#### **String Class**

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Parameters** | **Return Type** | **Description** |
| length() | None | int | Returns the length of the string. Useful in **looping**, **array/string constraints** in DSA. |
| charAt() | int index | char | Returns the character at the specified index. Used in **trie operations, hashing, and pattern matching**. |
| substring() | int beginIndex, int endIndex | String | Returns a substring from the given indices. Used in **string parsing, pattern matching**. **Example:** "HelloWorld".substring(0, 5) → "Hello". |
| indexOf() | String str | int | Returns the index of the first occurrence of the specified string. Useful in **searching substrings**. |
| replace() | String old, String new | String | Replaces occurrences of a substring with another string. Used in **modifying dynamic strings**. |
| toLowerCase() | None | String | Converts the string to lowercase. Used in **case-insensitive comparisons in hashing and searching**. |
| toUpperCase() | None | String | Converts the string to uppercase. Useful for **normalizing input data**. |
| split() | String regex | String[] | Splits the string into an array based on a delimiter. Useful in **tokenization, parsing CSV/log data**. |
| equals() | Object anotherString | boolean | Compares two strings for equality (case-sensitive). Used in **hashing, dictionary lookups**. **Example:** "Hello".equals("hello") → false. |
| compareTo() | String anotherString | int | Compares two strings lexicographically. Used in **sorting algorithms**. **Example:** "apple".compareTo("banana") → -1 (since "apple" is smaller). |

This makes String inefficient for frequent modifications (use StringBuffer or StringBuilder instead).  
  
Concatenation (+) inside loops leads to **O(n²) complexity** due to repeated object creation.

#### **StringBuilder Class**

* StringBuilder sb = new StringBuilder("Hello");

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Parameters** | **Return Type** | **Description** |
| append() | String str | StringBuilder | Appends the specified string to the sequence. **Example:** sb.append("World") on sb = new StringBuilder("Hello") results in "HelloWorld". |
| insert() | int offset, String str | StringBuilder | Inserts the string at the specified position. **Example:** sb.insert(5, "Java") on "Hello" gives "HelloJava". |
| delete() | int start, int end | StringBuilder | Deletes characters from the start index to the end index. **Example:** sb.delete(2, 4) on "abcdef" gives "abef". |
| deleteCharAt() | int index | StringBuilder | Deletes the character at the specified index. **Example:** sb.deleteCharAt(3) on "abcd" gives "abc". |
| reverse() | None | StringBuilder | Reverses the sequence. **Example:** sb.reverse() on "Hello" gives "olleH". |
| setCharAt() | int index, char ch | void | Sets the character at the specified index. **Example:** sb.setCharAt(1, 'a') on "Hello" gives "Hallo". |
| replace() | int start, int end, String str | StringBuilder | Replaces characters from start to end with the given string. **Example:** sb.replace(0, 2, "Hi") on "Hello" gives "Hi llo". |
| substring() | int start, int end | String | Returns a substring from start to end index. **Example:** sb.substring(1, 4) on "Hello" gives "ell". |
| capacity() | None | int | Returns the current capacity of the StringBuilder. |
| ensureCapacity() | int minimumCap | void | Ensures the minimum capacity of the StringBuilder object. |

Common functions between string and string buffer are :   
length(), charAt(), substring(), replace(), indexOf(), lastIndexOf(), toString(), equals(), compareTo()

#### **ArrayList Class**

* ArrayList<Integer> list = new ArrayList<>();

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Parameters** | **Return Type** | **Description** |
| add() | E element | boolean | Adds an element to the end of the list. **Example:** list.add(5); |
| add(index, element) | int index, E element | void | Inserts an element at the specified index. **Example:** list.add(1, 10); |
| get() | int index | E | Returns the element at the specified index. **Example:** list.get(2); |
| set() | int index, E element | E | Replaces the element at the given index. **Example:** list.set(0, 20); |
| remove() | int index or Object o | E or boolean | Removes the element at the specified index or the specified object. **Example:** list.remove(3); |
| size() | None | int | Returns the number of elements in the list. **Example:** list.size(); |
| contains() | Object o | boolean | Checks if the list contains a specific element. **Example:** list.contains(5); |
| indexOf() | Object o | int | Returns the index of the first occurrence of an element. **Example:** list.indexOf(10); |
| isEmpty() | None | boolean | Checks if the list is empty. **Example:** list.isEmpty(); |
| clear() | None | void | Removes all elements from the list. **Example:** list.clear(); |

ArrayList<Integer> list = new ArrayList<>(Arrays.asList(1,2,3,4,5));

Iterating over ArrayList

for (int i = 0; i < list.size(); i++) {

System.out.print(list.get(i) + " ");

}

**OR**

Iterator<Integer> iterator = list.iterator();

while (iterator.hasNext()) {

System.out.print(iterator.next() + " ");

}

**OR**

for (Integer num : list) {

System.out.print(num + " ");

}

**For sorting ArrayList and Arrays:**

import java.util.Comparator;  
  
Collections.sort(list);

Collections.sort(list, Comparator.reverseOrder());

**For sorting Objects using Comparable:**

class Student **implements** Comparable**<**Student**>** {

String name;

int age;

Student(*String name, int age*) {

this.name = name;

this.age = age;

}

// Implementing compareTo for sorting by age

**@Override**

public int **compareTo**(*Student other*) {

return Integer.compare(this.age, other.age); // Ascending order

}

}

**Calling Functions:**

// Sorting in Ascending Order (Natural Order)

Collections.sort(students);

// Sorting in Descending Order (Reverse Natural Order)

Collections.sort(students, Collections.reverseOrder());

Analyzing

return Integer.compare(this.age, other.age)  
return this.age – other.age;

* If 0, both are equal
* If -ve, this is smaller
* If +ve, other is smaller

#### **LinkedList Class**

* LinkedList<Integer> list = new LinkedList<>();

|  |  |  |  |
| --- | --- | --- | --- |
| Method | Parameters | Return Type | Description |
| addFirst() | E element | void | Inserts the element at the beginning. **Example:** list.addFirst(10); |
| addLast() | E element | void | Inserts the element at the end. **Example:** list.addLast(20); |
| removeFirst() | None | E | Removes and returns the first element. **Example:** list.removeFirst(); |
| removeLast() | None | E | Removes and returns the last element. **Example:** list.removeLast(); |
| getFirst() | None | E | Retrieves the first element without removing it. **Example:** list.getFirst(); |
| getLast() | None | E | Retrieves the last element without removing it. **Example:** list.getLast(); |
| pollFirst() | None | E | Retrieves and removes the first element, or returns null if empty. **Example:** list.pollFirst(); |
| pollLast() | None | E | Retrieves and removes the last element, or returns null if empty. **Example:** list.pollLast(); |
| offerFirst() | E element | boolean | Inserts the element at the front. **Example:** list.offerFirst(15); |
| offerLast() | E element | boolean | Inserts the element at the end. **Example:** list.offerLast(25); |

addFirst() is void, while offerFirst() returns a boolean indicating success/failure.

 **Exception Handling**: addFirst() throws an exception when it fails, whereas offerFirst() fails gracefully by returning false.

 **Usage**: offerFirst() is preferred in scenarios where capacity restrictions are in place and failures need to be handled without exceptions.

#### **Stack Class**

* Stack<Integer> stack = new Stack<>();

|  |  |  |  |
| --- | --- | --- | --- |
| Method | Parameters | Return Type | Description |
| push() | E element | E | Pushes an element onto the top of the stack. **Example:** stack.push(5); |
| pop() | None | E | Removes and returns the top element of the stack. **Example:** stack.pop(); |
| peek() | None | E | Returns the top element without removing it. **Example:** stack.peek(); |
| isEmpty() | None | boolean | Checks if the stack is empty. **Example:** stack.isEmpty(); |
| search() | Object o | int | Returns the position of an element from the top. **Example:** stack.search(10); |
| size() | None | int | Returns the number of elements in the stack. **Example:** stack.size(); |
| contains() | Object o | boolean | Checks if the stack contains a specific element. **Example:** stack.contains(5); |
| clear() | None | void | Removes all elements from the stack. **Example:** stack.clear(); |
| iterator() | None | Iterator<E> | Returns an iterator to traverse the stack. **Example:** Iterator<Integer> it = stack.iterator(); |
| clone() | None | Object | Returns a shallow copy of the stack. **Example:** Stack<Integer> stackCopy = (Stack<Integer>) stack.clone(); |

Iterating over Stack

Iterator<Integer> iterator = stack.iterator();

while (iterator.hasNext()) {

System.out.print(iterator.next() + " ");

}

**OR**

for (Integer num : stack) {

System.out.print(num + " ");

}

#### **Queue Class**

* Queue<Integer> queue = new LinkedList<>();

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Parameters** | **Return Type** | **Description** |
| add() | E element | boolean | Inserts the specified element into the queue. Throws an exception if full. Example: queue.add(10); |
| offer() | E element | boolean | Inserts the specified element into the queue. Returns false if full. Example: queue.offer(20); |
| remove() | None | E | Retrieves and removes the head of the queue. Throws an exception if empty. Example: int head = queue.remove(); |
| poll() | None | E | Retrieves and removes the head of the queue. Returns null if empty. Example: Integer head = queue.poll(); |
| element() | None | E | Retrieves the head of the queue without removing it. Throws exception if empty. Example: int head = queue.element(); |
| peek() | None | E | Retrieves the head of the queue without removing it. Returns null if empty. Example: Integer head = queue.peek(); |
| size() | None | int | Returns the number of elements in the queue. Example: int size = queue.size(); |
| isEmpty() | None | boolean | Checks if the queue is empty. Example: boolean empty = queue.isEmpty(); |
| contains() | Object o | boolean | Checks if the queue contains a specific element. Example: boolean exists = queue.contains(30); |
| clear() | None | void | Removes all elements from the queue. Example: queue.clear(); |

Iterating over Queue

Iterator<Integer> iterator = queue.iterator();

while (iterator.hasNext()) {

System.out.print(iterator.next() + " ");

}

**OR**

for (Integer num : queue) {

System.out.print(num + " ");

}

#### **Priority Queue Class**

* PriorityQueue<Integer> pq = new PriorityQueue<>();

|  |  |  |  |
| --- | --- | --- | --- |
| Method | Parameters | Return Type | Description |
| add() | E element | boolean | Inserts the specified element into the priority queue. Throws an exception if full. Example: pq.add(40); |
| offer() | E element | boolean | Inserts the specified element into the priority queue. Returns false if full. Example: pq.offer(10); |
| remove() | None | E | Retrieves and removes the head of the priority queue. Throws an exception if empty. Example: int head = pq.remove(); |
| poll() | None | E | Retrieves and removes the head of the priority queue. Returns null if empty. Example: Integer head = pq.poll(); |
| peek() | None | E | Retrieves the head of the priority queue without removing it. Returns null if empty. Example: Integer head = pq.peek(); |
| size() | None | int | Returns the number of elements in the priority queue. Example: int size = pq.size(); |
| isEmpty() | None | boolean | Checks if the priority queue is empty. Example: boolean empty = pq.isEmpty(); |
| contains() | Object o | boolean | Checks if the priority queue contains a specific element. Example: boolean exists = pq.contains(20); |
| clear() | None | void | Removes all elements from the priority queue. Example: pq.clear(); |
| comparator() | None | Comparator<? super E> | Returns the comparator used for ordering elements, or null if natural ordering is used. Example: Comparator cmp = pq.comparator(); |

Iterator on Priority Queue

Iterator<Integer> iterator = pq.iterator();

while (iterator.hasNext()) {

System.out.print(iterator.next() + " ");

}

**OR**

for (Integer num : pq) {

System.out.print(num + " ");

}

#### **HashMap Class**

* HashMap<Integer, String> map = new HashMap<>();

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Parameters** | **Return Type** | **Description** |
| put() | K key,  V value | V | Inserts a key-value pair into the HashMap. If the key exists, updates the value. Example: map.put("A", 1); |
| get() | Object key | V | Retrieves the value associated with the given key. Returns null if the key does not exist. Example: map.get("A"); |
| remove() | Object key | V | Removes the key-value pair for the given key. Example: map.remove("A"); |
| containsKey() | Object key | boolean | Checks if the HashMap contains the specified key. Example: map.containsKey("A"); |
| containsValue() | Object value | boolean | Checks if the HashMap contains the specified value. Example: map.containsValue(1); |
| size() | None | int | Returns the number of key-value pairs in the HashMap. Example: map.size(); |
| isEmpty() | None | boolean | Checks if the HashMap is empty. Example: map.isEmpty(); |
| clear() | None | void | Removes all key-value pairs from the HashMap. Example: map.clear(); |
| keySet() | None | Set<K> | Returns a set of all keys in the HashMap. Example: Set<String> keys = map.keySet(); |
| values() | None | Collection<V> | Returns a collection of all values in the HashMap. Example: Collection<Integer> values = map.values(); |
| entrySet() | None | Set<Map.Entry<K,V>> | Returns a set of all key-value pairs in the HashMap. Example: Set<Map.Entry<String, Integer>> entries = map.entrySet(); |

Iteration of HashMap:

1. Using entrySet() with a for-each loop (Recommended)

HashMap<Integer, String> map = new HashMap<>();

map.put(1, "Apple");

map.put(2, "Banana");

map.put(3, "Cherry");

for (Map.Entry<Integer, String> entry : map.entrySet()) {

System.out.println("Key: " + entry.getKey() + ", Value: " + entry.getValue());

}

1. Using keySet() to iterate over keys

for (Integer key : map.keySet()) {

System.out.println("Key: " + key + ", Value: " + map.get(key));

}

1. Using values() to iterate over values only

for (String value : map.values()) {

System.out.println("Value: " + value);

}

#### **HashSet Class**

* HashSet<String> set = new HashSet<>();

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Parameters** | **Return Type** | **Description** |
| add() | E e | boolean | Adds the specified element to the HashSet if it is not already present. Example: set.add(10); |
| remove() | Object o | boolean | Removes the specified element from the HashSet if it exists. Example: set.remove(10); |
| contains() | Object o | boolean | Checks if the HashSet contains the specified element. Example: set.contains(10); |
| size() | None | int | Returns the number of elements in the HashSet. Example: set.size(); |
| isEmpty() | None | boolean | Checks if the HashSet is empty. Example: set.isEmpty(); |
| clear() | None | void | Removes all elements from the HashSet. Example: set.clear(); |
| iterator() | None | Iterator<E> | Returns an iterator to iterate through the HashSet elements. Example: Iterator<Integer> it = set.iterator(); |
| forEach() | Consumer<E> | void | Performs an action for each element in the HashSet. Example: set.forEach(System.out::println); |
| clone() | None | Object | Creates a shallow copy of the HashSet. Example: HashSet<Integer> cloneSet = (HashSet<Integer>) set.clone(); |
| toArray() | None | Object[] | Converts the HashSet into an array. Example: Object[] arr = set.toArray(); |
| toArray(T[]) | T[] a | T[] | Converts the HashSet into an array of the specified type. Example: Integer[] arr = set.toArray(new Integer[0]); |

Iteration of HashSet

1. Using an **Enhanced for loop**:

for (String fruit : set) {

System.out.println(fruit);

}

 **HashMap** stores **key-value pairs**, where keys must be unique but values can be duplicated.

 **HashSet** only stores **unique elements** without any key-value mapping.