Unit 4

- Collection Framework
- Introduction, util Package interfaces, List, Set, Map, List interface & its classes, Set interface & its classes, Map interface & its classes

Collections

- A collection is an object that groups multiple elements into a single unit
- Very useful
  - store, retrieve and manipulate data
  - transmit data from one method to another
  - data structures and methods written by hotshots in the field

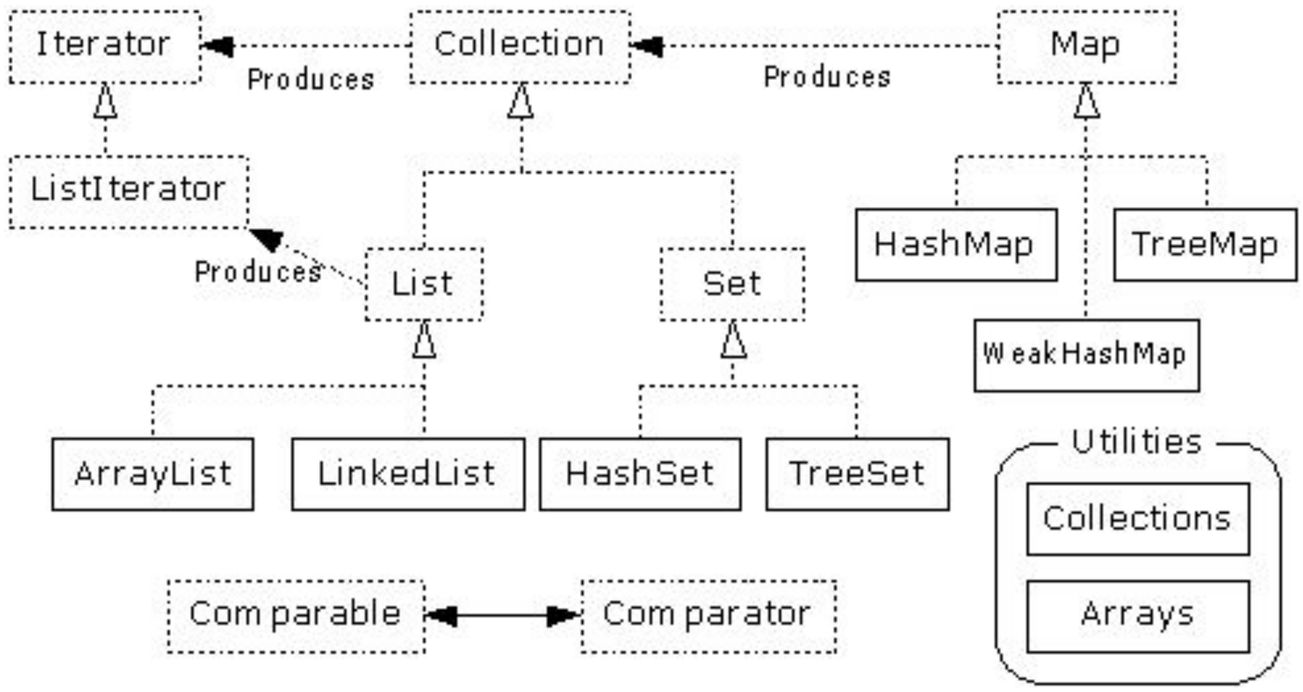
Collections Framework

- Unified architecture for representing and manipulating collections.
- A collections framework contains three things
  - Interfaces
  - Implementations
  - Algorithms

Why Collection Framework?

Before the Collection Framework was introduced in JDK 1.2, Java's approach to collections was using Arrays, Vectors, and Hash tables lacked a common interface. This meant each type of collection had its own set of methods, syntax, and constructors, with no standardization or correlation between them.
This made it difficult for users to remember the diverse functionalities of each collection type and hindered code consistency and reusability. The disparate nature of these collections highlighted the need for a unified Collection Framework to simplify and standardize collection operations in Java.

Collections Framework Diagram



•Interfaces, Implementations, and Algorithms
•From Thinking in Java, page 462

Advantages of the Java Collection Framework

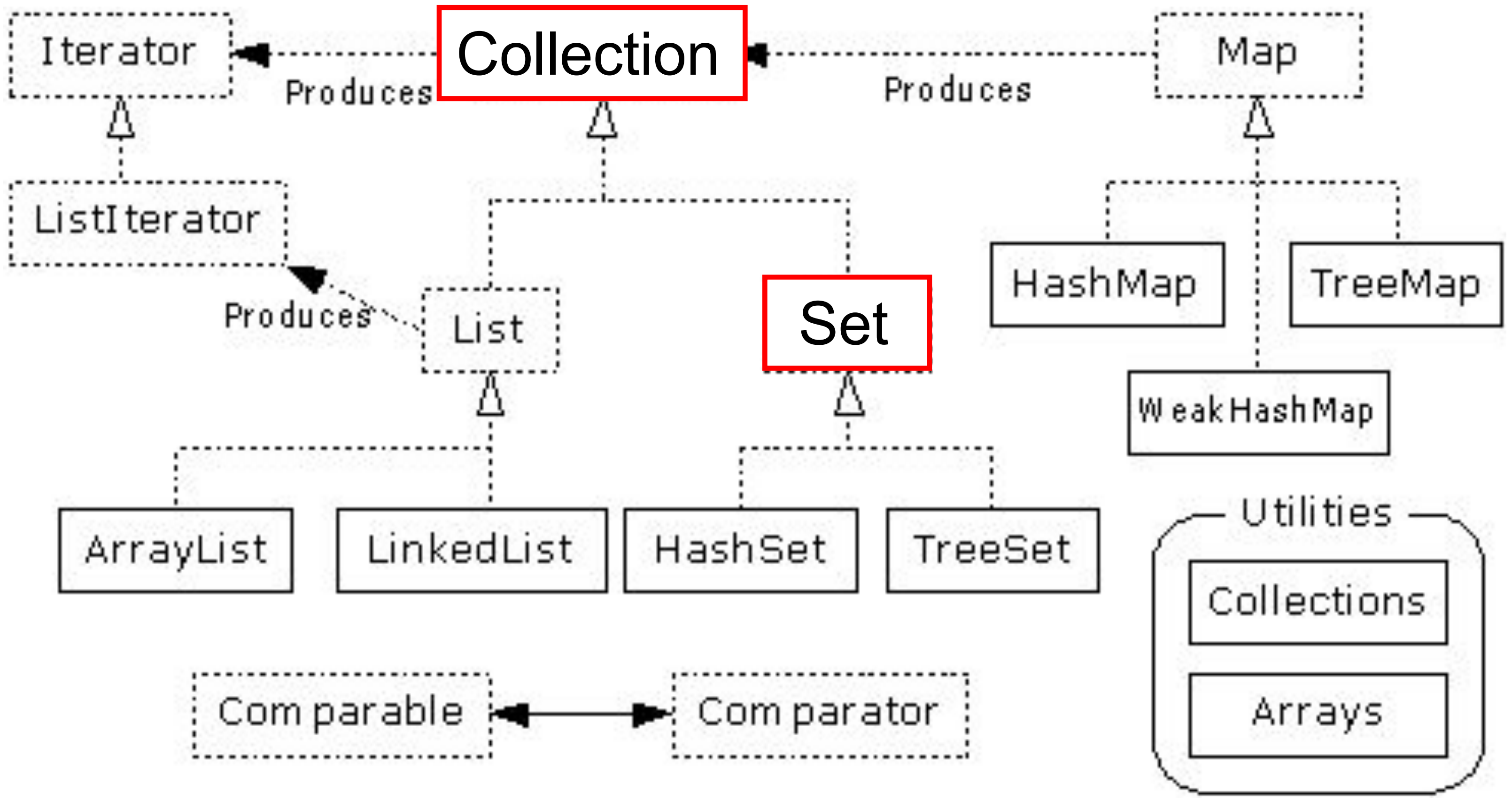
1. Reusability
2. Quality
3. Speed.
4. Maintenance
5. Reduces

Collection Interface

The Collection interface in Java is a core member of the Java Collections Framework located in the java.util package. It is one of the root interfaces of the Java Collection Hierarchy. The Collection interface is not directly implemented by any class. Instead, it is implemented indirectly through its sub-interfaces like List, Queue, and Set.

- Defines fundamental methods
  - int size();
  - boolean isEmpty();
  - boolean contains(Object element);
  - boolean add(Object element);   // Optional
  - boolean remove(Object element); // Optional
  - Iterator iterator();
- These methods are enough to define the basic behavior of a collection
- Provides an Iterator to step through the elements in the Collection

Set Interface Context



Set Interface

- Same methods as Collection
  - different contract - no duplicate entries
- Defines two fundamental methods
  - boolean add(Object o) - reject duplicates
  - Iterator iterator()
- Provides an Iterator to step through the elements in the Set
  - No guaranteed order in the basic Set interface
  - There is a SortedSet interface that extends Set

Iterator Interface

- Defines three fundamental methods
  - Object next()
  - boolean hasNext()
  - void remove()
- These three methods provide access to the contents of the collection
- An Iterator knows position within collection
- Each call to next() "reads" an element from the collection
  - Then you can use it or remove it

Limitations of Enumeration Interface:

An Iterator interface is used in place of Enumeration in Java Collection.

- **Enumeration is not a universal iterator and is used for legacy classes like Vector, Hashtable only.**
- Iterator **allows the caller to remove elements from the given collection** during iterating over the elements.
- Only **forward direction iteration is possible in an Enumeration**.

Advantages of Java Iterator

- The user **can apply these iterators to any of the classes** of the Collection framework.
- In Java Iterator, we can **use both of the read and remove operations**.
- If a user is working **with a for loop, they cannot modernize(add/remove) the Collection**, whereas, **if they use the Java Iterator, they can simply update the Collection.**
- The Java Iterator is considered the **Universal Cursor** for the Collection API.
- The method names in the Java Iterator are very easy and are very simple to use

Disadvantages of Java Iterator

- The Java Iterator only **preserves the iteration in the forward direction**. In simple words, the Java Iterator is a unidirectional Iterator.
- The **replacement and extension of a new component** are not approved by the Java Iterator.
- In CRUD Operations, the Java Iterator does not hold the various operations like CREATE and UPDATE.
- In comparison with the Spliterator, **Java Iterator does not support traversing elements in the parallel pattern** which implies that Java Iterator supports only Sequential iteration.
- In comparison with the Spliterator, Java Iterator **does not support more reliable execution to traverse the bulk volume of data.**

Iterator Position

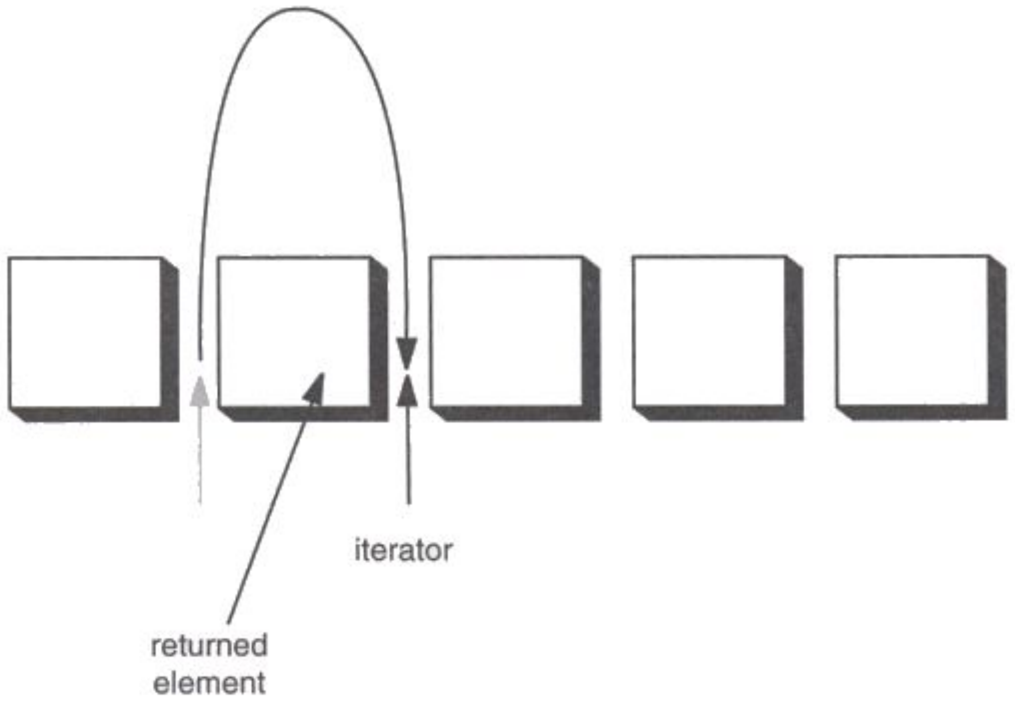
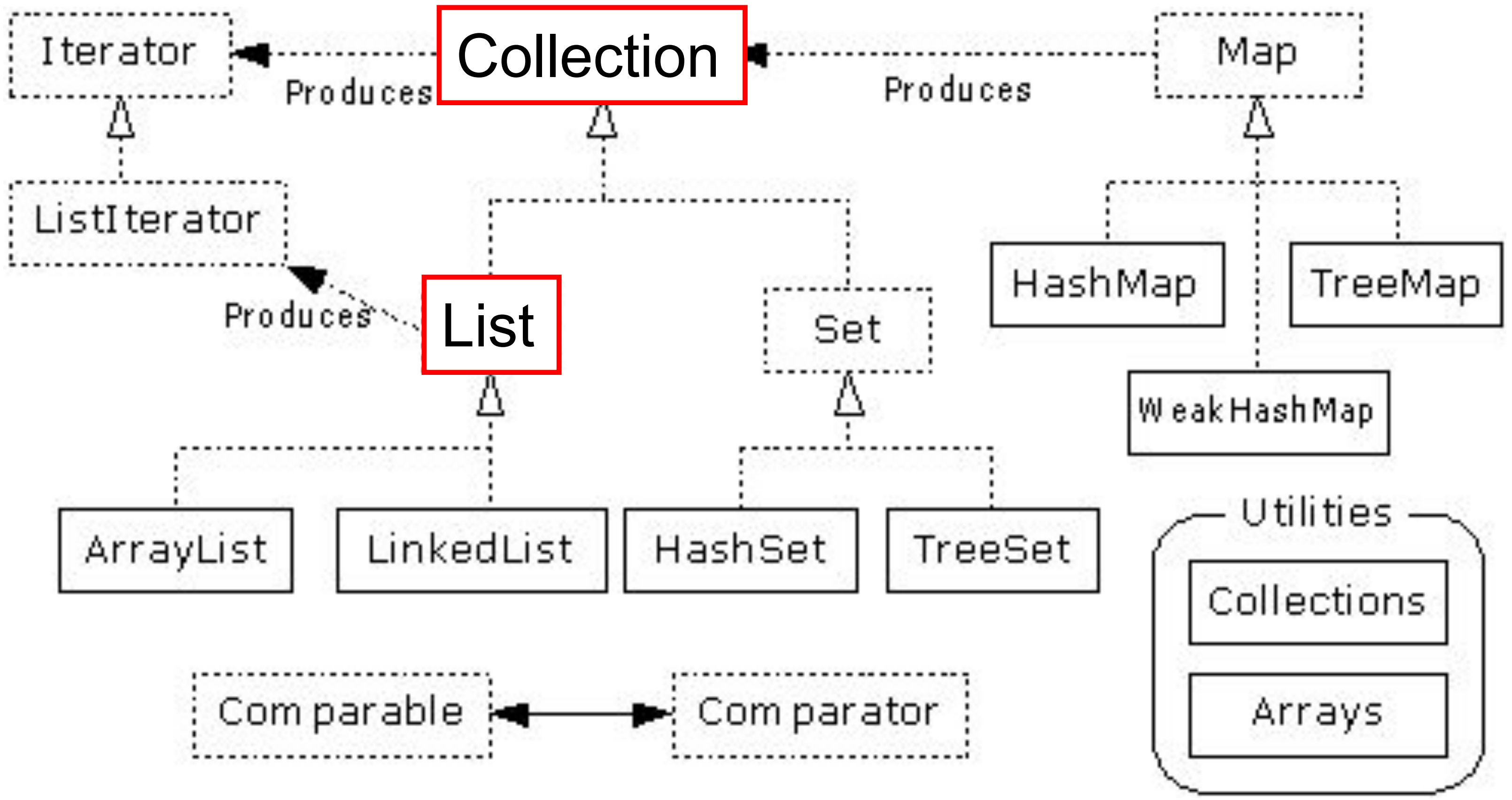


Figure 2–3: Advancing an iterator

List Interface



List Interface

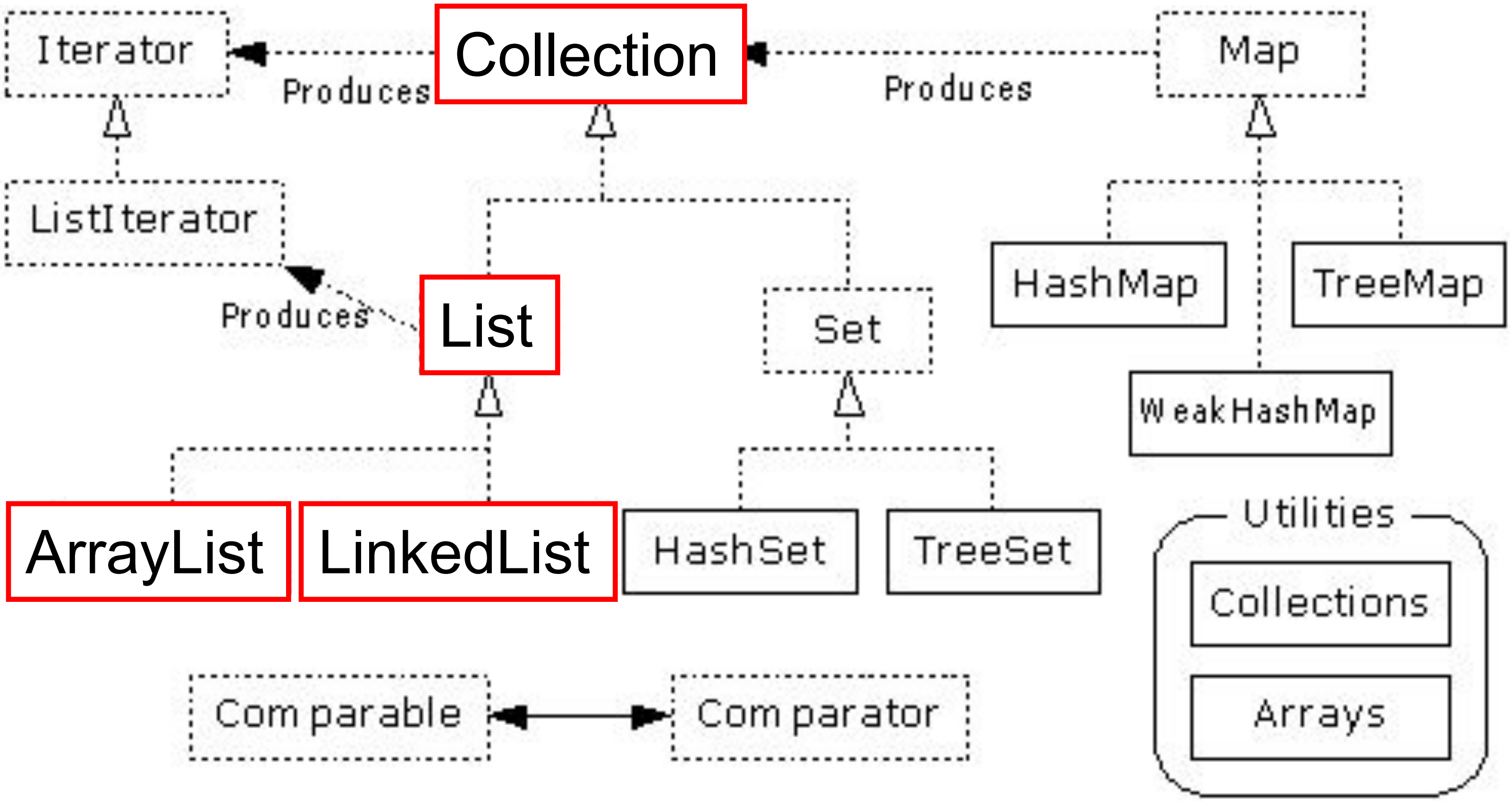
- The List interface adds the notion of order to a collection
- The user of a list has control over where an element is added in the collection
- Lists typically allow duplicate elements
- Provides a ListIterator to step through the elements in the list.

ListIterator Interface

- Extends the Iterator interface
- Defines three fundamental methods
  - void add(Object o) - before current position
  - boolean hasPrevious()
  - Object previous()
- The addition of these three methods defines the basic behavior of an ordered list
- A ListIterator knows position within list

List	**Set**
1. The List is an indexed sequence.	1. The Set is an non-indexed sequence.
2. List allows duplicate elements	2. Set doesn’t allow duplicate elements.
3. Elements by their position can be accessed.	3. Position access to elements is not allowed.
4. Multiple null elements can be stored.	4. Null element can store only once.
5. List implementations are ArrayList, LinkedList, Vector, Stack	5. Set implementations are HashSet, LinkedHashSet.

ArrayList and LinkedList Context



List Implementations

- ArrayList
  - low cost random access
  - high cost insert and delete
  - array that resizes if need be
- LinkedList
  - sequential access
  - low cost insert and delete
  - high cost random access

ArrayList overview

- Constant time positional access (it’s an array)
- One tuning parameter, the initial capacity

ArrayList methods

- The indexed get and set methods of the List interface are appropriate to use since ArrayLists are backed by an array
  - Object get(int index)
  - Object set(int index, Object element)
- Indexed add and remove are provided, but can be costly if used frequently
  - void add(int index, Object element)
  - Object remove(int index)
- May want to resize in one shot if adding many elements
  - void ensureCapacity(int minCapacity)

Property	Array	ArrayList
Resizable	The array is not resizable. At instantiation, you must provide its maximum size. Overfilling leads to an exception.	ArrayList is dynamically resizable. It grows as required, even beyond its initial capacity, offering flexibility in the number of elements it can store.
Performance	Arrays generally offer better performance than ArrayLists because they are of fixed size and don't require internal operations to store elements.	ArrayList can be slower, especially when increasing in size. This is because, when its capacity is exceeded, a new array is created internally, and existing elements are copied over, reducing its performance.
Size Method	Use Array.length to determine the length of an Array. For example: int arr[]=new int[4]; System.out.print(arr.length); will print 4.	The size() method determines the length of an ArrayList. Example: ArrayList<Integer> A = new ArrayList<Integer>(); A.add(1); A.add(2); System.out.println(A.size()); will print 2.
Dimension	Arrays support multiple dimensions (e.g., 2D, 3D arrays). For instance, int arr[][]=new int[4][3]; creates a 2D array with 4 rows and 3 columns.	ArrayList supports only single dimensions, and isn't suitable for tasks like matrix creation.
Data Type	Arrays support both primitive types (like int, char) and object types.	ArrayList supports only object types due to its use of generics. For instance, instead of int, you'd use Integer.
Null Values	Arrays can hold null values. However, for primitive arrays, there's no null (e.g., an int array defaults to 0 for each position unless otherwise assigned).	ArrayLists can contain and identify null elements.
Synchronization	Arrays aren't synchronized, so they're not inherently thread-safe.	ArrayList is also not synchronized by default. However, synchronization can be added externally using Collections.synchronizedList().

LinkedList overview

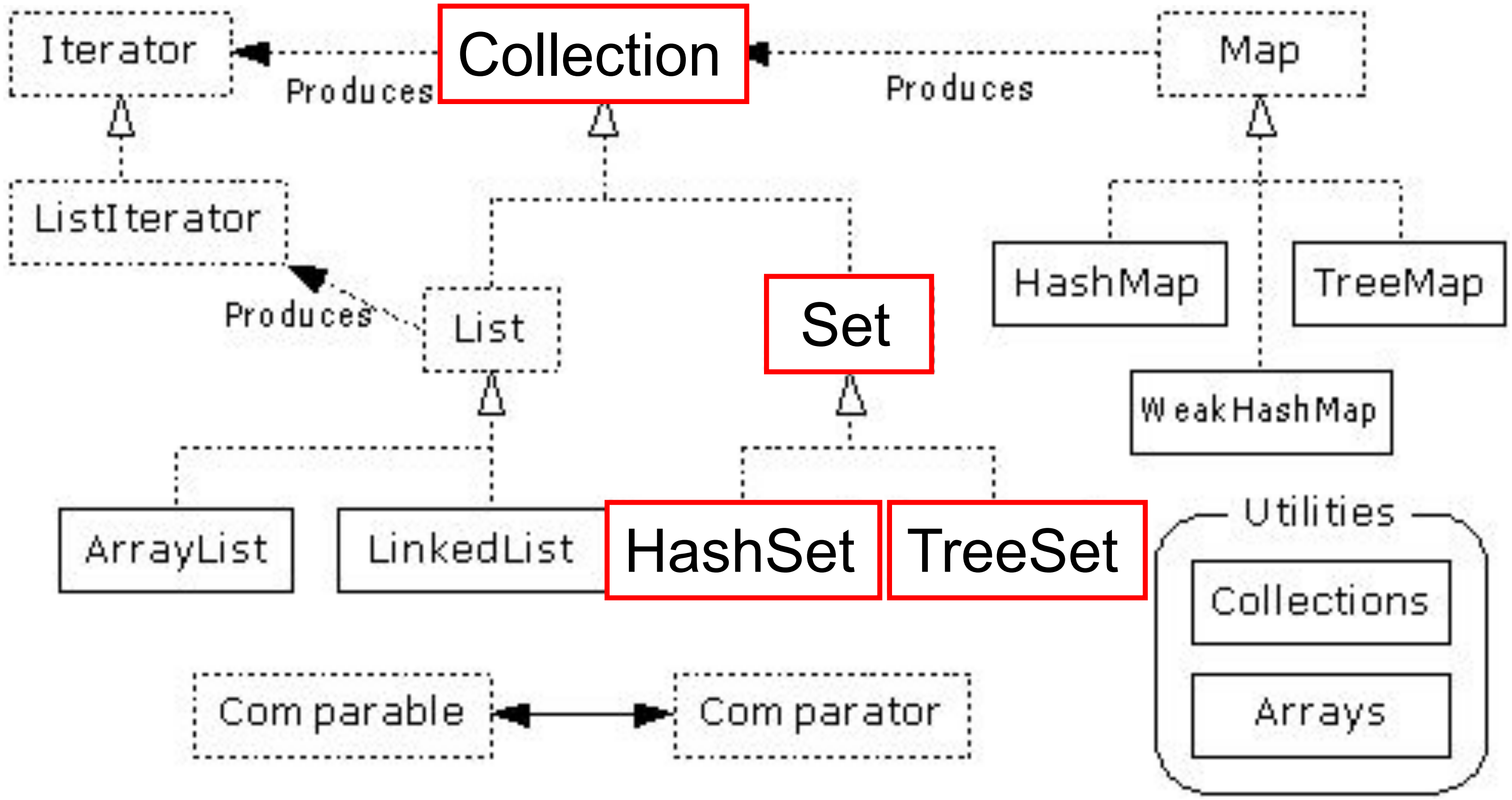
- Stores each element in a node
- Each node stores a link to the next and previous nodes
- Insertion and removal are inexpensive
  - just update the links in the surrounding nodes
- Linear traversal is inexpensive
- Random access is expensive
  - Start from beginning or end and traverse each node while counting

LinkedList methods

- The list is sequential, so access it that way
  - ListIterator listIterator()
- ListIterator knows about position
  - use add() from ListIterator to add at a position
  - use remove() from ListIterator to remove at a position
- LinkedList knows a few things too
  - void addFirst(Object o), void addLast(Object o)
  - Object getFirst(), Object getLast()
  - Object removeFirst(), Object removeLast()

ArrayList	LinkedList
This class uses a dynamic array to store the elements in it. With the introduction of generics, this class supports the storage of all types of objects.	This class uses a doubly linked list to store the elements in it. Similar to the ArrayList, this class also supports the storage of all types of objects.
Manipulating ArrayList takes more time due to the internal implementation. Whenever we remove an element, internally, the array is traversed and the memory bits are shifted.	Manipulating LinkedList takes less time compared to ArrayList because, in a doubly-linked list, there is no concept of shifting the memory bits. The list is traversed and the reference link is changed.
Inefficient memory utilization.	Good memory utilization.
Insertion operation is slow.	Insertion operation is fast.
This class works better when the application demands storing the data and accessing it.	This class works better when the application demands manipulation of the stored data.
Data access and storage is very efficient as it stores the elements according to the indexes.	Data access and storage is slow in LinkedList.
Deletion operation is not very efficient.	Deletion operation is very efficient.
It is used to store only similar types of data.	It is used to store any types of data.
Less memory is used.	More memory is used.
This is known as static memory allocation.	This is known as dynamic memory allocation

HashSet and TreeSet Context



HashSet

- Find and add elements very quickly
  - uses hashing implementation in HashMap
- Hashing uses an array of linked lists
  - The hashCode() is used to index into the array
  - Then equals() is used to determine if element is in the (short) list of elements at that index
- No order imposed on elements
- The hashCode() method and the equals() method must be compatible
  - if two objects are equal, they must have the same hashCode() value

S. No.	Hash Set	Tree Set
1	The Hash set is executed with the help of a HashTable.	The tree set is executed with the help of a tree structure.
2	It does not authorise a heterogeneous object.	It authorises a heterogeneous object.
3	It permits a null object.	It does not permit the null object.
4	To compare two objects, we use the equals method.	To compare two objects, we use the compare method.
5	It does not support any order	TreeSet supports an object in sorted order.

Java LinkedHashSet class

Java LinkedHashSet class is a Hashtable and Linked list implementation of the Set interface. It inherits the HashSet class and implements the Set interface.

The important points about the Java LinkedHashSet class are:

- Java LinkedHashSet class contains unique elements only like HashSet.
- Java LinkedHashSet class provides all optional set operations and permits null elements.
- Java LinkedHashSet class is non-synchronized.
- Java LinkedHashSet class maintains insertion order.

Advantages of LinkedHashSet

- It maintains insertion order.
- It allows quick insertion, deletion, and lookup of elements.
- It is useful for caching applications where insertion order is important.

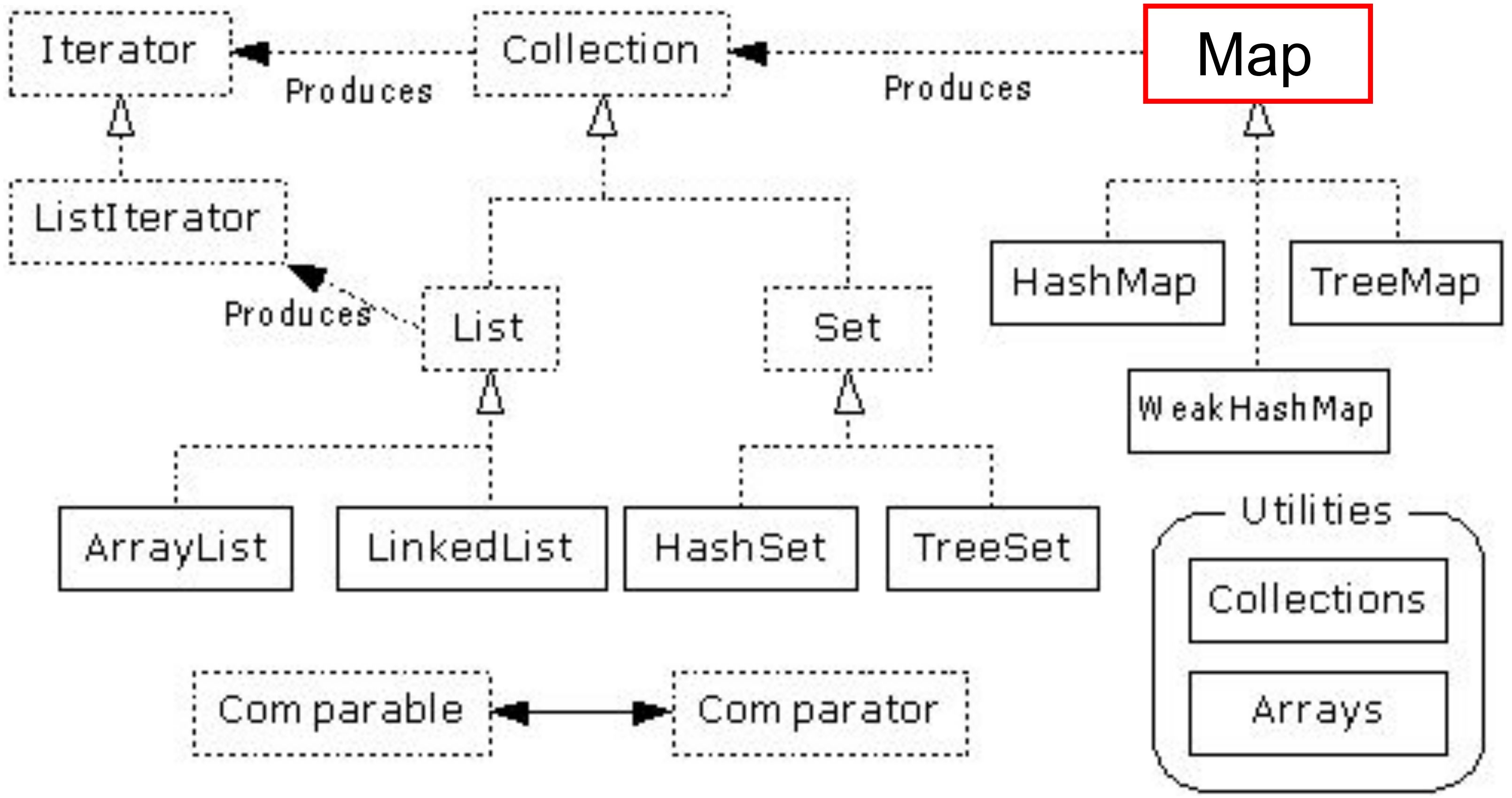
Disadvantages of LinkedHashSet

- It takes higher memory as compared to HashSet due to the linked list for maintaining insertion order.
- This is slightly slower operations compared to HashSet because of the linked structure.

TreeSet

- Elements can be inserted in any order
- The TreeSet stores them in order
    - Red-Black Trees out of Cormen-Leiserson-Rivest
- An iterator always presents them in order
- Default order is defined by natural order
    - objects implement the Comparable interface
    - TreeSet uses compareTo(Object o) to sort
- Can use a different Comparator
    - provide Comparator to the TreeSet constructor

Map Interface Context



Map Interface

Stores key/value pairs

Maps from the key to the value

Keys are unique

a single key only appears once in the Map

a key can map to only one value

Values do not have to be unique

A Map cannot contain duplicate keys and each key can map to at most one value. Some implementations allow null key and null values like the HashMap and LinkedHashMap, but some do not like the TreeMap.

The order of a map depends on the specific implementations. For example, TreeMap and LinkedHashMap have predictable orders, while HashMap does not.

There are two interfaces for implementing Map in Java. They are Map and SortedMap, and three classes: HashMap, TreeMap, and LinkedHashMap.

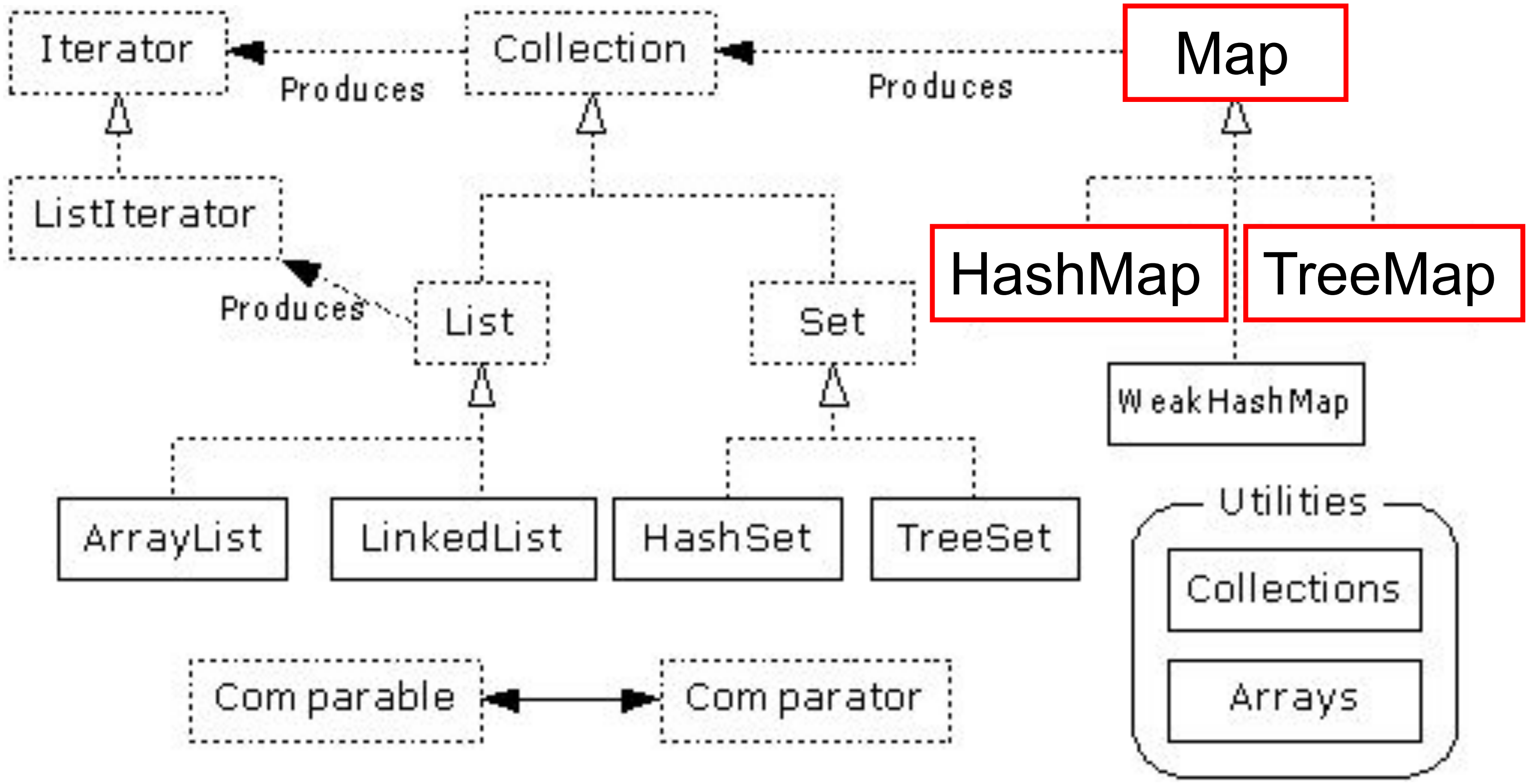
Map methods

- Object put(Object key, Object value)
- Object get(Object key)
- Object remove(Object key)
- boolean containsKey(Object key)
- boolean containsValue(Object value)
- int size()
- boolean isEmpty()

Map views

- A means of iterating over the keys and values in a Map
- Set keySet()
  - returns the Set of keys contained in the Map
- Collection values()
  - returns the Collection of values contained in the Map. This Collection is not a Set, as multiple keys can map to the same value.
- Set entrySet()
  - returns the Set of key-value pairs contained in the Map. The Map interface provides a small nested interface called Map.Entry that is the type of the elements in this Set.

HashMap and TreeMap Context



HashMap

In Java, HashMap is part of the Java Collections Framework and is found in the java.util package. It provides the basic implementation of the Map interface in Java. HashMap stores data in (key, value) pairs. Each key is associated with a value, and you can access the value by using the corresponding key.

Internally uses Hashing (similar to Hashtable in Java).

Not synchronized (unlike Hashtable in Java) and hence faster for most of the cases.

Allows to store the null keys as well, but there should be only one null key object, and there can be any number of null values.

Duplicate keys are not allowed in HashMap, if you try to insert the duplicate key, it will replace the existing value of the corresponding key.

HashMap uses keys in the same way as an Array uses an index.

HashMap allows for efficient key-based retrieval, insertion, and removal with an average O(1) time complexity.

TreeMap in Java

TreeMap is a part of the Java Collection Framework. It implements the Map and NavigableMap interface and extends the AbstarctMap class. It stores key-value pairs in a sorted order based on the natural ordering of keys or a custom Comparator. It uses a Red-Black Tree for efficient operations (add, remove, retrieve) with a time complexity of O(log n).

The keys in a TreeMap are always sorted.

Most operations, such as get, put, and remove have a time complexity of O(logn).

TreeMap does not allow null as a key, it allows null as a value. Attempting to insert a null key will result in NullPointerException.

TreeMap is not Synchronized. For thread-safe operations, we need to use Collections.synchronized map.

Entry pairs returned by the methods in this class and their views represent snapshots of mappings at the time they were produced. They do not support the Entry.setValue method.

HashMap	**TreeMap**
HashMap does not guarantee a sorted order of elements.	TreeMap guarantees a sorted order of elements based on the natural ordering of keys or a custom comparator.
HashMap is generally faster for insertion and retrieval operations, especially for large datasets.	TreeMap offers efficient range queries and operations like finding the closest key.
HashMap uses hashing for storing key-value pairs, providing constant-time performance for basic operations (e.g., get, put, remove) on average.	TreeMap uses a red-black tree internally, resulting in logarithmic-time complexity for most operations, making it slightly slower than HashMap for basic operations.
Iterating over elements in a HashMap does not guarantee any specific order.	Iterating over elements in a TreeMap results in sorted order based on the keys.
HashMap allows null values, and multiple null values can be associated with different keys.	TreeMap allows null values but not null keys.
HashMap is not synchronized, making it not thread-safe.	TreeMap is not synchronized, making it not thread-safe.
HashMap is part of the java.util package.	TreeMap is part of the java.util package and implements the NavigableMap interface

queue interface

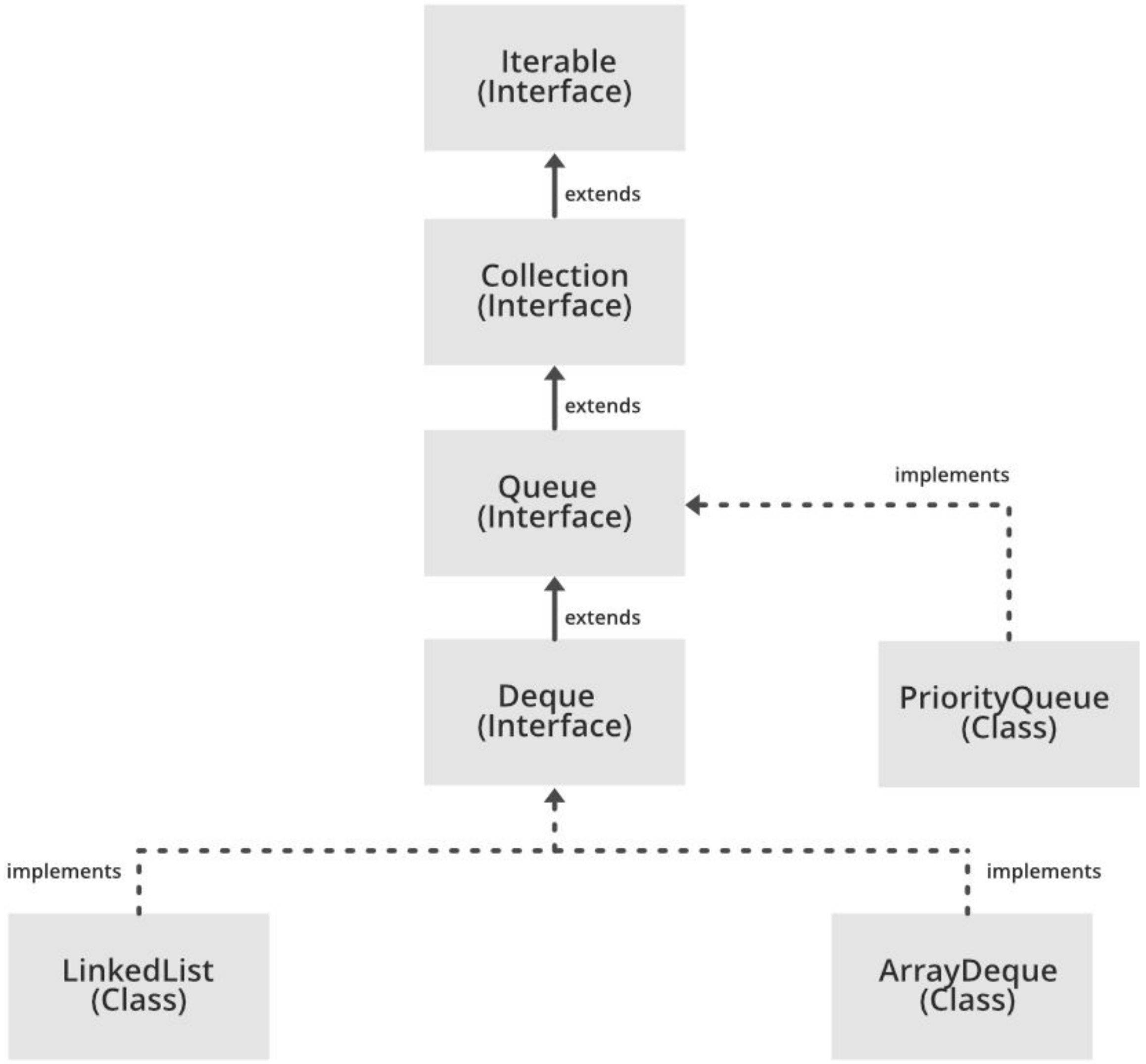
The Queue Interface is present in java.util package and extends the Collection interface. It stores and processes the data in FIFO(First In First Out) order. It is an ordered list of objects limited to inserting elements at the end of the list and deleting elements from the start of the list.

No Null Elements: Most implementations like PriorityQueue do not allow null elements.

Implementation Classes: LinkedList , PriorityQueue, ArrayDeque, ConcurrentLinkedQueue (for thread-safe operations).

Use Cases: Commonly used for Task scheduling, Message passing, and Buffer management in applications.

Iteration: Supports iterating through elements. The order of iteration depends on the implementation.


Iterable (Interface)
extends
Collection (Interface)
extends
Queue (Interface)
implements
extends
Deque (Interface)
PriorityQueue (Class)
implements
implements
LinkedList (Class)
ArrayDeque (Class)

```
import java.util.LinkedList;
import java.util.Queue;

public class QueueExample {
    public static void main(String[] args) {
        Queue<String> queue = new LinkedList<>();

        // add elements to the queue
        queue.add("apple");
        queue.add("banana");
        queue.add("cherry");

        // print the queue
        System.out.println("Queue: " + queue);

        // remove the element at the front of the queue
        String front = queue.remove();
        System.out.println("Removed element: " + front);

        // print the updated queue
        System.out.println("Queue after removal: " + queue);

        // add another element to the queue
        queue.add("date");

        // peek at the element at the front of the queue
        String peeked = queue.peek();
        System.out.println("Peeked element: " + peeked);

        // print the updated queue
        System.out.println("Queue after peek: " + queue);
    }
}
```

Output

```
Queue: [apple, banana, cherry]
Removed element: apple
Queue after removal: [banana,
cherry]
Peeked element: banana
  Queue after peek: [banana,
  cherry, date]
```

```
// Java program to demonstrate a Queue
import java.util.LinkedList;
import java.util.Queue;

public class QueueExample {
    public static void main(String[] args)
    {
        Queue<Integer> q
            = new LinkedList<>();
        // Adds elements {0, 1, 2, 3, 4} to
        // the queue
        for (int i = 0; i < 5; i++)
            q.add(i);
        // Display contents of the queue.
        System.out.println("Elements of queue "  + q);
        // To remove the head of queue.
        int removedele = q.remove();
        System.out.println("removed element-"  + removedele);
        System.out.println(q);
        // To view the head of queue
        int head = q.peek();
        System.out.println("head of queue-"    + head);
        // Rest all methods of collection
        // interface like size and contains
        // can be used with this
        // implementation.
        int size = q.size();
        System.out.println("Size of queue-"    + size);
    }
}
```

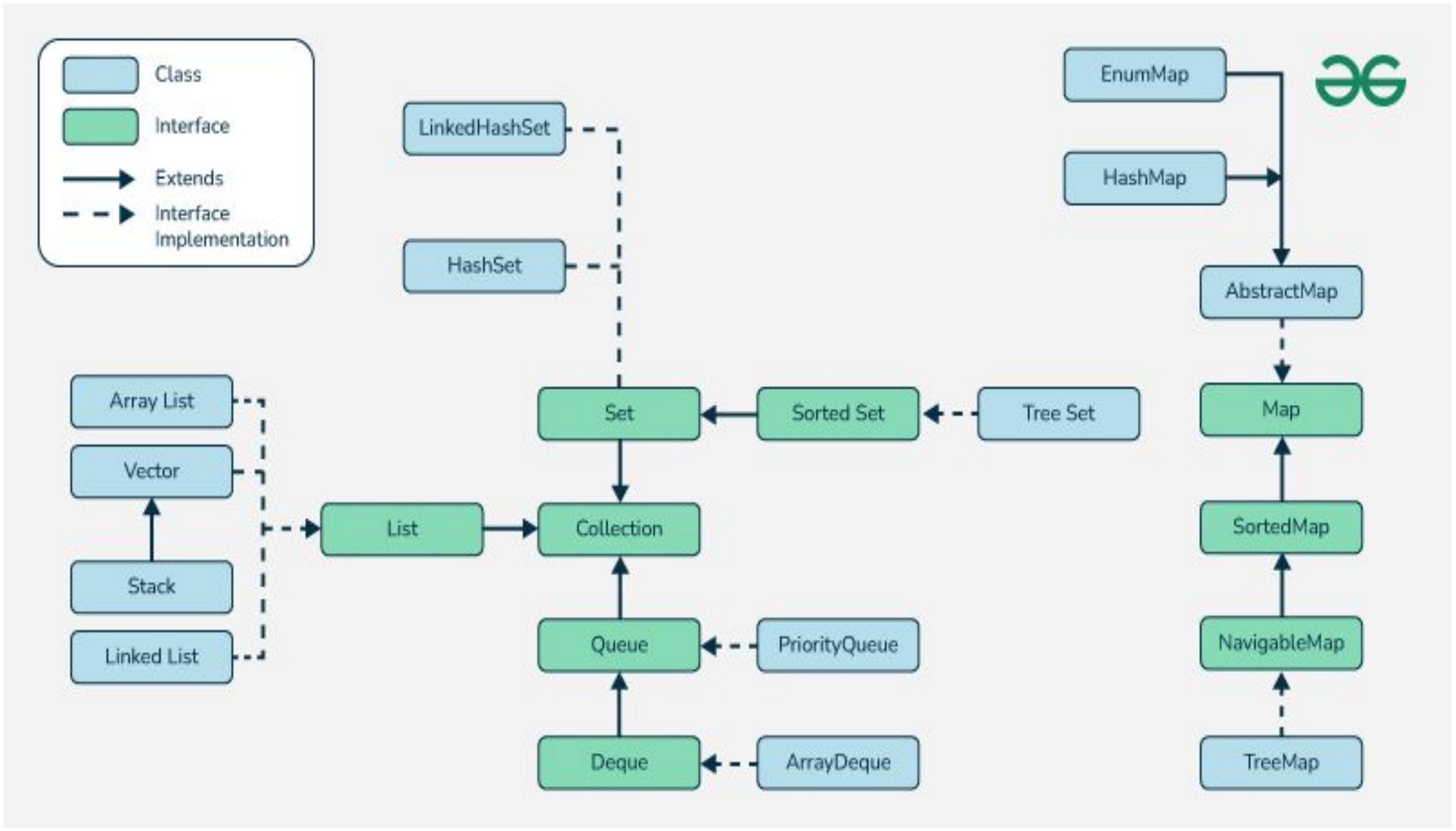
Output

```
Elements of queue [0, 1, 2, 3, 4]
removed element-0
[1, 2, 3, 4]
head of queue-1
  Size of queue-4
```

hashset

HashSet in Java implements the Set interface of Collections Framework. It is used to store the unique elements and it doesn’t maintain any specific order of elements.

- Can store the Null values.
- Uses HashMap (implementation of hash table data structure) internally.
- Also implements Serializable and Cloneable interfaces.
- HashSet is not thread-safe. So to make it thread-safe, synchronization needed externally.


Class
Interface
Extends
Interface Implementation
LinkedHashSet
HashSet
EnumMap
HashMap
AbstractMap
Map
Array List
Vector
Stack
Linked List
List
Collection
Set
Sorted Set
Tree Set
Queue
PriorityQueue
Deque
ArrayDeque
SortedMap
NavigableMap
TreeMap

```
// Java program to Adding Elements to HashSet
import java.util.*;

class GFG
{
   public static void main(String[] args)
   {
      // Creating an empty HashSet of string
entities
      HashSet<String> hs = new HashSet<String>();
      // Adding elements using add() method
      hs.add("Geek");
      hs.add("For");
      hs.add("Geeks");
      // Printing all string entries inside the Set
      System.out.println("HashSet : " + hs);
 // Removing the element B
      hs.remove("B");
      // Printing the updated HashSet elements
      System.out.println("HashSet after removing
element : " + hs);
      // Returns false if the element is not present
    System.out.println("B exists in Set : " +
hs.remove("B"));
```

```
// Using iterator() method to iterate Over
the HashSet
System.out.print("Using iterator : ");
Iterator<String> iterator = hs.iterator();
 // Traversing HashSet
while (iterator.hasNext())
      System.out.print(iterator.next() + ",
");
      System.out.println();
 // Using enhanced for loop to iterate
Over the HashSet
System.out.print("Using enhanced for
loop : ");
   for (String element : hs)
      System.out.print(element + " , ");
   }
}
```

Queue Interface In Java

The Queue Interface is present in java.util package and extends the Collection interface. It stores and processes the data in FIFO(First In First Out) order. It is an ordered list of objects limited to inserting elements at the end of the list and deleting elements from the start of the list.

No Null Elements: Most implementations like PriorityQueue do not allow null elements.

Implementation Classes: LinkedList , PriorityQueue, ArrayDeque, ConcurrentLinkedQueue (for thread-safe operations).

Use Cases: Commonly used for Task scheduling, Message passing, and Buffer management in applications.

Iteration: Supports iterating through elements. The order of iteration depends on the implementation.

Enums

An enum is a special "class" that represents a group of constants (unchangeable variables, like final variables).
To create an enum, use the enum keyword (instead of class or interface), and separate the constants with a comma. Note that they should be in uppercase letters:
Difference between Enums and Classes
An enum can, just like a class, have attributes and methods. The only difference is that enum constants are public, static and final (unchangeable - cannot be overridden).
An enum cannot be used to create objects, and it cannot extend other classes (but it can implement interfaces).
Why And When To Use Enums?
Use enums when you have values that you know aren't going to change, like month days, days, colors, deck of cards, etc.

}

```
public enum Type
{ AMPHIBIAN, MAMMAL, REPTILE, BIRD }

public enum Animal {
 ELEPHANT(Type.MAMMAL),
GIRAFFE(Type.MAMMAL),
TURTLE(Type.REPTILE),
SNAKE(Type.REPTILE),
FROG(Type.AMPHIBIAN);
private final Type type;
private Animal(final Type type) { this.type = type; }
public boolean isMammal() { return this.type == Type.MAMMAL; }
 public boolean isAmphibian() { return this.type == Type.AMPHIBIAN; }
 public boolean isReptile() { return this.type == Type.REPTILE; } // etc... }
```

```java
public class Main {
  enum Level {
    LOW,
    MEDIUM,
    HIGH
  }

  public static void main(String[] args) {
    Level myVar = Level.MEDIUM;
    System.out.println(myVar);
  }
```

1. Declaration outside the class

```
// A simple enum example where enum is declared outside any class (Note
enum keyword instead of // class keyword)
enum Color {
    RED,
    GREEN,
    BLUE;
}

public class Test {
    // Driver method
    public static void main(String[] args) {
        Color c1 = Color.RED;
        System.out.println(c1);
    }
}
```

Output

```
RED
```

2. Declaration inside a class

```
// enum declaration inside a class.
public class Test {
    enum Color {
        RED,
        GREEN,
        BLUE;
    }
    // Driver method
    public static void main(String[] args) {
        Color c1 = Color.RED;
        System.out.println(c1);
    }
}
```

Output

```
RED
```

1. Main Function Inside Enum

Enums can have a `main` function, allowing them to be invoked directly from the command line.

```
// main() inside enum class.

public enum Color {
    RED,
    GREEN,
    BLUE;

    // Driver method
    public static void main(String[] args) {
        Color c1 = Color.RED;
        System.out.println(c1);
    }
}
```

Output

```
RED
```

2. Loop through Enum

```
// to Print all the values inside the enum using for loop
import java.io.*;
// Enum Declared is global
enum Color {
    RED,
    GREEN,
    BLUE;
}

// Driver Class
class GFG {

    // Main Function
    public static void main(String[] args) {
        // Iterating over all the values in
        // enum using for loop
        for (Color var_1 : Color.values()) {
            System.out.println(var_1);
        }
    }
}
```

Output

```
RED
GREEN
BLUE
```

3. Enum in a Switch Statement

```
import java.io.*;
// Driver Class
class GFG {
       enum Color {
        RED,
        GREEN,
        BLUE,
        YELLOW;
    }

    // Main Function
    public static void main(String[] args) {
        Color var_1 = Color.YELLOW;

        // Switch case with Enum
        switch (var_1) {
        case RED:
            System.out.println("Red color observed");
            break;
        case GREEN:
            System.out.println("Green color observed");
            break;
        case BLUE:
            System.out.println("Blue color observed");
            break;
        default:
            System.out.println("Other color observed");
        }
    }
}
```

Output

```
Other color observed
```

Java Comparable Interface

The **Comparable interface in Java** is used to define the natural ordering of objects for a user-defined class.

It is part of **the java.lang** package and it provides a **compareTo()** method to compare instances of the class.

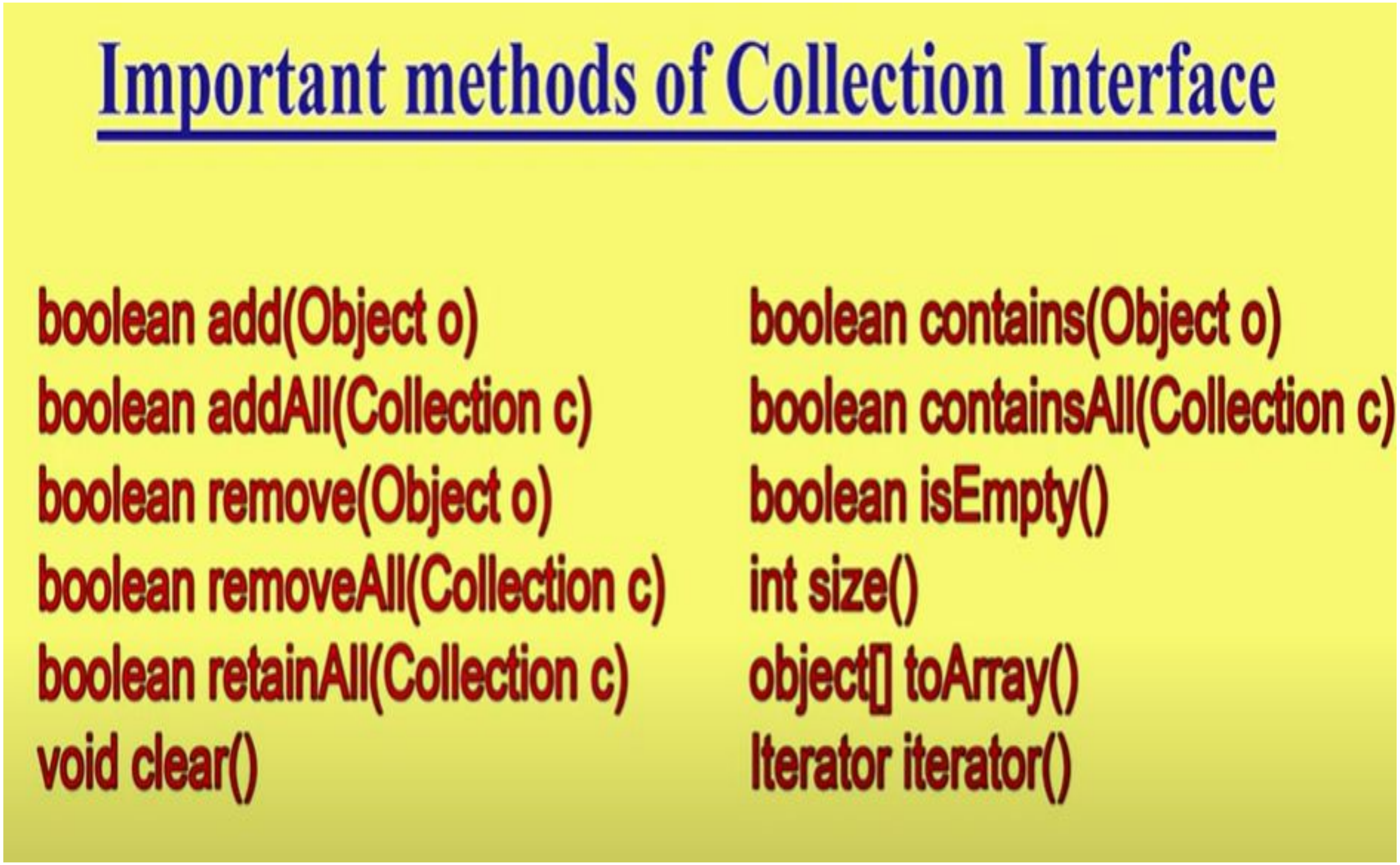
A class has to implement a Comparable interface to define its natural ordering.

Comparators

An object that implements the Comparator interface is called a comparator.
The Comparator interface allows you to create a class with a compare() method that compares two objects to decide which one should go first in a list.
The compare() method should return a number which is:

- Negative if the first object should go first in a list.
- Positive if the second object should go first in a list.
- Zero if the order does not matter.

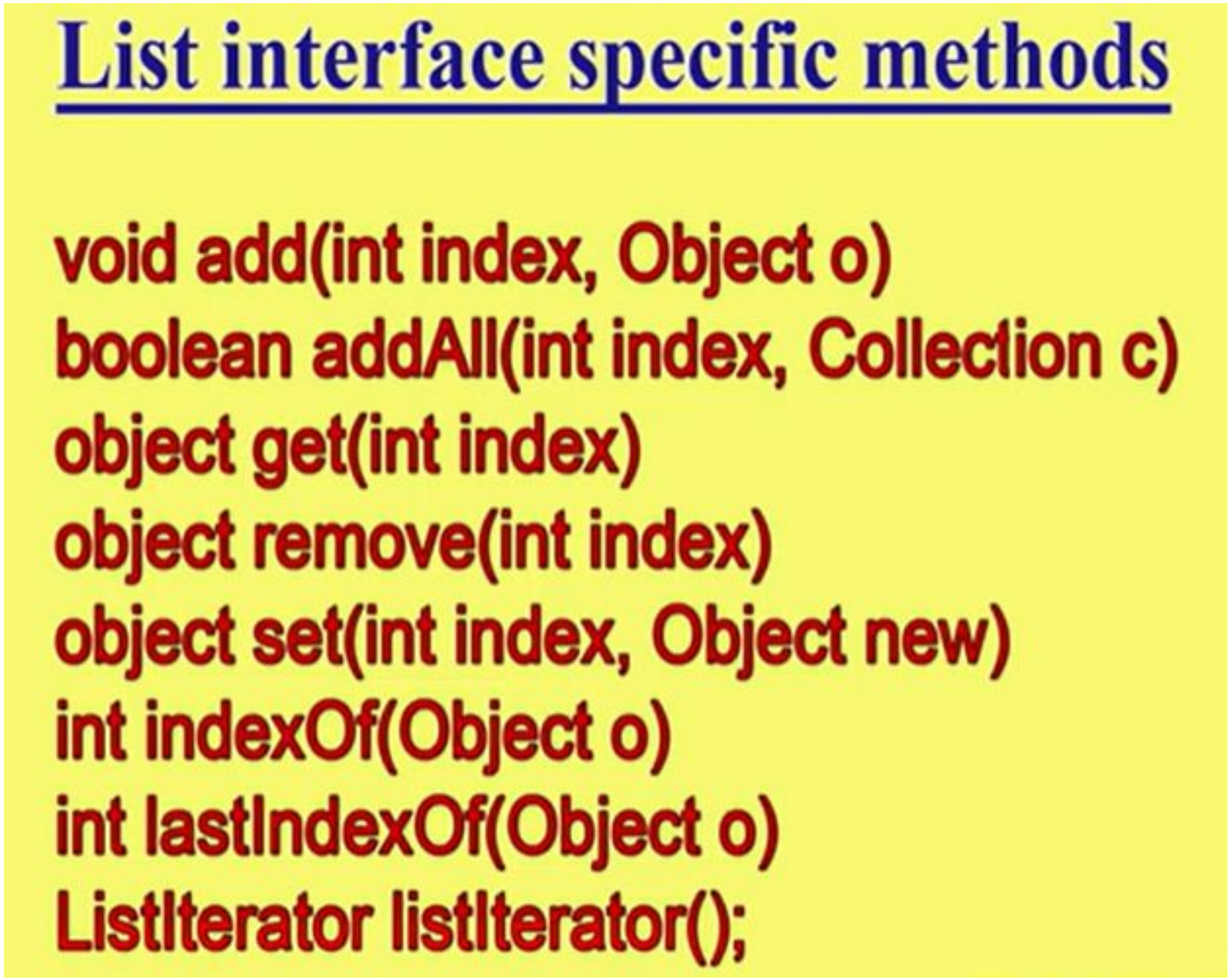
A class that implements the Comparator interface might look something like this:



Important methods of Collection Interface

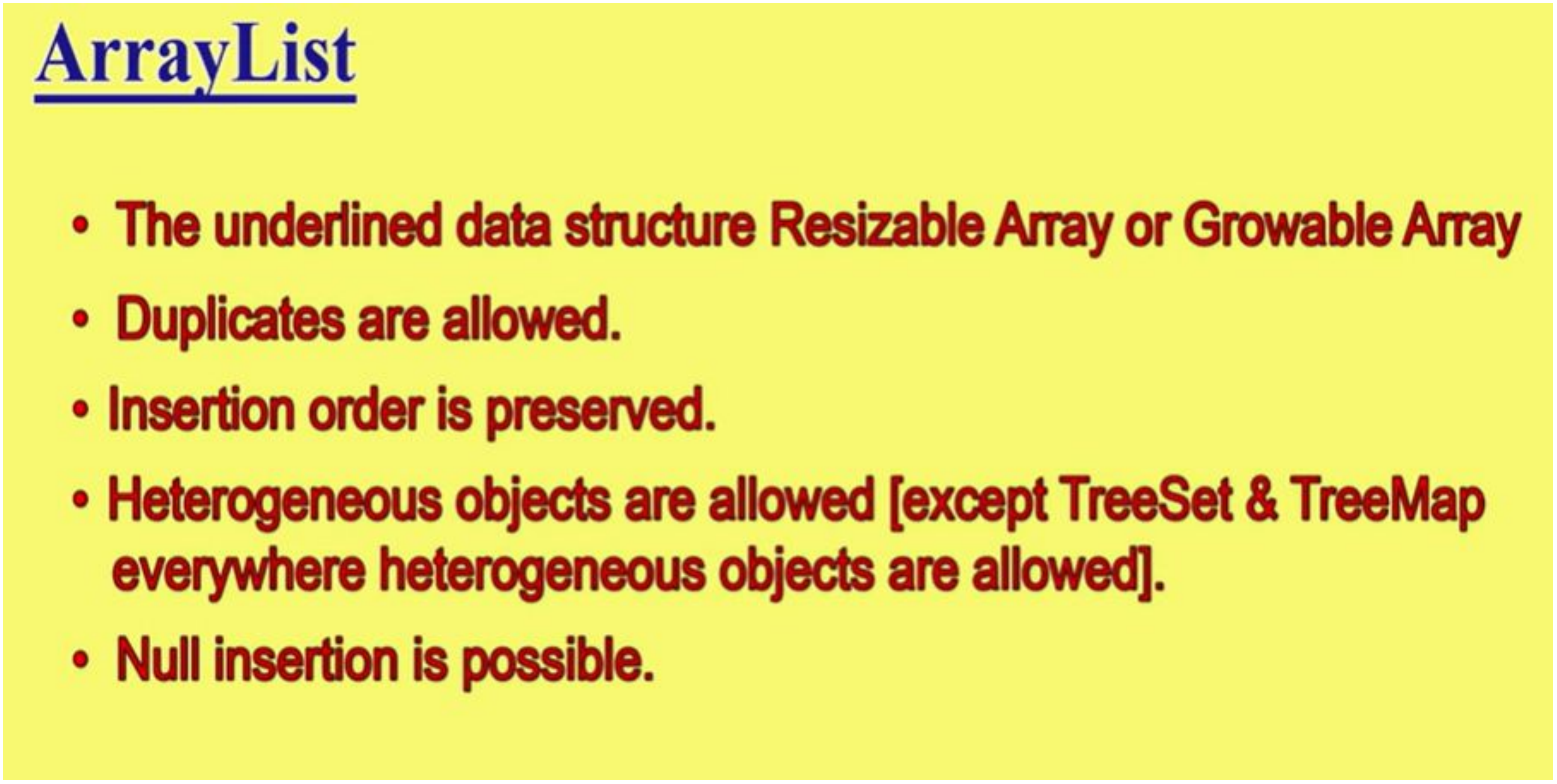
boolean add(Object o)
boolean addAll(Collection c)
boolean remove(Object o)
boolean removeAll(Collection c)
boolean retainAll(Collection c)
void clear()

boolean contains(Object o)
boolean containsAll(Collection c)
boolean isEmpty()
int size()
object[] toArray()
Iterator iterator()



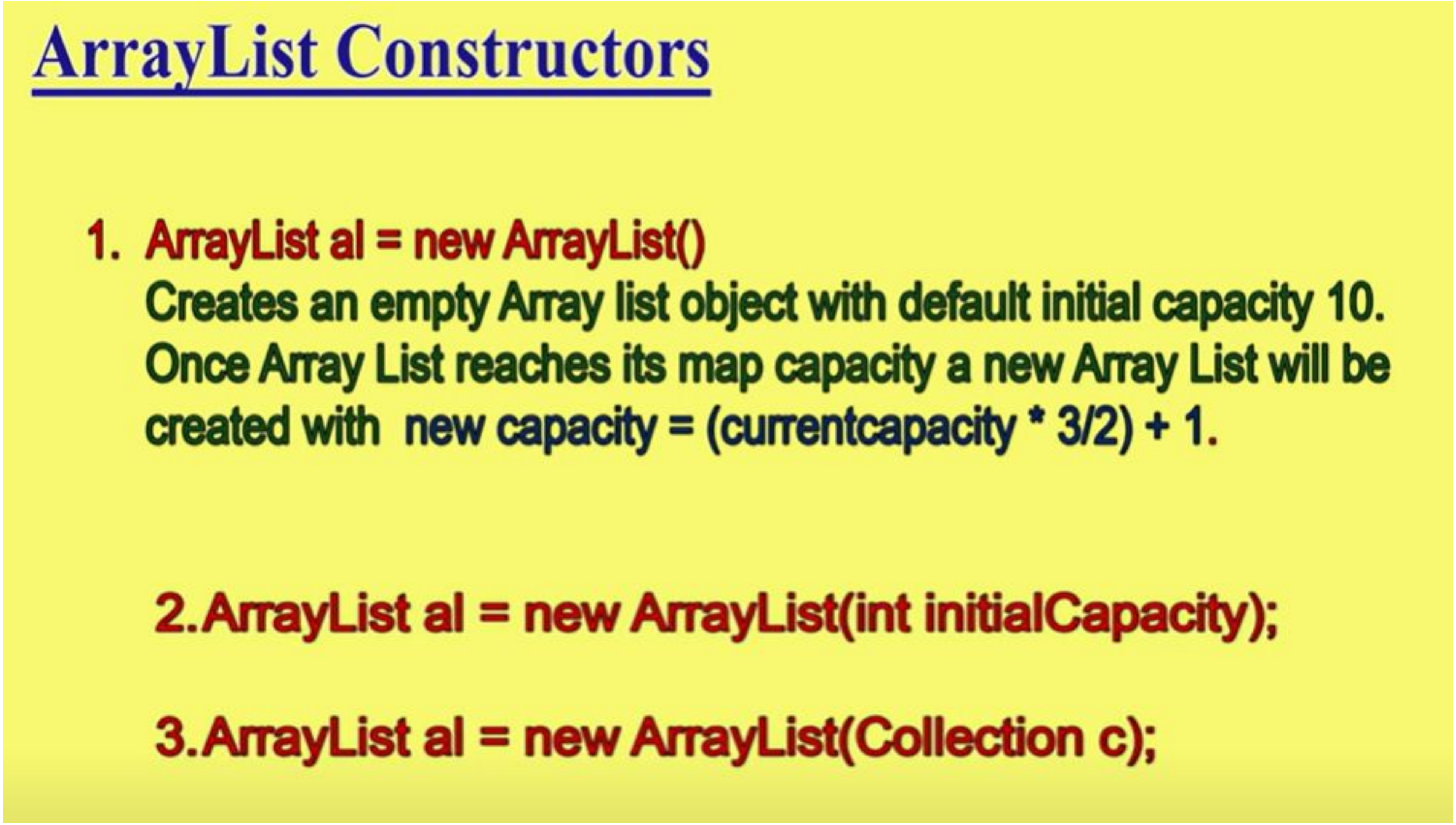
List interface specific methods

void add(int index, Object o)
boolean addAll(int index, Collection c)
object get(int index)
object remove(int index)
object set(int index, Object new)
int indexOf(Object o)
int lastIndexOf(Object o)
ListIterator listIterator();



ArrayList

- The underlined data structure Resizable Array or Growable Array
- Duplicates are allowed.
- Insertion order is preserved.
- Heterogeneous objects are allowed [except TreeSet & TreeMap everywhere heterogeneous objects are allowed].
- Null insertion is possible.



ArrayList Constructors

1. ArrayList al = new ArrayList()
   Creates an empty Array list object with default initial capacity 10.
   Once Array List reaches its map capacity a new Array List will be
   created with new capacity = (currentcapacity * 3/2) + 1.

2.ArrayList al = new ArrayList(int initialCapacity);

3.ArrayList al = new ArrayList(Collection c);