Collections in Java

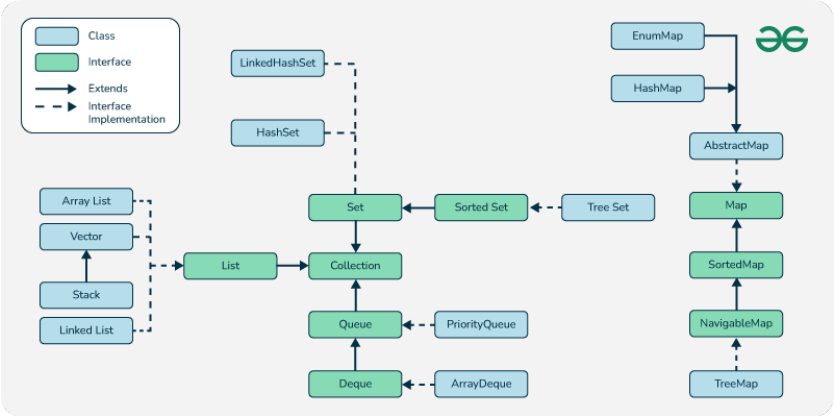
Autor: Jonathan Carrero

Contents

1 Introduction

Collections allow you to manage groups of data: store, recover and manipulate them. They are a set of interfaces, concrete and abstract classes that define abstract data types in Java that implement collections of elements. For example list, stack, queue, set, map.

Figure 1: Mind map of the different structures



Imagine you're a librarian responsible for managing a vast number of books. You need a system to keep track of all these books efficiently. In Java, the concept of *Collections* serves a similar purpose. Collections are like the digital equivalent of your organizational system in the library. They allow you to group and manage objects in a structured and flexible way.

Task: taking a look at the figure, are there instances of objects that would fail to be created? Give some examples.

As you can imagine, collections can have a certain order. This order could be linear, or non-linear, so there are different structures so that we can use each of them when we need them most.

Moreover, when working on real world Java applications, you often deal with large amounts of data. This data needs to be organized and accessed efficiently. Without collections, you would have to write a lot of custom code to manage data structures, difficult to maintain, and hard to scale.

Java Collections provide a standardized way to manage this data. They come with built in methods that simplify complex tasks like sorting, searching, and filtering. Indeed, collections are dynamic, so they can grow or shrink as needed, which is a significant advantage over arrays, which are fixed in size.

Figure 2: Data structures

Data Structures	Description
Array	Ordered collection of elements of the same data type.
Linked List	Dynamic data structure where elements are connected through nodes.
Stack	Follows Last In, First Out (LIFO) principle.
Queue	Follows First In, First Out (FIFO) principle.
Tree	Hierarchical data structures with a root node and branches.
Graph	Consists of nodes (vertices) and edges representing connections.

Task: search the Internet for the graphical representation of the data structures we have seen, so you can get an idea of how the information is handled and structured.

Figure 3: Algorithms

Algorithms	Description	Example
Sorting Algorithms	Techniques to arrange elements in a specific order.	Quick Sort, Merge Sort
Searching Algorithms	Techniques to find an element's presence and locate its position.	Binary Search, Linear Search
Graph Algorithms	Algorithms used to traverse or search through graphs.	Breadth-First Search, Depth-First Search
Dynamic Programming	Optimizing problems with overlapping subproblems.	Fibonacci Series
Divide and Conquer	Breaks a problem into smaller subproblems and solves them.	Binary Search
Greedy Algorithms	Make locally optimal choices at each stage.	Dijkstra's Algorithm

Task: search the internet for a video representation of how the algorithms we have seen work. Look at the differences in each of them and realise that they all have a purpose and must be used for a specific case.

2 Data Structures

Data structures are used to organize and store data in a computer. They help us keep our information in order and make it easy to use. Think of data structures like different types of boxes or containers for your things.

What are data structures?

- Data structures are special formats for organizing and storing data
- They are like containers that hold information in a specific way
- Examples include lists, stacks, and trees

Why are they important?

- Help manage and organize large amounts of data
- Make it easier and faster to find and use information
- Allow computers to work more efficiently
- Essential for writing good, fast computer programs

We'll start with the simpler ones and move to more complex structures. For each data structure, we will also look at their implementations and time complexity.

2.1 Arrays

Arrays are one of the simplest and most common data structures in Java. They are like a row of boxes where you can store things. An array is a collection of items of the same type, it stores items in order, one after another, each item has a number (called an index) to find it easily and they have a fixed size that doesn't change after creation.

What does the following code do?

```java
int[] numbers = new int[5];

numbers[0] = 10;
numbers[1] = 20;
numbers[2] = 30;
numbers[3] = 40;
numbers[4] = 50;

System.out.println("The third number is: " + numbers[2]);

for (int i = 0; i < numbers.length; i++) {
    System.out.println("Number " + (i + 1) + " is: " +
        numbers[i]);
}
```

Arrays are useful when you know how many items you need to store and when you want to access items quickly by their position.

Time complexity for array operations:

- Access: $O(1)$
- Search: $O(n)$
- Insertion: $O(n)$
- Deletion: $O(n)$

2.2 Lists

Lists are like improved arrays. They can grow or shrink as needed. Java has two main types of lists: ArrayList and LinkedList.

2.2.1 ArrayList

An ArrayList in Java is a resizable array-like data structure that allows you to store a list of elements. Unlike regular arrays, ArrayList can automatically adjust its size when you add or remove elements, making it more flexible. It provides methods to easily add, remove, and access elements, and it maintains the order in which elements are inserted.

Characteristics of an ArrayList:

- Similar to an array, but can change size.

- Good for storing and accessing items quickly.
- Not good for adding or removing items in the middle.

What does the following code do?

```
14 import java.util.ArrayList;
15
16 public class ArrayListExample {
17     public static void main(String[] args) {
18         ArrayList<String> fruits = new ArrayList<String>();
19         fruits.add("Apple");
20         fruits.add("Banana");
21         fruits.add("Cherry");
22
23         System.out.println("Fruits:" + fruits);
24         System.out.println("Second fruit:" + fruits.get(1));
25         fruits.set(0, "Apricot");
26         fruits.remove("Cherry");
27         System.out.println("Updated fruits: " + fruits);
28     }
29 }
```

Time complexity for ArrayList operations:

- Access: $O(1)$
- Search: $O(n)$
- Insertion (at end): $O(1)$
- Insertion (at specific index): $O(n)$
- Deletion (at end): $O(1)$
- Deletion (at specific index): $O(n)$

2.2.2 LinkedList

A LinkedList in Java is a data structure that consists of a sequence of elements, where each element is linked to the next and previous elements, forming a chain. Unlike an ArrayList, which stores elements in a contiguous block of memory, a LinkedList stores each element in a separate node that contains the element itself and references to the next and previous nodes. This makes LinkedList efficient for insertions and deletions at any position but slower for accessing elements by index.

Characteristics of an LinkedList:

- Made of separate items linked together
- Good for adding or removing items anywhere in the list
- Not as good for accessing items by position

What does the following code do?

```
30 import java.util.LinkedList;
31
32 public class LinkedListExample {
33     public static void main(String[] args) {
34         LinkedList<Integer> numbers = new LinkedList<Integer
               >();
35         numbers.add(10);
36         numbers.add(20);
37         numbers.add(30);
38
39         System.out.println("Numbers: " + numbers);
40         numbers.addFirst(5);
41         numbers.addLast(40);
42         System.out.println("Updated numbers: " + numbers);
43         numbers.removeFirst();
44         System.out.println("Final numbers: " + numbers);
45     }
46 }
```

Time complexity for LinkedList operations:

- Access: $O(1)$
- Search: $O(n)$
- Insertion (at beginning/end): $O(1)$
- Insertion (at specific index): $O(n)$
- Deletion (at beginning/end): $O(1)$
- Deletion (at specific index): $O(n)$

2.3 Stacks

A Stack in Java is a data structure that follows the Last-In-First-Out (LIFO) principle, meaning the last element added is the first one to be removed. You

can think of it like a stack of plates: you add new plates on top, and when you need one, you take the topmost plate. In Java, a Stack allows operations like push (to add elements), pop (to remove the top element), and peek (to look at the top element without removing it). It's commonly used for scenarios like backtracking, undo mechanisms, and parsing expressions. Thus, they are useful for tasks that need to reverse order or keep track of what happened last.

Task: without looking at the code, create a Stack<T> class that simulates a stack using an ArrayList<T>.

We can create a simple stack using an ArrayList. Here's an example:

```
47 import java.util.ArrayList;
48
49 public class Stack<T> {
50     private ArrayList<T> stack;
51
52     public Stack() {
53         stack = new ArrayList<T>();
54     }
55
56     // Add an item to the top of the stack
57     public void push(T item) {
58         stack.add(item);
59     }
60
61     // Remove and return the top item from the stack
62     public T pop() {
63         if (isEmpty()) {
64             throw new IllegalStateException("Stack is empty"
                   );
65         }
66         return stack.remove(stack.size() - 1);
67     }
68
69     // Look at the top item without removing it
70     public T peek() {
71         if (isEmpty()) {
72             throw new IllegalStateException("Stack is empty"
                   );
73         }
74         return stack.get(stack.size() - 1);
75     }
```

```
76
77      // Check if the stack is empty
78      public boolean isEmpty() {
79          return stack.isEmpty();
80      }
81
82      // Get the size of the stack
83      public int size() {
84          return stack.size();
85      }
86  }
```

Here's how to use this Stack:

```
87  public class StackExample {
88      public static void main(String[] args) {
89          Stack<String> bookStack = new Stack<String>();
90
91          // Add books to the stack
92          bookStack.push("Book 1");
93          bookStack.push("Book 2");
94          bookStack.push("Book 3");
95
96          System.out.println("Stack size: " + bookStack.size()
                ;
97          System.out.println("Top book: " + bookStack.peek());
98
99          // Remove the top book
100         String removedBook = bookStack.pop();
101         System.out.println("Removed book: " + removedBook);
102
103         System.out.println("New stack size: " + bookStack.
                size());
104         System.out.println("New top book: " + bookStack.peek
                ());
105     }
106 }
```

Time complexity of Stack operations:

- Push: $O(1)$
- Pop: $O(1)$

- Peek: $O(1)$
- Search: $O(n)$

2.4 Queues

A Queue is another important data structure. It works like a line of people waiting for something: the first person who joins the line is the first one to leave it. It follows the First-In-First-Out (FIFO) principle, meaning the first element added is the first one to be removed. It's like a line of people waiting for service: the first person in line is served first. In Java, a Queue provides operations like offer (to add elements), poll (to remove and return the front element), and peek (to view the front element without removing it). It's also useful for managing tasks in order, like print jobs or customer service requests.

Task: without looking at the code, create a Queue<T> class that simulates a queue using an LinkedList<T>.

We can create a simple queue using a LinkedList. Here's an example:

```
107 import java.util.LinkedList;
108
109 public class Queue<T> {
110     private LinkedList<T> queue;
111
112     public Queue() {
113         queue = new LinkedList<T>();
114     }
115
116     // Add an item to the back of the queue
117     public void enqueue(T item) {
118         queue.addLast(item);
119     }
120
121     // Remove and return the front item from the queue
122     public T dequeue() {
123         if (isEmpty()) {
124             throw new IllegalStateException("Queue is empty"
                    );
125         }
126         return queue.removeFirst();
127     }
128
```

```
    // Look at the front item without removing it
    public T peek() {
        if (isEmpty()) {
            throw new IllegalStateException("Queue is empty");
        }
        return queue.getFirst();
    }

    // Check if the queue is empty
    public boolean isEmpty() {
        return queue.isEmpty();
    }

    // Get the size of the queue
    public int size() {
        return queue.size();
    }
}
```

Here's how to use this Queue:

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

        // Add customers to the queue
        customerQueue.enqueue("Customer 1");
        customerQueue.enqueue("Customer 2");
        customerQueue.enqueue("Customer 3");

        System.out.println("Queue size: " + customerQueue.
            size());
        System.out.println("Front customer: " +
            customerQueue.peek());

        // Remove the front customer
        String servedCustomer = customerQueue.dequeue();
        System.out.println("Served customer: " +
            servedCustomer);

        System.out.println("New queue size: " +
```

```
                customerQueue.size());
164         System.out.println("New front customer: " +
                customerQueue.peek());
165     }
166 }
```

Time complexity of Queue operations:

- Enqueue: $O(1)$
- Dequeue: $O(1)$
- Peek: $O(1)$
- Search: $O(n)$

2.5 Trees

A Tree in Java is a hierarchical data structure where each element, called a node, is connected to others in a parent-child relationship. The top node is called the root, and every node can have zero or more child nodes. Trees are useful for representing hierarchical data, such as file systems, organizational structures, or decision processes. In Java, there are specific types of trees like binary trees, where each node has at most two children, and balanced trees like TreeMap or TreeSet, which maintain elements in sorted order for fast lookups and insertions. Trees are commonly used in searching, sorting, and hierarchical representations.

2.6 Binary Search Trees (BST)

It is a type of binary tree where each node has at most two children, and the tree maintains a specific order. For any given node, the left child contains values that are less than the node's value, and the right child contains values greater than the node's value. This property makes searching, insertion, and deletion operations efficient, as it allows for quick narrowing down of where a value should be placed or found. Binary search trees are commonly used for searching, sorting, and maintaining a dynamic set of ordered data.

By the way, we will use the Comparable class. This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's natural ordering, and the class's compareTo method is referred to as its natural comparison method.

Now, let's implement our first tree. To keep it simple, let's create a tree where

each node stores an `int` value. Note that, although it is a BST tree, this tree is not self-balancing. Trees that have this property are AVL trees, which are outside the scope of this topic due to their complexity.

Task: follow the teacher's indications to develop your first BST tree impelmentation. It is very important to understand each of the steps, then there will be a BST implementation activity.

Here's a simple implementation of a binary search tree using some Java classes. Notice that data can be generic like <T>:

```
167 public class BinarySearchTree<T extends Comparable<T>> {
168     private TreeNode<T> root;
169 
170     public void insert(T data) {
171         root = insertRec(root, data);
172     }
173 
174     private TreeNode<T> insertRec(TreeNode<T> root, T data)
        {
175         if (root == null) {
176             root = new TreeNode<T>(data);
177             return root;
178         }
179 
180         if (data.compareTo(root.data) < 0)
181             root.left = insertRec(root.left, data);
182         else if (data.compareTo(root.data) > 0)
183             root.right = insertRec(root.right, data);
184 
185         return root;
186     }
187 
188     public boolean search(T data) {
189         return searchRec(root, data);
190     }
191 
192     private boolean searchRec(TreeNode<T> root, T data) {
193         if (root == null || root.data.equals(data))
194             return root != null;
195 
196         if (data.compareTo(root.data) < 0)
197             return searchRec(root.left, data);
```

```
198
199             return searchRec(root.right, data);
200         }
201     }
```

Here's how to use this Binary Search Tree:

```
202 public class BSTExample {
203     public static void main(String[] args) {
204         BinarySearchTree<Integer> bst = new BinarySearchTree
                <>();
205
206         // Insert some numbers
207         bst.insert(50);
208         bst.insert(30);
209         bst.insert(70);
210         bst.insert(20);
211         bst.insert(40);
212
213         // Search for numbers
214         System.out.println("Is 20 in the tree? " + bst.
                search(20));
215         System.out.println("Is 60 in the tree? " + bst.
                search(60));
216     }
217 }
```

Time complexity of BST operations:

Figure 4: Complexity operations in BST

Operations	Best case time complexity	Average case time comp.	Worst case time comp.
Insertion	O(log n)	O(log n)	O(n)
Deletion	O(log n)	O(log n)	O(n)
Search	O(log n)	O(log n)	O(n)

Task: this task could simulate an examination exercise. Without consulting the code provided by the teacher, implement a BST that stores an `int` at each node with all the methods seen (except delete when the node has two children).

We already know Java, so the difficulty in this exercise is not to implement, but to think about what you want to implement.

2.7 Hash Tables

A Hash Table in Java is a data structure that stores key-value pairs, allowing for fast retrieval of values based on their keys. They're widely used in many applications where quick access to data is crucial. However, if multiple keys map to the same index (a collision), additional techniques like chaining or open addressing are used to resolve the conflict. Hash tables are commonly implemented in Java through classes like HashMap or Hashtable.

Here's an example:

```
218 import java.util.HashMap;
219
220 public class HashMapExample {
221     public static void main(String[] args) {
222         // Create a new HashMap
223         HashMap<String, Integer> ages = new HashMap<>();
224
225         // Add key-value pairs to the map
226         ages.put("Alice", 25);
227         ages.put("Bob", 30);
228         ages.put("Carol", 35);
229
230         // Get a value from the map
231         int bobAge = ages.get("Bob");
232         System.out.println("Bob's age: " + bobAge);
233
234         // Check if a key exists
235         if (ages.containsKey("David")) {
236             System.out.println("David's age is known");
237         } else {
238             System.out.println("David's age is unknown");
239         }
240
241         // Update a value
242         ages.put("Alice", 26);
243
244         // Remove a key-value pair
245         ages.remove("Carol");
246
```

```
247         // Print all entries
248         System.out.println("\nAll entries:");
249         for (String name : ages.keySet()) {
250             System.out.println(name + ": " + ages.get(name))
                    ;
251         }
252     }
253 }
```

HashMap is very efficient for storing and retrieving data when you have a unique key for each piece of data. It's particularly useful when you need to frequently look up values based on their keys.

Time complexity of HashMap in Java:

- Insert: $O(1)$ average case, $O(1)$ worst case
- Delete: $O(1)$ average case, $O(1)$ worst case
- Search: $O(1)$ average case, $O(1)$ worst case

Task: check the documentation for the available Hash Table methods.

Task: we are going to make a small phonebook using Hash Tables. Listen to the teacher's implementation details to get an idea of what to do in each method. Here is an example of what your implementation should do.

```
254 --- Phone Book Menu ---
255 1. Add/Update Contact
256 2. Delete Contact
257 3. Search Contact
258 4. List All Contacts
259 5. Exit
260 Enter your choice: 1
261 Enter contact name: Jonathan
262 Enter phone number: 643887102
263 Contact added/updated successfully.
264
265 --- Phone Book Menu ---
266 1. Add/Update Contact
267 2. Delete Contact
268 3. Search Contact
269 4. List All Contacts
270 5. Exit
271 Enter your choice: 1
```

```
Enter contact name: Luis
Enter phone number: 644398705
Contact added/updated successfully.

--- Phone Book Menu ---
1. Add/Update Contact
2. Delete Contact
3. Search Contact
4. List All Contacts
5. Exit
Enter your choice: 4
Contacts in the phone book:
Name: Jonathan, Phone Number: 643887102
Name: Luis, Phone Number: 644398705

--- Phone Book Menu ---
1. Add/Update Contact
2. Delete Contact
3. Search Contact
4. List All Contacts
5. Exit
Enter your choice: 3
Enter the contact name to search: Luis
Luis phone number is: 644398705

--- Phone Book Menu ---
1. Add/Update Contact
2. Delete Contact
3. Search Contact
4. List All Contacts
5. Exit
Enter your choice: 2
Enter the contact name to delete: Jonathan
Contact deleted successfully.

--- Phone Book Menu ---
1. Add/Update Contact
2. Delete Contact
3. Search Contact
4. List All Contacts
5. Exit
Enter your choice: 4
```

```
Contacts in the phone book:
Name: Luis, Phone Number: 644398705

--- Phone Book Menu ---
1. Add/Update Contact
2. Delete Contact
3. Search Contact
4. List All Contacts
5. Exit
Enter your choice: 5
Exiting Phone Book.
```