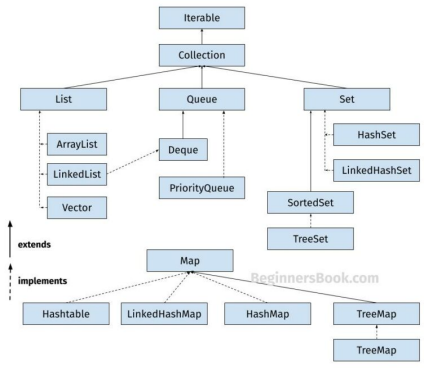
Java Collections



Arrays Class

import java.util.Arrays;
// Imports Arrays class from java.util package
/*Key Points about java.util.Arrays:
1. **Package: java.util**Contains utility classes for working with arrays
2. Arrays Class Features:
- Provides static methods for array operations
- Common operations include:
sort()- Sorting arrays
parallelSort() - Parallel sorting for large arrays
fill()- Filling arrays with values
binarySearch()- Searching in sorted arrays
copyOf()- Creating copies of arrays
equals()- Comparing arrays
toString() - Converting array to String
3. Usage Examples:
Arrays.sort(array) // Sorting
Arrays.fill(array, val) // Filling
Arrays.toString(array) // String representation
Arrays.copyOf(array, len) // Copying
Arrays.equals(arr1, arr2) // Comparing
4. Benefits:
- Simplified array manipulation
- Optimized implementations

Arrays Class (cont)

- Consistent behavior across different array types*/

Arrays Class

//Declare and initialize an array of integers with values from 1 to 10
//The array must be sorted in ascending order for binary search to work correctly
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
// Perform binary search to find element 4 in the array
// Arrays.binarySearch() returns:
// Index of the element if found (zero-based)
// Negative value if element not found (-(insertion point) - 1)
// Note: Array must be sorted before using binarySearch
int index = Arrays.binarySearch(numbers, 4);
// Print the result
// In this case, it will print index 3 because 4 is at position 3 (zero-based indexing)
System.out.println("The index of element 4 in the array is " + index);
//Output - **The index of element 4 in the array is 3**
/* Additional Notes:
1. Binary search is much faster than linear search for large arrays
2. Time complexity is O(log n) where n is array length
3. If array is not sorted, results will be unpredictable
4. If element is not found, return value will be negative
*/

Arrays Class

~// Initialize an Integer array with 10 elements
Integer[] numbers = { 10, 2, 32, 12, 15, 76, 17, 48, 79, 9 };
~// **Arrays.sort()** - Sorts array in ascending order using quicksort algorithm
~// After sort: [2, 9, 10, 12, 15, 17, 32, 48, 76, 79]
Arrays.sort(numbers);
~//**Arrays.parallelSort()** - Sorts array using parallel sorting for large arrays (threshold 8192 elements)
~// Since array is small (<8192), it will use same algorithm as Arrays.sort()
~// After parallel sort: [2, 9, 10, 12, 15, 17, 32, 48, 76, 79]
Arrays.parallelSort(numbers);
~// **Arrays.fill()** - Fills entire array with specified value (12)
~// After fill: [12, 12, 12, 12, 12, 12, 12, 12, 12, 12]
Arrays.fill(numbers, 12);
~// Print all elements of array using enhanced for loop
for (int i : numbers) {
System.out.print(i + " ");
}
/* Output:
12 12 12 12 12 12 12 12 12 12
*/

LinkedList,ArrayList

```
import java.util.ArrayList;
/*
1. Package: java.util
2. ArrayList class:
     - Implements List interface
     - Dynamic array implem -
ent ation
```

LinkedList,ArrayList (cont)

```
>    - Good for random access and storing/accessing elements
   - Not synchronized (not thread-safe)
   - Common operations:
     * add() - O(1) amortized
     * get() - O(1)
     * remove() - O(n)
     * contains() - O(n)
*/
import java.util.Iterator;
/*
1. Package: java.util
2. Iterator interface:
   - Provides methods to iterate through collections
   - Key methods:
     * hasNext() - checks if more elements exist
     * next() - returns next element
     * remove() - removes last element returned
   - Safer than for loop for modification during iteration
   - Used in for-each loops internally
*/
import java.util.LinkedList;
/*
1. Package: java.util
2. LinkedList class:
   - Implements List and Deque interfaces
   - Doubly-linked list implementation
   - Good for frequent insertions/deletions
   - Common operations:
     * add() - O(1)
     * get() - O(n)
     * remove() - O(1) if position known
     * contains() - O(n)
*/
import java.util.List;
```

LinkedList,ArrayList (cont)

```
> /*
1. Package: java.util
2. List interface:
   - Ordered collection (sequence)
   - Allows duplicates
   - Main implementations:
     * ArrayList
     * LinkedList
     * Vector (legacy, thread-safe)
   - Defines common list operations:
     * add()
     * remove()
     * get()
     * set()
     * etc.
*/
```

LinkedList,ArrayList

```
// Create a new LinkedList to store Integer values
List<I nte ger> list = new Linked Lis t<>();
// Add elements to the LinkedList
list.a dd(1);
list.a dd(2);
list.a dd(3);
System.ou t.p rin tln (list); // Output: [1, 2, 3]
list.a dd(4); // Add 4 at the end of the List
System.ou t.p rin tln (list); // Output: [1, 2, 3, 4]
list.a dd(1, 50); // Add 50 at index 1 (shifts other elements right)
System.ou t.p rin tln (list); // Output: [1, 50, 2, 3, 4]
// Create a new ArrayList to be added to original list
List<I nte ger> newList = new ArrayL ist();
newLis t.a dd( 150);
```

LinkedList,ArrayList (cont)

```
> newList.add(160);
list.addAll(newList); // Add all elements from newList to list
System.out.println(list); // Output: [1, 50, 2, 3, 4, 150, 160]
System.out.println(list.get(1)); // Output: 50 (element at index 1)
// Create a new ArrayList
List<Integer> list = new ArrayList<>();
list.add(10);
list.add(20);
list.add(30);
list.add(40);
list.add(50);
list.add(60);
list.add(70);
list.add(80);
System.out.println(list); // Output: [10, 20, 30, 40, 50, 60, 70, 80]
// Iterate using traditional for loop
for (int i = 0; i < list.size(); i++) {
    System.out.println("the element is " + list.get(i));
    /* Output:
    the element is 10
    the element is 20
    the element is 30
    the element is 40
    the element is 50
    the element is 60
    the element is 70
    the element is 80 */
}
// Iterate using foreach loop
for (Integer element: list) {
```

LinkedList,ArrayList (cont)

```
>     System.out.println("foreach element is "
+ element);
    /* Output:
    foreach element is 10
    foreach element is 20
    foreach element is 30
    foreach element is 40
    foreach element is 50
    foreach element is 60
    foreach element is 70
    foreach element is 80 */
}
// Iterate using Iterator
Iterator<Integer> it = list.iterator();
while (it.hasNext()) {
    System.out.println("iterator " + it.next());
    /* Output:
    iterator 10
    iterator 20
    iterator 30
    iterator 40
    iterator 50
    iterator 60
    iterator 70
    iterator 80 */
}
list.set(2, 1000); // Replace element at index
2 with 1000
System.out.println(list); // Output: [10, 20,
1000, 40, 50, 60, 70, 80]
System.out.println(list.contains(500)); //
Output: false (500 is not in the list)
list.remove(1); // Remove element at index 1
(20)
System.out.println(list); // Output: [10, 1000,
40, 50, 60, 70, 80]
```

LinkedList,ArrayList (cont)

```
> list.remove(Integer.valueOf(30)); //
Remove element 30 (if exists)
System.out.println(list); // Output: [10, 1000,
40, 50, 60, 70, 80]
list.clear(); // Remove all elements from the
list
System.out.println(list); // Output: []
```