (like Integer, Double). Using **List<? extends Number>** is suitable for a list of type Number or any of its subclasses whereas **List<Number>** works with the list of type Number only. So, **List<? extends Number>** is less restrictive than **List<Number>**.

```
import java.util.ArrayList;
 public class UpperBoundWildcard {
    private static Double add(ArrayList<? extends Number> num) {
     double sum=0.0;
     for(Number n:num)       {
        sum = sum+n.doubleValue();
     }
       return sum;    }
    public static void main(String[] args) {
      ArrayList<Integer> l1=new ArrayList<Integer>();
     l1.add(10);
     l1.add(20);
     System.out.println("displaying the sum= "+add(l1));
     ArrayList<Double> l2=new ArrayList<Double>();
     l2.add(30.0);
     l2.add(40.0);
     System.out.println("displaying the sum= "+add(l2));
      }
}
```

Lower Bounded Wildcards

- The purpose of lower bounded wildcards is to restrict the unknown type to be a specific type or a supertype of that type. It is used by declaring wildcard character ("?") followed by the super keyword, followed by its lower bound.
- List<? super Integer>
- Here,
- **?** is a wildcard character.
- **super**, is a keyword.
- **Integer**, is a wrapper class.
- Suppose, we want to write the method for the list of Integer and its supertype (like Number, Object). Using **List<? super Integer>** is suitable for a list of type Integer or any of its superclasses whereas **List<Integer>** works with the list of type Integer only. So, **List<? super Integer>** is less restrictive than **List<Integer>**.

```
import java.util.Arrays;
import java.util.List;
 public class LowerBoundWildcard {
    public static void addNumbers(List<? super Integer> list) {
      for(Object n:list)
     {
         System.out.println(n);
     }

  }
public static void main(String[] args) {
       List<Integer> l1=Arrays.asList(1,2,3);
    System.out.println("displaying the Integer values");
  addNumbers(l1);

  List<Number> l2=Arrays.asList(1.0,2.0,3.0);
    System.out.println("displaying the Number values");
  addNumbers(l2);
}

}
```

Unbounded Wildcards

- The unbounded wildcard type represents the list of an unknown type such as List<?>. This approach can be useful in the following scenarios: -
- When the given method is implemented by using the functionality provided in the Object class.
- When the generic class contains the methods that don't depend on the type parameter.

```
import java.util.List;
 public class UnboundedWildcard {
    public static void display(List<?> list)
  {
      for(Object o:list)
     {
        System.out.println(o);
     }
  }
 public static void main(String[] args)
{
  List<Integer> l1=Arrays.asList(1,2,3);
   System.out.println("displaying the Integer values");
   display(l1);
   List<String> l2=Arrays.asList("One","Two","Three");
    System.out.println("displaying the String values");
     display(l2);
  }
 }
```

Generic Constructors

It is also possible for constructors to be generic, even if their class is not.

For example, consider the short program given in **GenericsConstructor1.java** file.

Because **GenCons()** specifies a parameter of a generic type, which must be a subclass of **Number**, **GenCons()** can be called with any numeric type, including **Integer, Float, or Double.**

Therefore, even though **GenCons** is not a generic class, its constructor is generic.

```
// NOTE: The code is taken from The Complete Reference Book, Java, Seventh Edition.
// Use a generic constructor.
class GenCons
{private double val;
<T extends Number> GenCons(T arg) {
val = arg.doubleValue();}
void showval()
{System.out.println("val: " + val);}}
class GenericsConstructor1 {
public static void main(String args[]) {
GenCons test = new GenCons(100);
GenCons test2 = new GenCons(123.5F);
test.showval();
test2.showval();}}
```

Generic Interfaces

Generic interfaces are specified just like generic classes.

The generalized syntax for a generic interface:

```
interface interface-name<type-param-list> { // ...
```

Here, type-param-list is a comma-separated list of type parameters.

```
// NOTE: The code is taken from The Complete Reference Book, Java, Seventh Edition.
// A generic interface example.
// A Min/Max interface.
interface MinMax<T extends Comparable<T>> {
T min();
T max();
}
// Now, implement MinMax
class MyClass<T extends Comparable<T>> implements MinMax<T> {
T[]  vals;
MyClass(T[] o) {
vals = o;
}
// Return the minimum value in vals.
public T min() {
T v = vals[0];
for(int i=1; i < vals.length; i++)
if(vals[i].compareTo(v) < 0)
v = vals[i];
return v;
}
```

```
// Return the maximum value in vals.
public T max() {
T v = vals[0];
for(int i=1; i < vals.length; i++)
{
if(vals[i].compareTo(v) > 0){
//System.out.println(vals[i].compareTo(v));
v = vals[i];
}
}
return v;
}
}
class GenericsInterface1 {public static void main(String args[]) {
Integer inums[] = {3, 6, 2, 8, 6 };
Character chs[] = {'b', 'r', 'p', 'w' };
MyClass<Integer> iob = new MyClass<Integer>(inums);
MyClass<Character> cob = new MyClass<Character>(chs);
System.out.println("Max value in inums: " + iob.max());
System.out.println("Min value in inums: " + iob.min());
System.out.println("Max value in chs: " + cob.max());
System.out.println("Min value in chs: " + cob.min());
}}
```