

Tutorial 2

Scope:

1. Java Collections and their iterations
2. Complexity, Searching & Sorting

Java Collections

Collect ion	Description	Key Methods
Map	Stores key-value pairs, no duplicate keys	<code>put(K key, V value)</code> , <code>get(Object key)</code> , <code>containsKey(Object key)</code> , <code>remove(Object key)</code> , <code>keySet()</code> , <code>values()</code>
Set	Stores unique elements, no duplicates allowed	<code>add(E e)</code> , <code>remove(Object o)</code> , <code>contains(Object o)</code> , <code>size()</code> , <code>isEmpty()</code> , <code>clear()</code>
List	Ordered collection, allows duplicates	<code>add(E e)</code> , <code>get(int index)</code> , <code>remove(int index)</code> , <code>size()</code> , <code>set(int index, E e)</code> , <code>contains(Object o)</code>
Stack	LIFO (Last In, First Out) structure, extends <code>Vector</code>	<code>push(E e)</code> , <code>pop()</code> , <code>peek()</code> , <code>isEmpty()</code> , <code>search(Object o)</code>
Queue	FIFO (First In, First Out) structure	<code>add(E e)</code> , <code>offer(E e)</code> , <code>remove()</code> , <code>poll()</code> , <code>peek()</code> , <code>isEmpty()</code>

Java interfaces

Collection	Common Implementations
Map<K, V>	HashMap<K, V>, TreeMap<K, V>, LinkedHashMap<K, V>, Hashtable<K, V>
Set	HashSet<E>, TreeSet<E>, LinkedHashSet<E>
List	ArrayList<E>, LinkedList<E>, Vector<E>
Stack	Stack<E>, ArrayDeque<E>
Queue	LinkedList<E>, PriorityQueue<E>, ArrayDeque<E>

Iterator

Why Use Iterator Instead of for-each?

1. Removing Elements:

`for-each` can't remove items during iteration, but `Iterator` allows it using `remove()`.

2. More Control:

`Iterator` gives manual control with `hasNext()` and `next()`, allowing conditional iteration and early exit.

3. Custom Logic:

For complex iteration logic, `Iterator` is better than the simplicity of `for-each`.

4. Legacy APIs:

Some older APIs only provide `Iterator`, not `for-each`.

Iterator - ForEach



```
List<String> list = new ArrayList<>(Arrays.asList("A", "B", "C"));
System.out.println("List:");
for (String item : list) {
    System.out.println(item); // Output: A, B, C
}
```



```
List<String> list = new ArrayList<>(Arrays.asList("A", "B", "C"));
Iterator<String> listIterator = list.iterator();
System.out.println("List:");
while (listIterator.hasNext()) {
    System.out.println(listIterator.next()); // Output: A, B, C
}
```

Iterator - ForEach



```
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

public class IteratorVsForEachMap {
    public static void main(String[] args) {
        Map<Integer, String> map = new HashMap<>();
        map.put(1, "Value1");
        map.put(2, "Value2");
        map.put(3, "Value3");

        // Using for-each loop
        System.out.println("Using for-each loop:");
        for (Map.Entry<Integer, String> entry : map.entrySet()) {
            System.out.println("Key: " + entry.getKey() + ", Value: " + entry.getValue());
        }

        // Using Iterator
        System.out.println("\nUsing Iterator:");
        Iterator<Map.Entry<Integer, String>> iterator = map.entrySet().iterator();
        while (iterator.hasNext()) {
            Map.Entry<Integer, String> entry = iterator.next();
            System.out.println("Key: " + entry.getKey() + ", Value: " + entry.getValue());
        }
    }
}
```

Generics

```
import java.util.ArrayList;
import java.util.List;

public class GenericsExample {

    public static void main(String[] args) {

        // Without Generics
        List list = new ArrayList(); // No type specified
        list.add("Hello");
        list.add(10); // Can add any type, no restriction

        // Requires casting to retrieve elements
        String str1 = (String) list.get(0);
        System.out.println("Without Generics: " + str1);

        // With Generics
        List<String> stringList = new ArrayList<>(); // Restricts to Strings
        stringList.add("Hello");
        // stringList.add(10); // Compile-time error, prevents wrong type

        // No casting needed
        String str2 = stringList.get(0);
        System.out.println("With Generics: " + str2);
    }
}
```

Interface



An interface in Java defines a set of methods that a class must implement. It only provides method signatures, not their implementations.

```
import java.util.*;

public class Main {
    public static void main(String[] args) {

        // WRONG USAGE (WHY ?) //There are interfaces
        // Will not run
        Map<String, Integer> map = new Map<>(); // WRONG
        Set<String> set = new Set<>(); // WRONG
        List<String> list = new List<>(); // WRONG
        Stack<Integer> stack = new Stack<>(); //*** CORRECT, why ?
        Queue<String> queue = new Queue<>(); // WRONG

        // CORRECT USAGE:
        Map<String, Integer> map = new HashMap<>(); // HashMap implements Map
        Set<String> set = new HashSet<>(); // HashSet implements Set
        List<String> list = new ArrayList<>(); // ArrayList implements List
        Stack<Integer> stack = new Stack<>(); // Stack implements List
        Queue<String> queue = new LinkedList<>(); // LinkedList implements Queue

    }
}
```


Collection Syntax



```
public class ListExample {  
  
    public static void main(String[] args) {  
  
        ///VALID-----  
        /// Interface = Implemented  
        List<String> list1 = new ArrayList<>();  
        // Only List (interface) methods can be accessed.  
        // However, they are implemented according to the ArrayList class.  
        // => In LinkedList (which also implements the List interface),  
        // the method implementations can vary, meaning LinkedList will provide-  
        // its own specific way of handling the same List methods.  
  
        // Implemented = Implemented  
        ArrayList<String> list4 = new ArrayList<>();  
        // You can use all methods from ArrayList,  
        // including methods like ensureCapacity(), which are not part of the List interface.  
        list4.ensureCapacity(100); // Example of ArrayList-specific method  
  
        ///INVALID-----  
        /// Implemented = Interface  
        ArrayList<String> list2 = new List<>(); //WRONG  
        // This is invalid because you cannot instantiate an interface.  
        // Uncommenting this will result in a compilation error.  
  
        /// Interface = Interface  
        List<String> list3 = new List<>(); //WRONG  
        // Also invalid because an interface cannot be instantiated.  
        // List is an interface, so it cannot be used with the 'new' keyword to create an object.  
    }  
}
```

Same method, different implementation

```
import java.util.HashSet;
import java.util.TreeSet;
import java.util.Set;

public class SetExample {
    public static void main(String[] args) {

        // Using HashSet
        Set<String> hashSet = new HashSet<>();
        Set<String> treeSet = new TreeSet<>();
        treeSet.add("Banana"); hashSet.add("Banana");
        treeSet.add("Apple"); hashSet.add("Apple");
        treeSet.add("Mango"); hashSet.add("Mango");
        treeSet.add("Cherry"); hashSet.add("Cherry");

        System.out.println("HashSet: " + hashSet);
        System.out.println("TreeSet: " + treeSet);

        /* OUTPUT
        TreeSet: [Apple, Banana, Cherry, Mango]
           // Sorted in natural (alphabetical) order.

        HashSet: [Mango, Banana, Apple, Cherry]
           // An arbitrary order => may change
        */
    }
}
```

Study for home :Collections

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;

public class CollectionOfCollectionsExample {
    public static void main(String[] args) {
        // Creating a map where the key is an Integer and the value is a List of Strings
        Map<Integer, List<String>> mapOfLists = new HashMap<>();

        // Adding values to the map
        List<String> list1 = new ArrayList<>();
        list1.add("Item1");
        list1.add("Item2");
        mapOfLists.put(1, list1);

        List<String> list2 = new ArrayList<>();
        list2.add("Item3");
        list2.add("Item4");
        mapOfLists.put(2, list2);

        // Using for-each loop
        System.out.println("Using for-each loop:");
        for (Map.Entry<Integer, List<String>> entry : mapOfLists.entrySet()) {
            System.out.println("Key: " + entry.getKey());
            for (String item : entry.getValue()) {
                System.out.println("Value: " + item);
            }
        }

        // Using Iterator
        System.out.println("\nUsing Iterator:");
        Iterator<Map.Entry<Integer, List<String>>> mapIterator = mapOfLists.entrySet().iterator();
        while (mapIterator.hasNext()) {
            Map.Entry<Integer, List<String>> entry = mapIterator.next();
            System.out.println("Key: " + entry.getKey());

            Iterator<String> listIterator = entry.getValue().iterator();
            while (listIterator.hasNext()) {
                String item = listIterator.next();
                System.out.println("Value: " + item);
            }
        }
    }
}
```

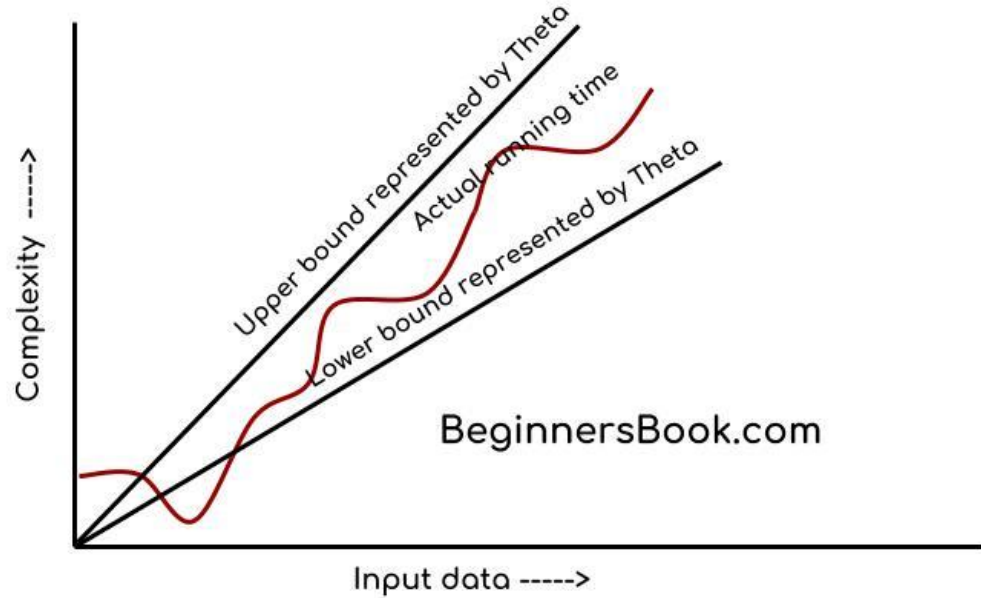
Complexity

Upper Bound (O): Maximum possible time.

Lower Bound (Ω): Minimum possible time.

Tight Bound (Θ): Average, typical time.

Image: [here](#).



1. Constant Time ($O(1)$, $\Omega(1)$, $\Theta(1)$)

Upper Bound: $O(1)$

This operation always takes the same amount of time, regardless of the array size. Accessing the first element is a direct memory access.

Lower Bound: $\Omega(1)$

The best case is the same: accessing the first element is constant time.

Average Case: $\Theta(1)$

The typical or average case also involves only one step, making the time constant.



```
public class ConstantTime {  
    public static int getFirstElement(int[] arr) {  
        return arr[0]; // Constant time operation  
    }  
}
```

2. Binary Search ($O(\log n)$, $\Omega(1)$, $\Theta(\log n)$)

Upper Bound: $O(\log n)$

In the worst case, you keep halving the search space until it becomes small. Halving occurs $\log n$ times, so the maximum number of steps is proportional to $\log n$.

Lower Bound: $\Omega(1)$

The best case occurs when the middle element is the target, so you find it in one step.

Average Case: $\Theta(\log n)$

Typically, you'll have to search through half the array repeatedly, which means logarithmic time is also the average case.

```
public class BinarySearch {  
    public static int binarySearch(int[] arr, int target) {  
        int left = 0, right = arr.length - 1;  
  
        while (left <= right) {  
            int mid = left + (right - left) / 2;  
  
            if (arr[mid] == target) return mid;  
            if (arr[mid] < target) left = mid + 1;  
            else right = mid - 1;  
        }  
        return -1; // Element not found  
    }  
}
```

3. Linear Search ($O(n)$, $\Omega(1)$, $\Theta(n)$)

Upper Bound: $O(n)$

In the worst case, the target element could be the last element or not present at all, so you'd have to check all n elements.

Lower Bound: $\Omega(1)$

The best case happens when the target is the first element, found in one comparison.

Average Case: $\Theta(n)$

On average, the target will be found halfway through the array, meaning you'll need to search through $n/2$ elements, which simplifies to n .



```
public class LinearSearch {  
    public static int linearSearch(int[] arr, int target) {  
        for (int i = 0; i < arr.length; i++) {  
            if (arr[i] == target) return i;  
        }  
        return -1; // Element not found  
    }  
}
```

4. Merge Sort ($O(n \log n)$, $\Omega(n \log n)$, $\Theta(n \log n)$)

Upper Bound: $O(n \log n)$

In the worst case, the array is split into halves $\log n$ times (because each division halves the size). Each level of recursion requires n comparisons to merge the arrays, so the time complexity becomes $n \log n$.

Lower Bound: $\Omega(n \log n)$

Even in the best case, the array must be split and merged the same way, resulting in $n \log n$ comparisons.

Average Case: $\Theta(n \log n)$

Whether the input is random or ordered, the process of splitting and merging remains the same, meaning the average case is also $n \log n$.



```
public class MergeSort {
    public static void mergeSort(int[] arr, int left, int right) {
        if (left < right) {
            int mid = (left + right) / 2;

            mergeSort(arr, left, mid);
            mergeSort(arr, mid + 1, right);
            merge(arr, left, mid, right);
        }
    }

    private static void merge(int[] arr, int left, int mid, int right) {
        int[] temp = new int[right - left + 1];
        int i = left, j = mid + 1, k = 0;

        while (i <= mid && j <= right) {
            if (arr[i] < arr[j]) temp[k++] = arr[i++];
            else temp[k++] = arr[j++];
        }

        while (i <= mid) temp[k++] = arr[i++];
        while (j <= right) temp[k++] = arr[j++];

        System.arraycopy(temp, 0, arr, left, temp.length);
    }
}
```


5. Bubble Sort ($O(n^2)$, $\Omega(n)$, $\Theta(n^2)$)

Upper Bound: $O(n^2)$

In the worst case (when the array is in reverse order), the algorithm has to make n passes, each involving up to n comparisons, leading to n^2 comparisons.

Lower Bound: $\Omega(n)$

In the best case (if the array is already sorted), a single pass through the array is enough to verify that no swaps are needed, resulting in n comparisons.

Average Case: $\Theta(n^2)$

Typically, the algorithm will need multiple passes and comparisons, which results in quadratic time complexity.

```
public class BubbleSort {  
    public static void bubbleSort(int[] arr) {  
        int n = arr.length;  
        for (int i = 0; i < n - 1; i++) {  
            boolean swapped = false;  
            for (int j = 0; j < n - 1 - i; j++) {  
                if (arr[j] > arr[j + 1]) {  
                    int temp = arr[j];  
                    arr[j] = arr[j + 1];  
                    arr[j + 1] = temp;  
                    swapped = true;  
                }  
            }  
            if (!swapped) break;  
        }  
    }  
}
```

6. Fibonacci (Recursive) ($O(2^n)$, $\Omega(2^n)$, $\Theta(2^n)$)

Upper Bound: $O(2^n)$

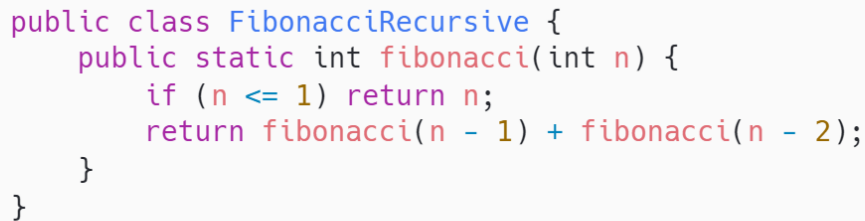
In the worst case, the recursion tree grows exponentially. Each call to `fibonacci` makes two recursive calls, leading to 2^n growth.

Lower Bound: $\Omega(2^n)$

Even in the best case, the recursive structure does not change, and the number of function calls remains exponential..

Average Case: $\Theta(2^n)$

The average case behaves similarly to the worst case because the recursive structure leads to exponential growth in function calls in both cases.



```
public class FibonacciRecursive {  
    public static int fibonacci(int n) {  
        if (n <= 1) return n;  
        return fibonacci(n - 1) + fibonacci(n - 2);  
    }  
}
```

Summary : The order of growth of functions

Algorithm	Upper Bound (O)	Lower Bound (Ω)	Average Case (Θ)
Constant Time	$O(1)$	$\Omega(1)$	$\Theta(1)$
Binary Search	$O(\log n)$	$\Omega(1)$	$\Theta(\log n)$
Linear Search	$O(n)$	$\Omega(1)$	$\Theta(n)$
Merge Sort	$O(n \log n)$	$\Omega(n \log n)$	$\Theta(n \log n)$
Bubble Sort	$O(n^2)$	$\Omega(n)$	$\Theta(n^2)$
Fibonacci (Recursive)	$O(2^n)$	$\Omega(2^n)$	$\Theta(2^n)$

Task 1 (ArrayList):

Develop a program Main.java that will:

- Create an ArrayList of 2 elements
- Add items to the ArrayList:
 - Try to insert strings (i.e., `String()` object) as well as integers (i.e., `Integer()`)
 - E.g., `list.add(new String("x")); list.add(2, new Integer(10));`
 - And print the list
 - Remove an item and print the list and check if the item you removed exists
 - Using `contains()` check if the list contains a specified element of your choice and print the output (tip: should be a Boolean `true` or `false`)
- Using the `ListIterator`, use a `while` loop and print every element of your list
 - Try also if you can do that using any other type of a loop (e.g., `for`)
- Create an object array from the ArrayList you have.

Task 2 (LinkedList):

Develop a program IntegerList.java that will create a LinkedList and add 4 Integer objects to it:

- Print the size of the LinkedList
- Add integer objects at the beginning and the end of the LinkedList and print them
 - Tip: You might have to use `getFirst()` and `getLast()` methods of the LinkedList class — check the documentation.
- Remove the first and last elements of the list and print the list
- Remove the first instance of `Integer(1)` object and print the list
- Add a `String` named as “NewYork” and `Long` objects to the LinkedList and print the list
- Get the index of the “New York” String object and print it
- Remove the 3rd object in the LinkedList and print the list
- Set the value of the second item to “one” and print the list
- Clone the LinkedList object (tip: using `list.clone()`) and print it.

Task 3 (Set):

Develop a program SetTest.java that will:

- Create a HashSet of 5 elements
 - Try to add both String and Integer objects to the HashSet (e.g., `set.add("hello");`, `set.add(42);`);
 - Print the HashSet
- Add new elements to the HashSet
 - Add 3 new elements to the set and print the updated set
- Remove an element from the HashSet
 - Remove one element and check if the removal was successful by printing the updated HashSet
- Check if a specific element is present
 - Use the `contains()` method to check if a specific element exists in the set (tip: Should return a Boolean value)
- Iterate over the elements in the HashSet and print them
 - Use a for-each loop to print every element in the set
- Test the behaviour of adding duplicate elements
 - Add a duplicate element and print the HashSet to see if duplicates are stored
- Create a TreeSet from the HashSet
 - Convert your HashSet to a TreeSet and print the elements in natural order.

Task 4 – Searching & Sorting

Exercise 1:

Step 1:

For this exercise, you need to read the file cars.csv and create an array list to insert each car. The CSV file has the following columns: - price, brand, model, year, mileage, color, vin

While you are reading the file, you need to ignore the first row since it contains the column's names. Also, note that each column is separated with a comma ,. Use the class Car.java that is uploaded on Blackboard.

Step 2:

Use Merge sort to sort the list of cars based on the field year in descending order. (From the newest to older).

Then print the newest car (considering there is only one with that year) and then change the year field to 2022.

Step 3:

Write a method public Car[] duplicatesBinarySearch(int year) that uses Binary search to find the car/cars with the given manufacture year. You need to take into consideration that some cars have the same manufacture year.

Step 4:

Use the above method to get the following data and print: 1. The details of the car with a year 2013 2. The details of the cars with a year 2015

Exercise 2:

Write a code that implements the Quicksort algorithm that uses the median as a pivot.

Then sort the following array int[] unsorted = {3, 10, 1, 45, 14, 22, 5, 36}.