Iterable<T> : from <java.lang>

Methods in Iterable<T> Interface

Method	Description
Iterator<T> iterator()	Returns an Iterator to traverse elements
void forEach(Consumer<? super T> action)	Performs the given action for each element
Spliterator<T> spliterator()	Returns a Spliterator for parallel processing (Java 8)

- **hasNext()** – Checks if the next element exists.
- **next()** – Moves to the next element.

Note :

✓ **Iterable<T> is the root interface** of the Java Collection Framework.

✓ Provides **iterator()** for manual traversal.

✓ Supports **enhanced for-loops (for-each).**

✓ Java 8 **forEach()** method allows functional-style iteration.

✓ You can **make any class Iterable** by implementing iterator().

ASHISH 23

Iterator<I>

Iterator<E> Methods Breakdown

1. boolean hasNext()

- Purpose: Checks if the iteration has more elements.
- Returns: true if there are more elements to iterate, false otherwise.

2. E next()

- Purpose: Returns the next element in the iteration.
- Throws: NoSuchElementException if no more elements exist.
- Important: Always check hasNext() before calling next().

3. default void remove()

- Purpose: Removes the last element returned by the iterator from the underlying collection.
- Optional: Not all collections support this (e.g., List.of(...) is immutable).
- Throws: IllegalStateException if next() hasn't been called or remove() has already been called after the last next().

4. default void forEachRemaining(Consumer<? super E> action)

- Purpose: Performs the given action for each remaining element until all elements have been processed or the action throws an exception.
- Introduced in: Java 8 (as a default method)

ASHISH_23

Ex :

```
public static void main(String[] args) {

        List<Integer> numList = new ArrayList<Integer>();
        numList.add(1);
        numList.add(2);
        numList.add(3);
        numList.add(4);
        numList.add(5);
        Iterator<Integer> iterator = numList.iterator();

        while (iterator.hasNext()) {
             Integer num = iterator.next();
             System.out.println(num);
             if(num == 3 ) {
                  iterator.remove();
             }
        }

        System.out.println(numList);



    }
```

Collection<I> :

🔍 **Core Collection Methods**

Method	Description	Example
int size()	Returns number of elements.	collection.size(); → 5

Method	Description	Example
boolean isEmpty()	Returns true if collection has no elements.	collection.isEmpty();
boolean contains(Object o)	Checks if a specific element exists.	collection.contains("Java");

Iteration

Method	Description	Example
Iterator<E> iterator()	Returns an Iterator to loop through the collection.	Iterator<E> it = collection.iterator();

Conversion

Method	Description	Example
Object[] toArray()	Converts collection to Object array.	Object[] arr = collection.toArray();
<T> T[] toArray(T[] a)	Converts to array of specific type.	String[] arr = collection.toArray(new String[0]);
default <T> T[] toArray(IntFunction<T[]>)	Java 11+. Advanced	String[] arr = list.toArray(String[]::new);

Method	Description	Example
	array creation.	

✚ Modification

Method	Description	Example
boolean add(E e)	Adds an element.	collection.add("Java");
boolean remove(Object o)	Removes specific element.	collection.remove("Java");
boolean addAll(Collection<? extends E> c)	Adds all elements from another collection.	collection.addAll(anotherList);
boolean removeAll(Collection<?> c)	Removes all elements that exist in another collection.	collection.removeAll(toRemove);

Method	Description	Example
boolean retainAll(Collection<?> c)	Keeps only elements that exist in another collection.	collection.retainAll(toKeep);
void clear()	Removes all elements.	collection.clear();

🧠 Predicate-based Removal

Method	Description	Example
default boolean removeIf(Predicate<? super E> filter)	Removes all elements that match the condition.	list.removeIf(e -> e < 10);

🧪 Equality & Hashing

Method	Description	Example
boolean equals(Object o)	Checks if two collections are equal.	list1.equals(list2);
int hashCode()	Returns hash code for the collection.	int hash = collection.hashCode();

Streams and Spliterators (Java 8+)

Method	Description	Example
default Stream<E> stream()	Returns a **sequential stream**.	collection.stream().forEach(...);
default Stream<E> parallelStream()	Returns a **parallel stream**.	collection.parallelStream().forEach(...);
default Spliterator<E> spliterator()	Used for advanced iteration, usually in parallel processing.	Spliterator<E> spliterator = collection.spliterator();

ASH

Collections<C>:

java.util.Collections is a utility class that provides many static methods to operate on or return collections (like List, Set, Map).

Methods in Collections<C>

🔢 1. Sorting & Searching

Method	Description	Example
sort(List<T>)	Sorts in natural ascending order	Collections.sort(list);
sort(List<T>, Comparator)	Sorts with custom logic	Collections.sort(list, Comparator.reverseOrder());
binarySearch(List, key)	Searches a **sorted** list	Collections.binarySearch(list, 10);

🔁 2. Rearranging & Modifying

Method	Description	Example
reverse(List<?>)	Reverses element order	Collections.reverse(list);
shuffle(List<?>)	Randomly shuffles list	Collections.shuffle(list);

Method	Description	Example
swap(List<?>, i, j)	Swaps two elements	Collections.swap(list, 0, 2);
fill(List<? super T>, T)	Replaces all with one value	Collections.fill(list, 0);
replaceAll(List<T>, oldVal, newVal)	Replaces oldVal with newVal	Collections.replaceAll(list, 2, 100);

📊 3. Min, Max, Frequency

Method	Description	Example
min(Collection)	Returns smallest element	Collections.min(list);
max(Collection)	Returns largest element	Collections.max(list);
frequency(Collection, obj)	Counts obj occurrences	Collections.frequency(list, "Java");

🚫 4. Immutability / Unmodifiable Collections

Method	Description	Example
unmodifiableList(List)	Returns unmodifiable list	Collections.unmodifiableList(list);
unmodifiableSet(Set)	Returns unmodifiable set	Collections.unmodifiableSet(set);
unmodifiableMap(Map)	Returns unmodifiable map	Collections.unmodifiableMap(map);

🔐 5. Thread-Safe (Synchronized Collections)

Method	Description	Example
synchronizedList(List)	Thread-safe list	Collections.synchronizedList(list);
synchronizedSet(Set)	Thread-safe set	Collections.synchronizedSet(set);
synchronizedMap(Map)	Thread-safe map	Collections.synchronizedMap(map);

💡 6. Singleton & Constants

Method	Description	Example
singleton(T)	Returns immutable set with 1 element	Collections.singleton("Java");
nCopies(n, T)	Returns immutable list with n copies	Collections.nCopies(5, "A");
emptyList()	Returns immutable empty list	Collections.emptyList();
emptySet()	Returns immutable empty set	Collections.emptySet();
emptyMap()	Returns immutable empty map	Collections.emptyMap();

🔍 7. Utilities

Method	Description	Example
disjoint(c1, c2)	Returns true if no common elements	Collections.disjoint(list1, list2);
frequency(Collection, obj)	Number of times obj appears	Collections.frequency(list, 5);

List<I> :

Important Methods in List<E>

Category	Method	Description
Basic Ops	add(E e)	Adds element to the end of the list.
	add(int index, E element)	Inserts element at a specified position.
	remove(int index)	Removes element at index.
	remove(Object o)	Removes the first occurrence of the object.
	set(int index, E element)	Replaces the element at specified index.
	get(int index)	Retrieves the element at index.
	clear()	Removes all elements.
	size()	Returns number of elements.
Search	contains(Object o)	Checks if element is in the list.
	indexOf(Object o)	Returns first index of element, or -1.
	lastIndexOf(Object o)	Returns last index of element, or -1.

Category	Method	Description
Iteration	iterator()	Returns an Iterator<E>.
	listIterator()	Returns a ListIterator<E>.
	spliterator()	Returns a Spliterator<E> for parallelism.
Sublist	subList(int from, int to)	Returns view of portion of list.
Sorting	sort(Comparator)	Sorts the list using a custom comparator.
	replaceAll(UnaryOperator)	Replaces all elements with the result of applying the operator.
Factory Methods	List.of(...)	Creates immutable list with given elements.
	List.copyOf(Collection)	Copies a collection into an immutable list.

Note :

🆕 Java 21 – Sequenced Methods (From SequencedCollection<I>)

New Method	Description
getFirst()	Returns the first element.
getLast()	Returns the last element.
addFirst(E e)	Adds element at the beginning.
addLast(E e)	Adds element at the end.
removeFirst()	Removes and returns the first element.
removeLast()	Removes and returns the last element.
reversed()	Returns a view of the list in reverse order (not a new list, just a view).

Note:

Unmodifiable List vs Immutable List

Feature	Unmodifiable List	Immutable List
Definition	A wrapper over an existing list that prevents modification	A list that is truly immutable, created as-is, unchangeable

Feature	Unmodifiable List	Immutable List
Backed By Original List?	✅ Yes — changes in the original list reflect in the unmodifiable view	❌ No — it's a completely independent copy
Can underlying data change?	✅ Yes (original list can be modified)	❌ No, never
Modification Methods	Throws UnsupportedOperationException when called	Not available / not supported
Created using	Collections.unmodifiableList(list)	List.of(...) (Java 9+) or libraries like Guava
Mutability of Elements	Elements inside can be mutable	Elements inside can still be mutable unless explicitly frozen
Example	java List<String> list = new ArrayList<>(List.of("A", "B")); List<String> unmod = Collections.unmodifiableList(list);	java List<String> immut = List.of("A", "B");

Feature	Unmodifiable List	Immutable List
Behavior if Original Changes	Unmodifiable list reflects those changes	Immutable list remains unchanged

Note:

Use Case	**Prefer**
You want to expose a read-only view of a modifiable list	unmodifiableList()
You want to guarantee immutability (thread-safety, no accidental changes)	List.of() or other immutable implementations

Note: Where to use immutableList and unmodifiableList

Need	Use
Read-only view of a modifiable list	Collections.unmodifiableList()
Fixed-size, fully immutable list	List.of(...) (Java 9+)
Share across threads without mutation	List.of(...)
Prevent API clients from altering data	Either (depending on context)
Still want to update original list	unmodifiableList, not immutable

ArrayList<C>:

🏗️ Constructors

Constructor	Description
ArrayList()	Creates an empty list with default capacity.
ArrayList(int capacity)	Creates a list with given initial capacity.
ArrayList(Collection<? extends E> c)	Creates a list containing elements from another collection.

📌 Important Methods (Grouped)

🔢 Size & Access

Method	Description
size()	Number of elements.
isEmpty()	Checks if list is empty.
get(int index)	Get element at index.
set(int index, E element)	Replace element at index.

➕ Adding Elements

Method	Description
add(E element)	Appends to end.
add(int index, E element)	Inserts at index.
addFirst(E) / addLast(E)	Java 21+: Adds to front or back.
addAll(Collection)	Adds all elements of another collection.

➖ Removing Elements

Method	Description
remove(Object o)	Removes first matching element.
remove(int index)	Removes element at index.
removeFirst() / removeLast()	Java 21+: Removes from ends.
removeAll(Collection)	Removes all in the collection.
retainAll(Collection)	Keeps only elements in the collection.
clear()	Empties the list.

🔍 Searching

Method	Description
contains(Object)	Checks if element exists.

Method	Description
indexOf(Object)	Index of first match.
lastIndexOf(Object)	Index of last match.

🛠️ Utilities

Method	Description
clone()	Creates a shallow copy of the list.
toArray()	Returns array of all elements.
replaceAll(UnaryOperator)	Replace each element with result of function.
sort(Comparator)	Sorts the list with custom logic.
forEach(Consumer)	Runs action for each element.
removeIf(Predicate)	Removes elements satisfying condition.

🔁 Iterators

Method	Description
iterator()	Returns Iterator<E>.
listIterator()	Returns ListIterator<E>.
spliterator()	For parallel processing (Streams).

Sublist & Internal

Method	Description
subList(int from, int to)	Returns view between indices.
removeRange(int from, int to)	Protected; used internally.
checkInvariants()	Internal validation method.

LinkedList<C>

Note :

public class java.util.LinkedList<E> extends java.util.AbstractSequentialList<E> implements java.util.List<E>, java.util.Deque<E>, java.lang.Cloneable, java.io.Serializable

Now we need to Understand about the Deque<I> :

Note :

public interface java.util.Deque<E> extends java.util.Queue<E>, java.util.SequencedCollection<E> {

ASHISH_23

Now got o Queue<I> :

```
public interface java.util.Queue<E> extends java.util.Collection<E> {
  public abstract boolean add(E);
  public abstract boolean offer(E);
  public abstract E remove();
  public abstract E poll();
  public abstract E element();
  public abstract E peek();
}
```

✅ Queue<E> Interface Overview

The Queue interface is used when you want to process elements in the order they were added, like in:

- **Print queues**
- **Task schedulers**
- **BFS algorithms (trees/graphs)**
- **Message processing systems**

Methods In Queue<I>:

ASHISH_23

Method	Throws Exception	Returns Special Value	Description
add(E e)	✅ Yes (IllegalStateException)	❌	Inserts element if possible (capacity-bounded queues may throw error)
offer(E e)	❌ No	✅ false if it can't add	Adds element safely, returns false if full (non-blocking)
remove()	✅ Yes (NoSuchElementException)	❌	Retrieves & removes head; error if empty
poll()	❌ No	✅ null if empty	Retrieves & removes head, safely
element()	✅ Yes (NoSuchElementException)	❌	Retrieves (but doesn't remove) head; error if empty
peek()	❌ No	✅ null if empty	Retrieves (but doesn't remove) head, safely

🆚 **Exception vs. Null-based Methods**

Operation	Unsafe (throws)	Safe (returns null/false)
Add	add()	offer()
Remove	remove()	poll()
Peek	element()	peek()

🔔 **Best Practice: In production, prefer offer, poll, and peek to avoid unhandled exceptions.**

Deque<I> :

Methods :

🧠 **What is Deque?**

Deque stands for Double-Ended Queue. It supports:

- **FIFO (like regular Queue): insert at tail, remove from head.**
- **LIFO (like Stack): insert and remove at head**

HISH_23

Methods :

✅ **Insertion**

Method	Description
addFirst(e)	Inserts at the front (head)
addLast(e)	Inserts at the back (tail)
offerFirst(e)	Same as addFirst, but safe
offerLast(e)	Same as addLast, but safe
push(e)	Stack-style add (same as addFirst)

Method	Description
add(e)	Adds at tail (from Queue)
offer(e)	Safe tail insertion

❌ Removal

Method	Description
removeFirst()	Removes from front, throws if empty
removeLast()	Removes from end, throws if empty
pollFirst()	Removes from front, returns null if empty
pollLast()	Removes from end, returns null if empty
pop()	Stack-style pop (same as removeFirst)
remove()	Removes from front (Queue style)
remove(Object o)	Removes first match
removeFirstOccurrence(o)	Removes first matching element
removeLastOccurrence(o)	Removes last matching element

👁️ Access (Peek)

Method	Description
getFirst()	Returns first element, throws if empty
getLast()	Returns last element, throws if empty
peekFirst()	Returns first element or null if empty
peekLast()	Returns last element or null if empty
element()	Like getFirst()
peek()	Like peekFirst()

🔄 Other Utilities

Method	Description
reversed()	Returns a reversed view of the deque *(Java 21+)*
descendingIterator()	Returns reverse iterator
iterator()	Normal forward iterator
contains(Object)	Checks if an element exists
size()	Returns number of elements
addAll(Collection)	Adds all elements from a collection

💡 Common Implementations of Deque

Class	Description
ArrayDeque	Resizable array-based Deque (fastest, most used)
LinkedList	Linked node implementation of Deque
ConcurrentLinkedDeque	Thread-safe concurrent deque

SHISH_2

LinkedList<C> :

🧠 **Key Features**

Feature	Description
Efficient insert/delete	Especially at head/tail (O(1) time)
Slower random access	O(n) vs O(1) in ArrayList
Doubly-linked	Can traverse forwards and backwards

✅ 1. Deque Methods (Head/Tail Access)

These are inherited from Deque (Java 6+):

Method	Description
addFirst(E)	Inserts element at the front
addLast(E)	Inserts element at the end
getFirst()	Gets element at front (throws if empty)
getLast()	Gets element at end (throws if empty)
removeFirst()	Removes from front (throws if empty)
removeLast()	Removes from end (throws if empty)
offerFirst(E)	Inserts at front (returns false if fails)
offerLast(E)	Inserts at end (returns false if fails)
peekFirst()	Returns front element or null
peekLast()	Returns last element or null
pollFirst()	Removes and returns front or null
pollLast()	Removes and returns last or null
removeFirstOccurrence(Object)	Removes first matching element
removeLastOccurrence(Object)	Removes last matching element

✅ 2. Stack-like Methods (LIFO)

These are often forgotten:

Method	Description
push(E)	Same as addFirst()
pop()	Same as removeFirst()

✅ 3. Reverse Access – Java 21+

These are new in Java 21 and come from SequencedCollection and Deque:

Method	Description
reversed()	Returns a reversed view of the list
SequencedCollection.reversed()	Interface method
Deque.reversed()	Reversed deque order access (view)

⚠️ **Reversed views are read/write backed by original list but in reverse order (Java 21+ only).**

✅ 4. Utility Methods (Less Common)

Method	Description
descendingIterator()	Iterator from tail to head

ADL 23

Method	Description
listIterator()	Bi-directional list iterator
clone()	Shallow copy
toArray()	Convert to Object[] or typed array
spliterator()	Splits list for parallelism
reversed()	Java 21+, returns reversed view

🧠 Summary

Category	Methods	Java Version
Double-ended ops	addFirst, removeLast, peekFirst, etc.	Java 6+
Stack ops	push, pop	Java 6+
Reversed views	reversed()	✅ Java 21+
Iterators	descendingIterator()	Java 6+
Cloning	clone()	Java 1.2+

📚 List/Index Support

Method	Description
get(int)	Gets element at index
set(int, E)	Replaces element at index
add(int, E)	Inserts at index
remove(int)	Removes at index
indexOf(Object)	Returns first index of element
lastIndexOf(Object)	Last index of element

Note :

✅ Comparison Table: add(), offer(), push(), addFirst(), addLast()

Method	Inserts at	Throws Exception on Failure	Returns	Use Case
add(E e)	Tail	✅ Yes (IllegalStateException)	true/Exception	General-purpose List/Queue insertion
offer(E e)	Tail	❌ No	true/false	Preferred for Queue, returns safely

ASHISH123

Method	Inserts at	Throws Exception on Failure	Returns	Use Case
addFirst(E e)	Head	✅ Yes	void	Use with Deque to add at front
offerFirst(E e)	Head	❌ No	true/false	Safe version of addFirst()
addLast(E e)	Tail	✅ Yes	void	Use with Deque to add at end
offerLast(E e)	Tail	❌ No	true/false	Safe version of addLast()
push(E e)	Head	✅ Yes	void	Stack-style insertion (LIFO)

🧠 Summary by Use Case

Scenario	Best Method	Notes
Simple append to end	add() or offer()	offer() is safer (no exception)
Add to beginning	addFirst() / offerFirst()	For Deque usage
Add to end explicitly	addLast() / offerLast()	For Deque usage
Stack (LIFO) behavior	push()	Same as addFirst()
Safe (non-throwing) insert	offer() variants	Better in concurrent/limited-capacity contexts

- Use offer*() methods when you want non-exception behavior (returns false if fails).
- Use addFirst()/addLast() if you want explicit control over ends.
- Use push() if you want stack-like behavior with a Deque.

LinkedList vs ArrayList

Feature	ArrayList	LinkedList
Access time	Fast (O(1))	Slow (O(n))
Insert/delete	Slow (O(n))	Fast (O(1) with node)
Memory	Compact	Overhead per node
Queue/Stack	Not ideal	Very efficient
Random access	Excellent	Poor

Real-World Spring Boot Use Case Comparison

Use Case	Recommended	Why
Sending list of entities from service/controller	ArrayList	Fast, compact, good for read-heavy

Use Case	Recommended	Why
Maintaining a background job queue (e.g., email/SMS)	**LinkedList or ArrayDeque**	**Efficient head/tail ops**
Managing recent search/view history (like a stack)	**LinkedList or Deque**	**Supports push/pop**
Processing ordered batch updates to DB	**ArrayList**	**Order + size known ahead**
Implementing undo/redo logic	**LinkedList**	**Stack-like LIFO operations**
Streaming data records	**LinkedList (if buffer)**	**Head-based processing**

Note :

- In a microservices architecture where JSON payloads matter, ArrayList is more serialization-friendly (due to linear layout).
- Use LinkedList only when you clearly benefit from frequent adds/removes at both ends.

Vector<C>:

Vector is a legacy synchronized list implementation introduced in Java 1.0. It is similar to ArrayList, but all its methods are synchronized, which makes it thread-safe but also slower in single-threaded applications.

ASHIS

Characteristics

Feature	Details
Type	Resizable array (like ArrayList)
Thread-safe	✅ Yes, all methods synchronized
Grow strategy	Doubles capacity by default (unless specified)
Performance	Slower due to synchronization overhead
Use today?	❌ Not recommended unless maintaining legacy code

✅ **Important Fields**

Field	Description
elementData	The backing object array
elementCount	Number of elements in vector
capacityIncrement	Optional value to increase capacity manually

1. Constructors

Vector() // Default (capacity 10)

Vector(int initialCapacity)

Vector(int initialCapacity, int capacityIncrement)

Vector(Collection<? extends E> c)

2. Basic Methods

Method	Description
size()	Returns current size
capacity()	Returns current capacity
trimToSize()	Shrinks capacity to match size
ensureCapacity(int)	Grows capacity if needed
setSize(int)	Resizes list (nulls inserted if increased)

3. Element Access

Method	Description
add(E e)	Adds to end
add(int, E)	Adds at index
get(int)	Gets element at index
set(int, E)	Replaces at index
remove(int)	Removes by index
remove(Object)	Removes first matching element

Method	Description
clear()	Empties the vector

4. Legacy Methods (Not in List interface — unique to Vector)

Method	Description
addElement(E)	Legacy version of add()
removeElement(Object)	Legacy version of remove()
insertElementAt(E, int)	Insert at index
setElementAt(E, int)	Set at index
elementAt(int)	Get at index
firstElement()	Get first element
lastElement()	Get last element
removeElementAt(int)	Remove at index
removeAllElements()	Clears vector
copyInto(Object[] a)	Copy contents to array (manual copy)

5. Iteration / Splitting

Method	Description
iterator()	Standard iterator
listIterator()	Bi-directional iterator
elements()	Legacy Enumeration object
spliterator()	For parallel stream

---

6. Sorting / Replacing

Method	Description
sort(Comparator)	Java 8+ sorting
replaceAll(UnaryOp)	Replaces each element
removeIf(Predicate)	Removes conditionally

⚠️ Thread Safety

All methods like add(), remove(), get() are **synchronized**, which makes Vector thread-safe — but also slower. Prefer Collections.synchronizedList(new ArrayList<>()) or CopyOnWriteArrayList in new codebases.

⚖️ Vector vs ArrayList vs CopyOnWriteArrayList

ASHISH_23

Feature	Vector	ArrayList	CopyOnWriteArrayList
Thread-safe	✅ Yes (synchronized)	❌ No	✅ Yes (lock-free reads)
Performance	❌ Slower	✅ Fast	❌ Slower on writes
Legacy/Modern	❌ Legacy	✅ Modern	✅ Modern (concurrent)
Iterator safety	❌ Fail-fast	❌ Fail-fast	✅ No ConcurrentModificationException

Note :

Don’t prefer Vector for Realtime projects (only for java 1.1 v projects)

- ArrayList in single-threaded scenarios.
- CopyOnWriteArrayList or ConcurrentLinkedDeque for concurrency.

Stack<C>:

public class java.util.Stack<E> extends java.util.Vector<E> {

public java.util.Stack();

public E push(E);

public synchronized E pop();

ASHISH

```
  public synchronized E peek();

  public boolean empty();

  public synchronized int search(java.lang.Object);
}
```

📦 What is Stack<E>?

Stack is a class introduced in **Java 1.0** that extends Vector. It represents a **LIFO (Last In, First Out)** stack of objects — useful for things like undo/redo operations, parsing, or expression evaluation.

🔴 **Important**: Stack is legacy. In modern Java, prefer Deque (like ArrayDeque) for stack behavior.

SHISH_23

✅ Important Stack Methods

Method	Description
push(E item)	Adds item to the top of the stack
pop()	Removes and returns the top item (throws EmptyStackException if empty)
peek()	Returns top item without removing it

Method	Description
empty()	Returns true if the stack is empty
search(Object o)	Returns 1-based position from top of the stack; -1 if not found

Comparison: Stack vs ArrayDeque

Feature	Stack	ArrayDeque
Thread-safe	✅ Yes (synchronized)	❌ No
Performance	❌ Slower	✅ Faster
Introduced	Java 1.0	Java 6
Preferred now	❌ No	✅ Yes

🧠 When to Use Stack?

Use Case	Stack	Deque (Preferred)
Parsing expressions	✅	✅
Undo/Redo	✅	✅
Multi-threading	✅ (rare)	❌ use ConcurrentLinkedDeque

Use Case	Stack	Deque (Preferred)
Production	❌ avoid	✅

✅ Real-Time Use Cases of Stack

1. Undo/Redo functionality (e.g., form editing, CMS changes)
2. Browser navigation history
3. Backtracking algorithms (e.g., pathfinding, maze solving)
4. Expression evaluation (e.g., infix to postfix conversion)
5. Balanced parentheses or tag checking (HTML, code parsing)
6. Recursive method simulation (custom call stack)
7. Tree/Graph traversal (Depth First Search)
8. Transactional rollback management (manual undo of state)
9. Stack-based state management in workflows
10. Tracking nested operations (e.g., request logging stack)
11. Function call tracking in interpreters or DSLs
12. Text editor undo stack
13. Bracket parsing in compilers
14. History tracking in Spring MVC web applications
15. Navigation stack in mobile/web apps (menu tabs, breadcrumbs)

ASHISH_23