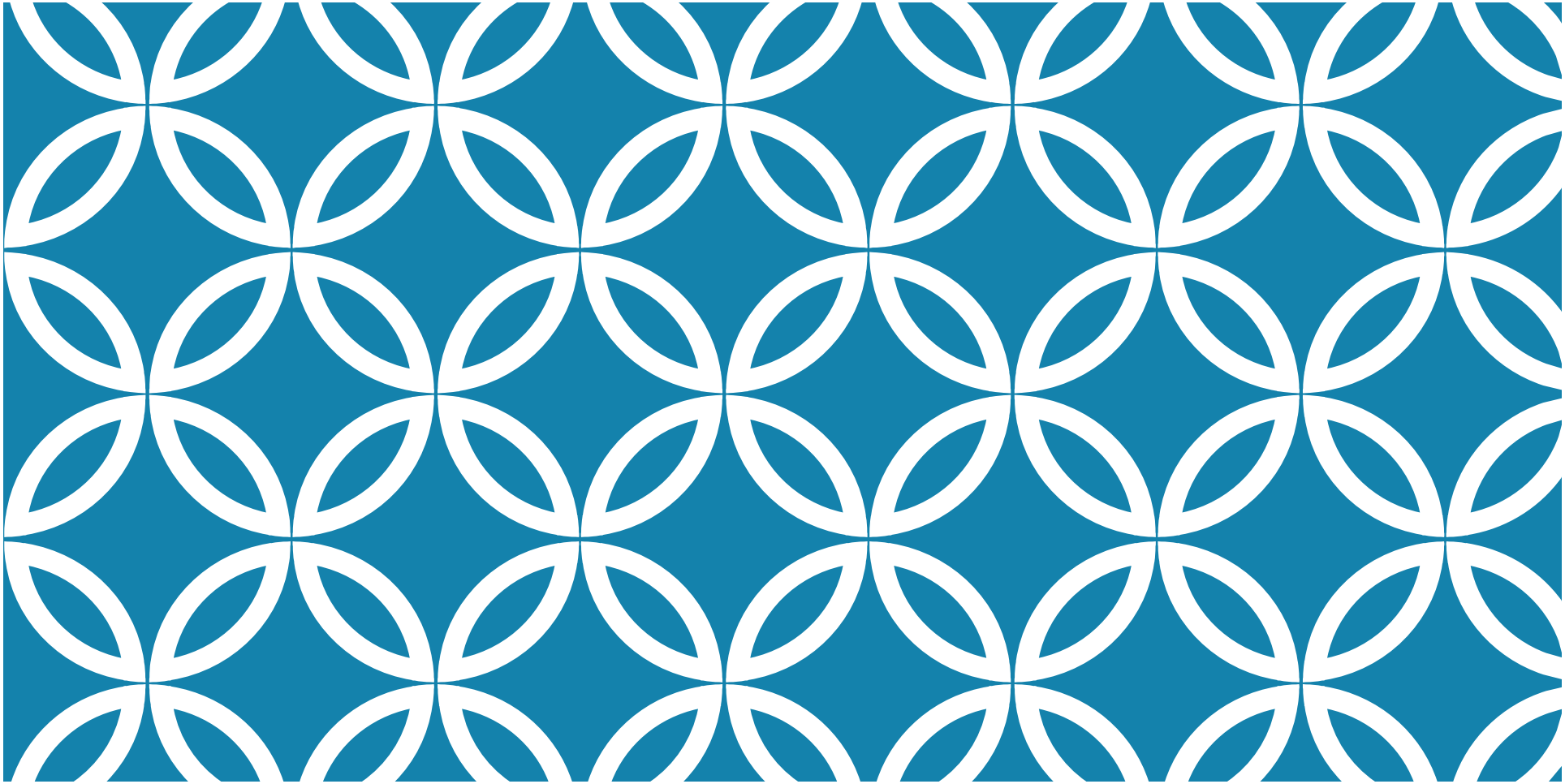
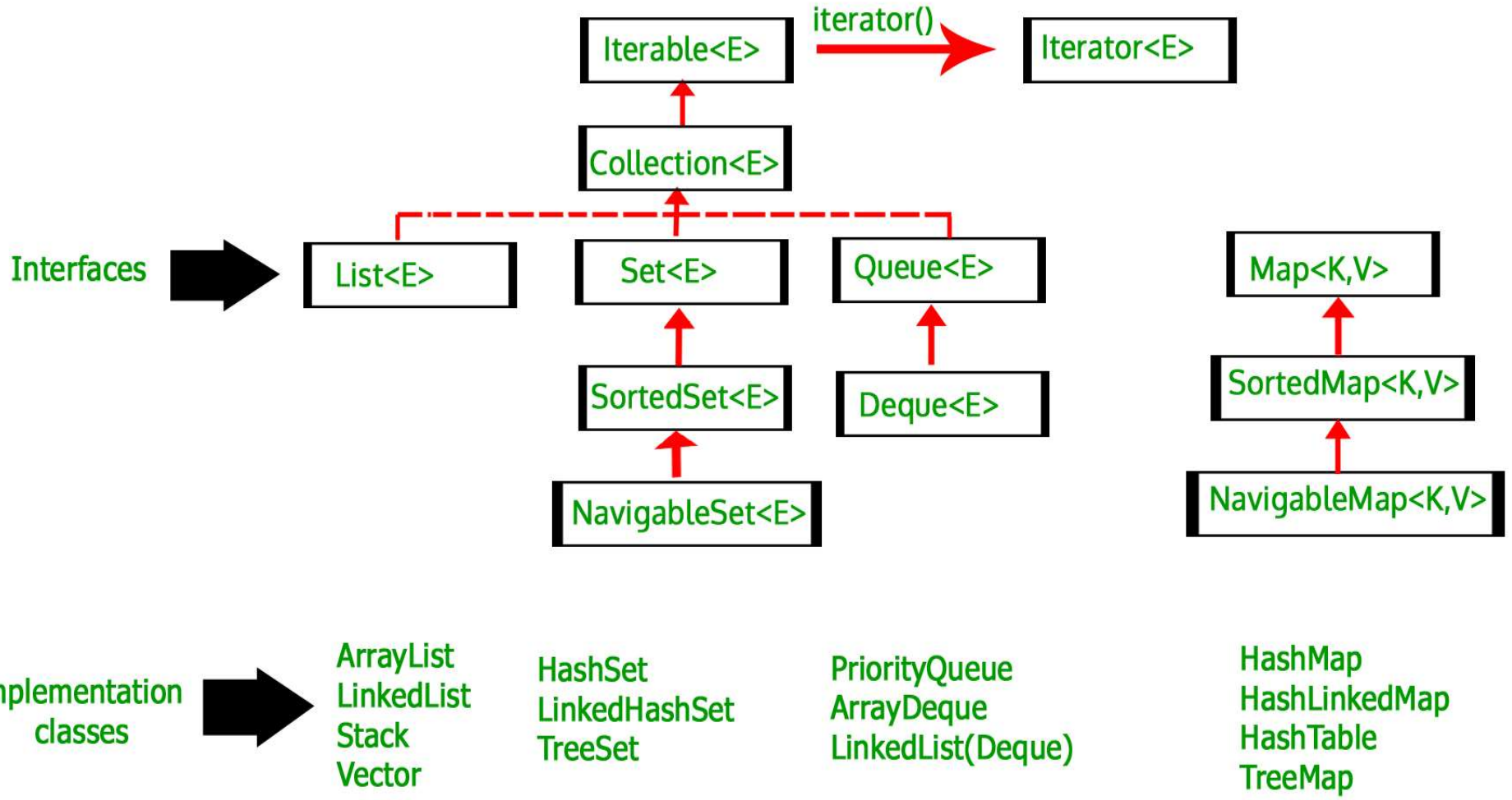


COLLECTIONS FRAMEWORK

Set, Trees & Queue

COMMONLY USED INTERFACES & COLLECTION CLASSES



SET INTERFACE

- It extends Collection and specifies the behavior of a collection that does not allow duplicate elements.
- add( ) method returns false if an attempt is made to add duplicate elements to a set.
- No specific ordering of elements
- No additional methods of its own.

ABSTRACTSET CLASS

- Extends AbstractCollection and implements most of the Set interface.
- Constructor - protected AbstractSet()
  - it doesn't allow to create an AbstractSet object.
- Methods :
- equals(Object o): Compares the specified object with this set for equality.
- removeAll(Collection c): Removes from this set all of its elements that are contained in the specified collection

INHERITED METHODS

Methods inherited from interface java.util.Set

- add, addAll, clear, contains, containsAll, isEmpty, iterator, remove, retainAll, size, toArray, toArray

TREESET

- TreeSet extends AbstractSet and implements the NavigableSet interface.
- It creates a collection that uses a tree for storage. Objects are stored in sorted, ascending order.
- Access and retrieval times are quite fast, which makes TreeSet an excellent choice when storing large amounts of sorted information that must be found quickly.

EXAMPLE

```
// Demonstrate TreeSet.
import java.util.*;

class TreeSetDemo {
  public static void main(String args[]) {
    // Create a tree set.
    TreeSet<String> ts = new TreeSet<String>();

    // Add elements to the tree set.
    ts.add("C");
    ts.add("A");
    ts.add("B");
    ts.add("E");
    ts.add("F");
    ts.add("D");
    System.out.println(ts);
  }
}
```

```
[A, B, C, D, E, F]
```

SUBSET()

- Methods defined by NavigableSet can be used to retrieve elements of a TreeSet.
- Eg: subSet( ) to obtain a subset of ts that contains the elements between C (inclusive) and F (exclusive)

```
System.out.println(ts.subSet("C", "F"));
```

The output from this statement is shown here:

```
[C, D, E]
```

EXAMPLE

```
import java.util.*;
public class Test {
	public static void main(String[] args) {

	// Creating object of AbstractSet<Integer>
	AbstractSet<Integer> abs_set = new TreeSet<Integer>();

	// Populating abs_set
	abs_set.add(1);
	abs_set.add(2);
	abs_set.add(3);
	abs_set.add(4);
	abs_set.add(5);
	// print abs_set
	System.out.println("AbstractSet: "+ abs_set);
	}
}
```

EXAMPLE-CONTD.,

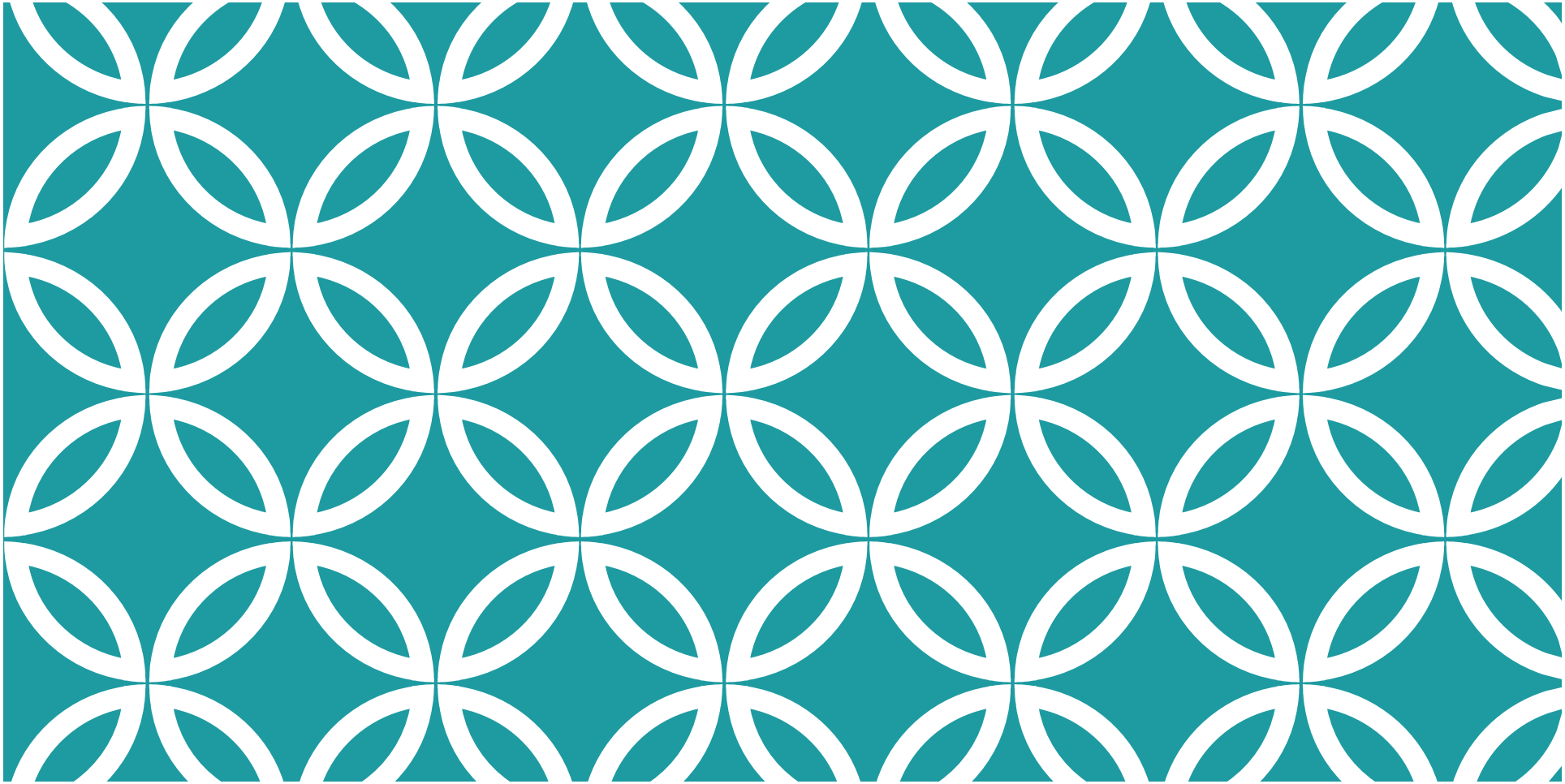
```
// Creating another object of ArrayList<Integer>
 Collection<Integer>  arrlist2 = new ArrayList<Integer>();
 arrlist2.add(1);
arrlist2.add(2);
 arrlist2.add(3);

// print arrlist2
System.out.println("Collection Elements to be removed : "  + arrlist2);
// Removing elements from AbstractSet  specified in arrlist2 using removeAll()
method
  abs_set.removeAll(arrlist2);

// print arrlist1
  System.out.println("AbstractSet after removeAll() operation : " + abs_set);
```

Output:

AbstractSet: [1, 2, 3, 4, 5]
Collection Elements to be removed : [1, 2, 3]
AbstractSet after removeAll() operation : [4, 5]



QUEUE

QUEUE INTERFACE

- The **Queue** interface extends **Collection** and declares the behavior of a queue, which is often a first-in, first-out list.
- The Queues in java.util package are Unbounded Queues
- The Queues in java.util.concurrent package are the Bounded Queues.
- All Queues except the Deques supports insertion and removal at the tail and head of the queue respectively.
- The Deques support element insertion and removal at both ends.
- LinkedList and PriorityQueue are commonly used implementations

METHODS IN QUEUE

add()	add elements at the tail of queue. Returns true on success and IllegalStateException (if capacity exceeded for bounded queues) on failure
offer()	add elements at the tail of queue. Returns true on success and false if no space available.
peek()	view the head of queue without removing it. It returns Null if the queue is empty.
element()	similar to peek(). It throws NoSuchElementException when the queue is empty.
remove()	removes and returns the head of the queue. throws NoSuchElementException when the queue is empty.
poll()	removes and returns the head of the queue. It returns null if the queue is empty.
size()	returns the no. of elements in the queue.

EXAMPLE

```
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 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);
  }
}
```

DEQUE INTERFACE

- The **Deque** interface extends **Queue** and declares the behavior of a double-ended queue.
- Double-ended queues can function as standard, first-in-first-out queues or as last-in-first-out stacks.

METHODS IN DEQUE

Queue Method	Equivalent Deque Method
add()	addLast()
offer()	offerLast()
remove()	removeFirst()
poll()	pollFirst()
element()	getFirst()
peek()	peekFirst()

Stack Method	Equivalent Deque Method
push()	addFirst()
pop()	removeFirst()
peek()	peekFirst()

EXERCISE

- Implement a Java program to create a double ended queue and try all the in-built methods