Java Core + Collections + Built-in Methods Cheatsheet

📚 Part 1: Built-in Methods by Java Version

☕ Java 8 Key Methods

Stream API

`stream()` **- Collection<E>.stream()**

- **Java Version:** 8
- **Purpose:** Creates a sequential stream from a collection
- **Example:**

```
List<String> list = Arrays.asList("a", "b", "c");
Stream<String> stream = list.stream();
```

- **Performance Note:** Creates internal spliterator; avoid creating streams for simple operations

`filter()` **- Stream<T>.filter(Predicate<? super T> predicate)**

- **Java Version:** 8
- **Purpose:** Filter elements based on a condition
- **Example:**

```
List<String> filtered = names.stream()
    .filter(s → s.length() > 3)
    .collect(Collectors.toList());
```

- **Performance Note:** Lazy evaluation; only processes elements when terminal operation is called

`map()` **- Stream<T>.map(Function<? super T, ? extends R> mapper)**

- **Java Version:** 8
- **Purpose:** Transform each element using a function
- **Example:**

```
List<Integer> lengths = names.stream()
    .map(String::length)
    .collect(Collectors.toList());
```

- **Performance Note:** Creates a new object for each element; be cautious with large streams

`flatMap()` **- Stream<T>.flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)**

- **Java Version:** 8
- **Purpose:** Transform and flatten nested streams
- **Example:**

```
List<String> words = sentences.stream()
    .flatMap(s → Arrays.stream(s.split(" ")))
    .collect(Collectors.toList());
```

- **Performance Note:** Useful for handling nested collections; can be performance-intensive

`collect()` **- Stream<T>.collect(Collector<? super T, A, R> collector)**

- **Java Version:** 8
- **Purpose:** Accumulate elements into a collection
- **Example:**

```
List<String> collected = stream.collect(Collectors.toList());
```

- **Performance Note:** Terminal operation; consider specialized collectors for better performance

`reduce()` **- Stream<T>.reduce(BinaryOperator<T> accumulator)**

- **Java Version:** 8
- **Purpose:** Combine stream elements into a single result
- **Example:**

```
Optional<Integer> sum = numbers.stream().reduce(Integer::sum);
```

- **Performance Note:** Identity-based overload available; consider using specialized methods (sum, max) when possible

Optional API

`of()` - Optional.of(T value)

- **Java Version:** 8
- **Purpose:** Create Optional with non-null value
- **Example:**

```
Optional<String> opt = Optional.of("value");
```

- **Performance Note:** Throws NullPointerException if value is null; use ofNullable() when unsure

`ofNullable()` - Optional.ofNullable(T value)

- **Java Version:** 8
- **Purpose:** Create Optional that may contain null
- **Example:**

```
Optional<String> opt = Optional.ofNullable(possiblyNull);
```

- **Performance Note:** Returns empty Optional if value is null

`orElse()` - Optional<T>.orElse(T other)

- **Java Version:** 8
- **Purpose:** Get value or default if empty
- **Example:**

```
String value = opt.orElse("default");
```

- **Performance Note:** Always evaluates the default value, even when Optional is not empty

`orElseGet()` - Optional<T>.orElseGet(Supplier<? extends T> supplier)

- **Java Version:** 8
- **Purpose:** Get value or compute default if empty
- **Example:**

```
String value = opt.orElseGet(() → computeDefault());
```

- **Performance Note:** Lazy evaluation; supplier only called when Optional is empty

`map()` **- Optional<T>.map(Function<? super T, ? extends U> mapper)**

- **Java Version:** 8
- **Purpose:** Transform value if present
- **Example:**

```
Optional<Integer> length = opt.map(String::length);
```

- **Performance Note:** Returns empty Optional if original is empty

`flatMap()` **- Optional<T>.flatMap(Function<? super T, Optional<U>> mapper)**

- **Java Version:** 8
- **Purpose:** Transform to another Optional
- **Example:**

```
Optional<User> user = getUserById(id).flatMap(this::getAddressOpt);
```

- **Performance Note:** Prevents nested Optionals; returns empty if original is empty

Map Interface

`forEach()` **- Map<K,V>.forEach(BiConsumer<? super K, ? super V> action)**

- **Java Version:** 8
- **Purpose:** Iterate through map entries
- **Example:**

```
map.forEach((k, v) → System.out.println(k + "=" + v));
```

- **Performance Note:** Not guaranteed to process in any particular order

`getOrDefault()` **- Map<K,V>.getOrDefault(Object key, V defaultValue)**

- **Java Version:** 8
- **Purpose:** Get value or default if key not present
- **Example:**

```
String value = map.getOrDefault("key", "default");
```

- **Performance Note:** Default is returned if key is not found or mapped to null

`computeIfAbsent()` **- Map<K,V>.computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction)**

- **Java Version:** 8
- **Purpose:** Compute value if key not present
- **Example:**

```
map.computeIfAbsent("key", k → expensiveOperation());
```

- **Performance Note:** Atomic put-if-absent functionality; great for initialization and caching

`computeIfPresent()` **- Map<K,V>.computeIfPresent(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction)**

- **Java Version:** 8
- **Purpose:** Compute new value if key present
- **Example:**

```
map.computeIfPresent("key", (k, v) → v + 1);
```

- **Performance Note:** Returns null to remove entry; atomic

`merge()` **- Map<K,V>.merge(K key, V value, BiFunction<? super V, ? super V, ? extends V> remappingFunction)**

- **Java Version:** 8
- **Purpose:** Merge value with existing or insert
- **Example:**

```
map.merge("key", 1, Integer::sum);
```

- **Performance Note:** Perfect for counters and aggregations; null result removes entry

Date/Time API

`now()` **- LocalDate.now()**

- **Java Version:** 8
- **Purpose:** Get current date
- **Example:**

```
LocalDate today = LocalDate.now();
```

- **Performance Note:** System clock dependent; use Clock for testing

`of()` **- LocalDate.of(int year, int month, int day)**

- **Java Version:** 8
- **Purpose:** Create date from components
- **Example:**

```
LocalDate date = LocalDate.of(2023, 3, 15);
```

- **Performance Note:** Month is 1-based, unlike legacy Date APIs

`parse()` **- LocalDate.parse(CharSequence text)**

- **Java Version:** 8
- **Purpose:** Parse date from string
- **Example:**

```
LocalDate date = LocalDate.parse("2023-03-15");
```

- **Performance Note:** Throws DateTimeParseException for invalid formats

`plus()` **- LocalDate.plus(TemporalAmount amount)**

- **Java Version:** 8
- **Purpose:** Add time unit to date
- **Example:**

```
LocalDate nextWeek = today.plus(1, ChronoUnit.WEEKS);
```

- **Performance Note:** Immutable; returns new instance

`format()` **- LocalDate.format(DateTimeFormatter formatter)**

- **Java Version:** 8
- **Purpose:** Format date to string

- **Example:**

```
String formatted = date.format(DateTimeFormatter.ofPattern("dd/MM/yyyy"));
```

- **Performance Note:** Use predefined formatters when possible

CompletableFuture

`supplyAsync()` **- CompletableFuture.supplyAsync(Supplier<U> supplier)**

- **Java Version:** 8
- **Purpose:** Run task asynchronously with return value
- **Example:**

```
CompletableFuture<String> cf = CompletableFuture.supplyAsync(() → fetchData
());
```

- **Performance Note:** Uses common ForkJoinPool by default; specify executor for control

`thenApply()` **- CompletableFuture<T>.thenApply(Function<? super T, ? extends U> fn)**

- **Java Version:** 8
- **Purpose:** Transform result when available
- **Example:**

```
CompletableFuture<Integer> length = cf.thenApply(String::length);
```

- **Performance Note:** Executes in calling thread if result already available

`thenCombine()` **- CompletableFuture<T>.thenCombine(CompletableFuture<? extends U> other, BiFunction<? super T, ? super U, ? extends V> fn)**

- **Java Version:** 8
- **Purpose:** Combine two futures when both complete
- **Example:**

```
future1.thenCombine(future2, (r1, r2) → r1 + r2);
```

- **Performance Note:** Parallel execution; completes when both inputs complete `exceptionally()` **- CompletableFuture<T>.exceptionally(Function<Throwable, ? extends T> fn)**

- **Java Version:** 8
- **Purpose:** Handle exceptions
- **Example:**

```
cf.exceptionally(ex → "Error: " + ex.getMessage());
```

- **Performance Note:** Executes only if previous stage completes exceptionally

☕ Java 9-16 Key Methods

Collection Factory Methods

`of()` **- List.of(E... elements)**

- **Java Version:** 9
- **Purpose:** Create immutable list
- **Example:**

```
List<String> list = List.of("a", "b", "c");
```

- **Performance Note:** Immutable; throws NPE if any element is null

`of()` **- Set.of(E... elements)**

- **Java Version:** 9
- **Purpose:** Create immutable set
- **Example:**

```
Set<String> set = Set.of("a", "b", "c");
```

- **Performance Note:** Immutable; throws NPE if any element is null or duplicate

`of()` **- Map.of(K k1, V v1, K k2, V v2...)**

- **Java Version:** 9
- **Purpose:** Create immutable map (up to 10 entries)
- **Example:**

```
Map<String, Integer> map = Map.of("a", 1, "b", 2);
```

- **Performance Note:** Limited to 10 entries; use ofEntries for more

`ofEntries()` **- Map.ofEntries(Map.Entry<? extends K, ? extends V>... entries)**

- **Java Version:** 9
- **Purpose:** Create immutable map with arbitrary entries
- **Example:**

```
Map<String, Integer> map = Map.ofEntries(Map.entry("a", 1), Map.entry("b", 2));
```

- **Performance Note:** Immutable; no size limitation

`copyOf()` **- List.copyOf(Collection<? extends E> coll)**

- **Java Version:** 10
- **Purpose:** Create immutable copy of collection
- **Example:**

```
List<String> immutableCopy = List.copyOf(originalList);
```

- **Performance Note:** Returns original if already immutable; detects at runtime

String Methods

`strip()` **- String.strip()**

- **Java Version:** 11
- **Purpose:** Remove leading/trailing whitespace
- **Example:**

```
String trimmed = " text ".strip();
```

- **Performance Note:** Unicode-aware, unlike trim()

`stripLeading()` **- String.stripLeading()**

- **Java Version:** 11
- **Purpose:** Remove leading whitespace
- **Example:**

```
String trimmed = " text".stripLeading();
```

- **Performance Note:** Unicode-aware

`stripTrailing()` **- String.stripTrailing()**

- **Java Version:** 11
- **Purpose:** Remove trailing whitespace
- **Example:**

```
String trimmed = "text ".stripTrailing();
```

- **Performance Note:** Unicode-aware

`isBlank()` **- String.isBlank()**

- **Java Version:** 11
- **Purpose:** Check if string is empty or only whitespace
- **Example:**

```
boolean isEmpty = "   ".isBlank();
```

- **Performance Note:** More useful than checking isEmpty()

`lines()` **- String.lines()**

- **Java Version:** 11
- **Purpose:** Split string into stream of lines
- **Example:**

```
Stream<String> lines = text.lines();
```

- **Performance Note:** Handles different line separators automatically

`repeat()` **- String.repeat(int count)**

- **Java Version:** 11
- **Purpose:** Repeat string n times
- **Example:**

```
String repeated = "abc".repeat(3);
```

- **Performance Note:** Throws IllegalArgumentException if count is negative

File Methods

`readString()` **- Files.readString(Path path)**

- **Java Version:** 11
- **Purpose:** Read file contents as string
- **Example:**

```
String content = Files.readString(Path.of("file.txt"));
```

- **Performance Note:** Uses UTF-8 by default; closes resources automatically

`writeString()` **- Files.writeString(Path path, CharSequence csq)**

- **Java Version:** 11
- **Purpose:** Write string to file
- **Example:**

```
Files.writeString(Path.of("file.txt"), "content");
```

- **Performance Note:** Creates file if doesn't exist; overwrites if exists

Stream Methods

`takeWhile()` **- Stream<T>.takeWhile(Predicate<? super T> predicate)**

- **Java Version:** 9
- **Purpose:** Take elements while condition is true
- **Example:**

```
Stream.of(1, 2, 3, 4, 2).takeWhile(n → n < 3);
```

- **Performance Note:** Short-circuiting; stops at first false predicate

`dropWhile()` **- Stream<T>.dropWhile(Predicate<? super T> predicate)**

- **Java Version:** 9
- **Purpose:** Skip elements while condition is true
- **Example:**

```
Stream.of(1, 2, 3, 4, 2).dropWhile(n → n < 3);
```

- **Performance Note:** Keeps all elements after first false predicate

`ofNullable()` **- Stream.ofNullable(T t)**

- **Java Version:** 9
- **Purpose:** Create stream of 0 or 1 elements
- **Example:**

```
Stream<String> stream = Stream.ofNullable(possiblyNull);
```

- **Performance Note:** Creates empty stream if input is null

`toList()` **- Stream<T>.toList()**

- **Java Version:** 16
- **Purpose:** Collect stream elements to list
- **Example:**

```
List<String> list = stream.toList();
```

- **Performance Note:** More concise than collect(Collectors.toList())

Optional Methods

`isEmpty()` **- Optional<T>.isEmpty()**

- **Java Version:** 11
- **Purpose:** Check if optional is empty
- **Example:**

```
boolean empty = opt.isEmpty();
```

- **Performance Note:** Complement of isPresent(); more readable in some cases

`orElseThrow()` **- Optional<T>.orElseThrow()**

- **Java Version:** 10
- **Purpose:** Get value or throw NoSuchElementException
- **Example:**

```
String value = opt.orElseThrow();
```

- **Performance Note:** Simpler than orElseThrow(NoSuchElementException::new)

`stream()` **- Optional<T>.stream()**

- **Java Version:** 9
- **Purpose:** Convert to stream with 0 or 1 element
- **Example:**

```
Stream<String> stream = opt.stream();
```

- **Performance Note:** Useful for flatmapping with other streams

☕ Java 17 Key Methods

Sealed Classes

`getPermittedSubclasses()` **- Class<?>.getPermittedSubclasses()**

- **Java Version:** 17
- **Purpose:** Get permitted subclasses of sealed class
- **Example:**

```
Class<?>[] subclasses = MyClass.class.getPermittedSubclasses();
```

- **Performance Note:** Runtime reflection support for sealed classes

Pattern Matching

`instanceof` **with pattern variable**

- **Java Version:** 16/17
- **Purpose:** Type check and cast in one step
- **Example:**

```
if (obj instanceof String s) {
    System.out.println(s.length());
}
```

- **Performance Note:** Eliminates explicit casting; scope limited to true branch

Records

`isRecord()` **- Class<?>.isRecord()**

- **Java Version:** 16/17
- **Purpose:** Check if class is a record
- **Example:**

```
boolean isRecord = MyClass.class.isRecord();
```

- **Performance Note:** Runtime reflection support for records

`getRecordComponents()` **- Class<?>.getRecordComponents()**

- **Java Version:** 16/17
- **Purpose:** Get components of record
- **Example:**

```
RecordComponent[] components = MyRecord.class.getRecordComponents();
```

- **Performance Note:** Useful for frameworks and serialization

Random Numbers

`random()` **- RandomGenerator.getDefault().nextInt(int bound)**

- **Java Version:** 17
- **Purpose:** Get random number with improved API
- **Example:**

```
int random = RandomGenerator.getDefault().nextInt(100);
```

- **Performance Note:** More predictable algorithm selection than legacy Random

☕ Java 21 Key Methods

Virtual Threads

`startVirtualThread()` **- Thread.startVirtualThread(Runnable task)**

- **Java Version:** 21
- **Purpose:** Start a lightweight virtual thread
- **Example:**

```
Thread vt = Thread.startVirtualThread(() → System.out.println("Virtual thread"));
```

- **Performance Note:** Efficient for IO-bound tasks; not for CPU-intensive work

`ofVirtual()` **- Thread.Builder.ofVirtual()**

- **Java Version:** 21
- **Purpose:** Create virtual thread builder
- **Example:**

```
Thread.Builder builder = Thread.ofVirtual().name("worker-", 0);
```

- **Performance Note:** Configure thread name, daemon status, etc.

`newVirtualThreadPerTaskExecutor()` **- Executors.newVirtualThreadPerTaskExecutor()**

- **Java Version:** 21
- **Purpose:** Create executor using virtual threads
- **Example:**

```
ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor();
```

- **Performance Note:** Ideal for server applications with many concurrent connections

Structured Concurrency

`StructuredTaskScope` **- try (var scope = new StructuredTaskScope.ShutdownOnFailure())**

- **Java Version:** 21
- **Purpose:** Manage group of concurrent tasks
- **Example:**

```
try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
    var f1 = scope.fork(() → task1());
    var f2 = scope.fork(() → task2());
    scope.join();
}
```

- **Performance Note:** Ensures all subtasks complete before parent proceeds; improves error handling

Pattern Matching for Switch

Switch expressions with patterns

- **Java Version:** 21
- **Purpose:** Switch based on type and pattern
- **Example:**

```
String result = switch(obj) {
    case Integer i → "Int: " + i;
    case String s → "String: " + s;
    default → "Unknown";
};
```

- **Performance Note:** Exhaustiveness checking; safer than instanceof chains

Records Patterns

Record patterns in instanceof

- **Java Version:** 21
- **Purpose:** Deconstruct records with pattern matching
- **Example:**

```
if (obj instanceof Point(int x, int y)) {
    System.out.println(x + ", " + y);
}
```

- **Performance Note:** Combines type check and field extraction

String Templates

`STR` **- StringTemplate.STR."Text \{expression}"**

- **Java Version:** 21
- **Purpose:** Create formatted strings with expressions
- **Example:**

```
String message = STR."Hello, \\{name}! Today is \\{day}.";
```

- **Performance Note:** More readable than string concatenation or format

Sequenced Collections

`getFirst()` **- SequencedCollection<E>.getFirst()**

- **Java Version:** 21
- **Purpose:** Get first element of a sequenced collection
- **Example:**

```
E first = sequencedCollection.getFirst();
```

- **Performance Note:** Unified API across List, Deque, etc.

`getLast()` **- SequencedCollection<E>.getLast()**

- **Java Version:** 21
- **Purpose:** Get last element of a sequenced collection
- **Example:**

```
E last = sequencedCollection.getLast();
```

- **Performance Note:** Unified API across List, Deque, etc.

📚 Part 2: Built-in Methods by Data Type

📝 String Methods

`length()`

- **Description:** Returns string length
- **Example:** `int len = str.length();`
- **When to Use:** Basic operation; always O(1)

`charAt(int index)`

- **Description:** Returns char at position
- **Example:** `char c = str.charAt(5);`
- **When to Use:** Direct access; throws IndexOutOfBoundsException if invalid

`substring(int beginIndex, int endIndex)`

- **Description:** Returns substring
- **Example:** `String sub = str.substring(2, 5);`

- **When to Use:** Creates new string; avoid in tight loops

`indexOf(String str)`

- **Description:** Finds first occurrence
- **Example:** `int pos = str.indexOf("test");`
- **When to Use:** Returns -1 if not found; overloads available

`lastIndexOf(String str)`

- **Description:** Finds last occurrence
- **Example:** `int pos = str.lastIndexOf("test");`
- **When to Use:** Returns -1 if not found; useful for file extensions

`startsWith(String prefix)`

- **Description:** Checks if string starts with prefix
- **Example:** `boolean starts = str.startsWith("http");`
- **When to Use:** Faster than regex for simple prefix checking

`endsWith(String suffix)`

- **Description:** Checks if string ends with suffix
- **Example:** `boolean ends = str.endsWith(".java");`
- **When to Use:** Faster than regex for simple suffix checking

`contains(CharSequence s)`

- **Description:** Checks if string contains sequence
- **Example:** `boolean has = str.contains("key");`
- **When to Use:** Simple containment check; use indexOf for position

`trim()`

- **Description:** Removes leading/trailing whitespace
- **Example:** `String trimmed = str.trim();`
- **When to Use:** Only handles ASCII whitespace; use strip() in Java 11+

`replace(char oldChar, char newChar)`

- **Description:** Replaces all occurrences
- **Example:** `String replaced = str.replace('a', 'b');`
- **When to Use:** Creates new string; use StringBuilder for multiple replacements

`replaceAll(String regex, String replacement)`

- **Description:** Replaces by regex

- **Example:** `String fixed = str.replaceAll("\\\\s+", " ");`
- **When to Use:** Powerful but slower than non-regex methods

`split(String regex)`

- **Description:** Splits string by regex
- **Example:** `String[] parts = str.split(",");`
- **When to Use:** Returns array; can specify limit; empty trailing strings discarded by default

`toLowerCase()`

- **Description:** Converts to lowercase
- **Example:** `String lower = str.toLowerCase();`
- **When to Use:** Locale-sensitive; specify Locale for consistent behavior

`toUpperCase()`

- **Description:** Converts to uppercase
- **Example:** `String upper = str.toUpperCase();`
- **When to Use:** Locale-sensitive; specify Locale for consistent behavior

`matches(String regex)`

- **Description:** Checks if entire string matches regex
- **Example:** `boolean isEmail = str.matches("^[\\\\w.-]+@[\\\\w.-]+\\\\.[a-z]{2,}$");`
- **When to Use:** Matches entire string, not substring; use sparingly due to performance

`format(String format, Object... args)`

- **Description:** Formats string
- **Example:** `String formatted = String.format("Name: %s, Age: %d", name, age);`
- **When to Use:** Similar to printf; use template strings in Java 21+

`join(CharSequence delimiter, CharSequence... elements)`

- **Description:** Joins strings with delimiter
- **Example:** `String joined = String.join(", ", list);`
- **When to Use:** Efficient; preferable to manual concatenation

`isEmpty()`

- **Description:** Checks if length is 0
- **Example:** `boolean empty = str.isEmpty();`
- **When to Use:** Faster than length() == 0; doesn't check for whitespace

`isBlank()` (Java 11+)

- **Description:** Checks if empty or whitespace
- **Example:** `boolean blank = str.isBlank();`
- **When to Use:** Better than trim().isEmpty()

`strip()` (Java 11+)

- **Description:** Unicode-aware trim
- **Example:** `String stripped = str.strip();`
- **When to Use:** Handles all Unicode whitespace; prefer over trim()

`repeat(int count)` (Java 11+)

- **Description:** Repeats string n times
- **Example:** `String repeated = str.repeat(3);`
- **When to Use:** Efficient; throws if count negative

📋 List Methods

`add(E e)`

- **Description:** Adds element at end
- **Example:** `list.add("item");`
- **When to Use:** O(1) for ArrayList (amortized), O(1) for LinkedList

`add(int index, E element)`

- **Description:** Adds element at index
- **Example:** `list.add(0, "first");`
- **When to Use:** O(n) for ArrayList, O(n) for LinkedList (unless at ends)

`get(int index)`

- **Description:** Returns element at index
- **Example:** `String item = list.get(5);`
- **When to Use:** O(1) for ArrayList, O(n) for LinkedList

`remove(int index)`

- **Description:** Removes element at index
- **Example:** `String removed = list.remove(3);`
- **When to Use:** O(n) for ArrayList, O(n) for LinkedList (unless at ends)

`remove(Object o)`

- **Description:** Removes first occurrence
- **Example:** `boolean removed = list.remove("item");`
- **When to Use:** O(n) for both; returns boolean

`set(int index, E element)`

- **Description:** Replaces element at index
- **Example:** `String old = list.set(1, "new");`
- **When to Use:** O(1) for ArrayList, O(n) for LinkedList

`size()`

- **Description:** Returns number of elements
- **Example:** `int size = list.size();`
- **When to Use:** O(1) for both

`isEmpty()`

- **Description:** Checks if list has no elements
- **Example:** `boolean empty = list.isEmpty();`
- **When to Use:** Faster than size() == 0

`contains(Object o)`

- **Description:** Checks if list contains element
- **Example:** `boolean has = list.contains("item");`
- **When to Use:** O(n) search; uses equals()

`indexOf(Object o)`

- **Description:** Finds index of first occurrence
- **Example:** `int index = list.indexOf("item");`
- **When to Use:** O(n) search; returns -1 if not found

`lastIndexOf(Object o)`

- **Description:** Finds index of last occurrence
- **Example:** `int last = list.lastIndexOf("item");`
- **When to Use:** O(n) search; returns -1 if not found

`clear()`

- **Description:** Removes all elements
- **Example:** `list.clear();`
- **When to Use:** O(n) to nullify references

`addAll(Collection<? extends E> c)`

- **Description:** Adds all elements from collection
- **Example:** `list.addAll(otherList);`
- **When to Use:** O(n) where n is size of collection to add

`subList(int fromIndex, int toIndex)`

- **Description:** Returns view of portion of list
- **Example:** `List<String> sub = list.subList(2, 5);`
- **When to Use:** Changes to subList affect original list

`toArray(T[] a)`

- **Description:** Converts to array of type T
- **Example:** `String[] array = list.toArray(new String[0]);`
- **When to Use:** Array size 0 is efficient; Java creates right-sized array

`sort(Comparator<? super E> c)` **(Java 8+)**

- **Description:** Sorts list using comparator
- **Example:** `list.sort(Comparator.naturalOrder());`
- **When to Use:** Uses Arrays.sort for ArrayList; stable sort

`replaceAll(UnaryOperator<E> operator)` **(Java 8+)**

- **Description:** Replaces each element with result of operator
- **Example:** `list.replaceAll(String::toUpperCase);`
- **When to Use:** In-place transformation

`removeIf(Predicate<? super E> filter)` **(Java 8+)**

- **Description:** Removes elements matching predicate
- **Example:** `list.removeIf(s → s.isEmpty());`
- **When to Use:** Efficient; replaces loop with remove

🗺️ Map Methods

`put(K key, V value)`

- **Description:** Associates key with value
- **Example:** `map.put("key", value);`
- **When to Use:** O(1) average for HashMap; returns previous value if key existed

`get(Object key)`

- **Description:** Returns value for key
- **Example:** `Value value = map.get("key");`
- **When to Use:** O(1) average for HashMap; returns null if key not present

`remove(Object key)`

- **Description:** Removes mapping for key
- **Example:** `Value removed = map.remove("key");`
- **When to Use:** O(1) average for HashMap; returns removed value

`containsKey(Object key)`

- **Description:** Checks if map contains key
- **Example:** `boolean has = map.containsKey("key");`
- **When to Use:** O(1) average for HashMap; use instead of get() != null

`containsValue(Object value)`

- **Description:** Checks if map contains value
- **Example:** `boolean has = map.containsValue(value);`
- **When to Use:** O(n) for HashMap; expensive operation

`keySet()`

- **Description:** Returns set view of keys
- **Example:** `Set<String> keys = map.keySet();`
- **When to Use:** View is backed by map; changes affect the map

`values()`

- **Description:** Returns collection view of values
- **Example:** `Collection<Value> values = map.values();`
- **When to Use:** View is backed by map; changes affect the map

`entrySet()`

- **Description:** Returns set view of mappings
- **Example:** `Set<Map.Entry<String, Value>> entries = map.entrySet();`
- **When to Use:** Best way to iterate over map

`putIfAbsent(K key, V value)` **(Java 8+)**

- **Description:** Puts value if key not present
- **Example:** `Value previous = map.putIfAbsent("key", value);`
- **When to Use:** Atomic; returns null if added, existing value otherwise

`computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction)` **(Java 8+)**

- **Description:** Computes value if key not present
- **Example:** `Value value = map.computeIfAbsent("key", k → expensiveOperation());`
- **When to Use:** Perfect for lazy initialization and caching

`merge(K key, V value, BiFunction<? super V, ? super V, ? extends V> remappingFunction)` **(Java 8+)**

- **Description:** Merges value with existing or puts new
- **Example:** `Value result = map.merge("key", 1, Integer::sum);`
- **When to Use:** Perfect for counters; null return removes entry

🔍 Set Methods

`add(E e)`

- **Description:** Adds element
- **Example:** `boolean added = set.add("item");`
- **When to Use:** O(1) average for HashSet; returns false if already present

`remove(Object o)`

- **Description:** Removes element
- **Example:** `boolean removed = set.remove("item");`
- **When to Use:** O(1) average for HashSet; returns boolean

`contains(Object o)`

- **Description:** Checks if set contains element
- **Example:** `boolean has = set.contains("item");`
- **When to Use:** O(1) average for HashSet; O(log n) for TreeSet

`addAll(Collection<? extends E> c)`

- **Description:** Adds all elements from collection
- **Example:** `set.addAll(otherSet);`
- **When to Use:** Union operation; O(n) where n is collection size

`removeAll(Collection<?> c)`

- **Description:** Removes all elements in collection
- **Example:** `set.removeAll(toRemove);`
- **When to Use:** Difference operation; O(n*m) where n is this size and m is collection size

`retainAll(Collection<?> c)`

- **Description:** Keeps only elements in collection
- **Example:** `set.retainAll(toKeep);`
- **When to Use:** Intersection operation; O(n*m) where n is this size and m is collection size

TreeSet Specific Methods:

`first()`

- **Description:** Returns first (lowest) element
- **Example:** `E first = treeSet.first();`
- **When to Use:** O(log n); throws NoSuchElementException if empty

`last()`

- **Description:** Returns last (highest) element
- **Example:** `E last = treeSet.last();`
- **When to Use:** O(log n); throws NoSuchElementException if empty

`ceiling(E e)`

- **Description:** Returns least element greater than or equal to e
- **Example:** `E ceiling = treeSet.ceiling(element);`
- **When to Use:** O(log n); returns null if no such element

`floor(E e)`

- **Description:** Returns greatest element less than or equal to e
- **Example:** `E floor = treeSet.floor(element);`
- **When to Use:** O(log n); returns null if no such element

📊 Array Methods (via Arrays Utility Class)

`sort(T[] a)`

- **Description:** Sorts array
- **Example:** `Arrays.sort(array);`
- **When to Use:** O(n log n); uses dual-pivot quicksort for primitives, mergesort for objects

`binarySearch(T[] a, T key)`

- **Description:** Searches sorted array
- **Example:** `int index = Arrays.binarySearch(array, "key");`

- **When to Use:** O(log n); array must be sorted; returns negative insertion point if not found

`equals(T[] a, T[] a2)`

- **Description:** Checks if arrays equal
- **Example:** `boolean equals = Arrays.equals(array1, array2);`
- **When to Use:** Deep equals for elements

`fill(T[] a, T val)`

- **Description:** Fills array with value
- **Example:** `Arrays.fill(array, "default");`
- **When to Use:** O(n); useful for initialization

`copyOf(T[] original, int newLength)`

- **Description:** Copies array with new length
- **Example:** `String[] copy = Arrays.copyOf(array, array.length * 2);`
- **When to Use:** Good for resizing; pads with nulls or zeros if longer

`asList(T... a)`

- **Description:** Returns fixed-size List view
- **Example:** `List<String> list = Arrays.asList("a", "b", "c");`
- **When to Use:** Fixed-size; changes to list reflect in array; use List.of() for immutable list

`stream(T[] array)` **(Java 8+)**

- **Description:** Returns sequential Stream
- **Example:** `Stream<String> stream = Arrays.stream(array);`
- **When to Use:** Useful for functional operations on arrays

`parallelSort(T[] a)` **(Java 8+)**

- **Description:** Sorts array in parallel
- **Example:** `Arrays.parallelSort(array);`
- **When to Use:** Uses fork/join for large arrays; better for large datasets

🌊 Stream API Methods

Stream Creation

`stream()` **- Collection<E>.stream()**

- **Description:** Creates sequential stream
- **Example:** `Stream<String> stream = list.stream();`
- **When to Use:** Basic stream creation from collection

`of(T... values)` **- Stream.of(T... values)**

- **Description:** Creates stream from values
- **Example:** `Stream<Integer> stream = Stream.of(1, 2, 3);`
- **When to Use:** Convenient for small number of elements

`iterate(T seed, UnaryOperator<T> f)` **- Stream.iterate(T seed, UnaryOperator<T> f)**

- **Description:** Creates infinite stream by iteration
- **Example:** `Stream<Integer> stream = Stream.iterate(0, n → n + 2);`
- **When to Use:** For sequences with pattern; use with limit()

`generate(Supplier<T> s)` **- Stream.generate(Supplier<T> s)**

- **Description:** Creates infinite stream from supplier
- **Example:** `Stream<Double> stream = Stream.generate(Math::random);`
- **When to Use:** For sequences without pattern; use with limit()

Intermediate Operations

`filter(Predicate<? super T> predicate)`

- **Description:** Filters elements
- **Example:** `stream.filter(s → s.length() > 3)`
- **When to Use:** Keep elements matching condition; lazy evaluation

`map(Function<? super T, ? extends R> mapper)`

- **Description:** Transforms elements
- **Example:** `stream.map(String::toUpperCase)`
- **When to Use:** One-to-one transformation; creates new objects

`flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)`

- **Description:** Transforms and flattens
- **Example:** `stream.flatMap(s → Arrays.stream(s.split(",")))`
- **When to Use:** One-to-many transformation; flattens nested streams

`distinct()`

- **Description:** Removes duplicates

- **Example:** `stream.distinct()`
- **When to Use:** Uses equals() and hashCode(); may be costly for large streams

`sorted()`

- **Description:** Sorts elements (natural order)
- **Example:** `stream.sorted()`
- **When to Use:** Elements must be Comparable; may be costly for large streams

`limit(long maxSize)`

- **Description:** Truncates to maxSize
- **Example:** `stream.limit(10)`
- **When to Use:** Short-circuiting; useful for infinite streams

`skip(long n)`

- **Description:** Skips first n elements
- **Example:** `stream.skip(5)`
- **When to Use:** Complements limit() for pagination

Terminal Operations

`forEach(Consumer<? super T> action)`

- **Description:** Performs action for each element
- **Example:** `stream.forEach(System.out::println)`
- **When to Use:** Side effects; no result returned

`toArray()`

- **Description:** Collects to Object array
- **Example:** `Object[] array = stream.toArray()`
- **When to Use:** Use overload for typed array

`reduce(BinaryOperator<T> accumulator)`

- **Description:** Reduces to single result
- **Example:** `Optional<Integer> sum = stream.reduce(Integer::sum)`
- **When to Use:** Combines elements; returns Optional

`collect(Collector<? super T, A, R> collector)`

- **Description:** Mutable reduction
- **Example:** `List<String> list = stream.collect(Collectors.toList())`
- **When to Use:** Most flexible terminal operation; many predefined collectors

`min(Comparator<? super T> comparator)`

- **Description:** Finds minimum element
- **Example:** `Optional<String> min = stream.min(Comparator.naturalOrder())`
- **When to Use:** Returns Optional; more specific than reduce

`anyMatch(Predicate<? super T> predicate)`

- **Description:** Checks if any element matches
- **Example:** `boolean any = stream.anyMatch(s → s.startsWith("A"))`
- **When to Use:** Short-circuiting; stops at first match

`findFirst()`

- **Description:** Finds first element
- **Example:** `Optional<String> first = stream.findFirst()`
- **When to Use:** Returns Optional; respects encounter order

🔄 Optional Methods

`of(T value)`

- **Description:** Creates Optional with non-null value
- **Example:** `Optional<String> opt = Optional.of("value");`
- **When to Use:** Throws NullPointerException if value is null

`ofNullable(T value)`

- **Description:** Creates Optional that may contain null
- **Example:** `Optional<String> opt = Optional.ofNullable(possiblyNull);`
- **When to Use:** Returns empty Optional if value is null

`isPresent()`

- **Description:** Checks if value is present
- **Example:** `boolean hasValue = opt.isPresent();`
- **When to Use:** Basic presence check

`ifPresent(Consumer<? super T> consumer)`

- **Description:** Executes consumer if value present
- **Example:** `opt.ifPresent(System.out::println);`
- **When to Use:** Side effect without explicit isPresent check

`orElse(T other)`

- **Description:** Gets value or default

- **Example:** `String value = opt.orElse("default");`
- **When to Use:** Always evaluates default; use for cheap defaults

`orElseGet(Supplier<? extends T> supplier)`

- **Description:** Gets value or computed default
- **Example:** `String value = opt.orElseGet(() → computeDefault());`
- **When to Use:** Lazy evaluation; supplier only called if empty

`map(Function<? super T, ? extends U> mapper)`

- **Description:** Transforms value if present
- **Example:** `Optional<Integer> length = opt.map(String::length);`
- **When to Use:** One-to-one transformation

`flatMap(Function<? super T, Optional<U>> mapper)`

- **Description:** Transforms to another Optional
- **Example:** `Optional<User> user = getOptionalId().flatMap(this::findUserById);`
- **When to Use:** Prevents nested Optionals

📁 File/Path/IO Methods (Java NIO)

`get(String first, String... more)` **- Paths.get(String first, String... more)**

- **Description:** Creates Path from string
- **Example:** `Path path = Paths.get("dir", "file.txt");`
- **When to Use:** Platform-independent path creation

`createFile(Path path, FileAttribute<?>... attrs)` **- Files.createFile(Path path, FileAttribute<?>... attrs)**

- **Description:** Creates new empty file
- **Example:** `Files.createFile(path);`
- **When to Use:** Throws if file exists; use with exists() check

`readAllBytes(Path path)` **- Files.readAllBytes(Path path)**

- **Description:** Reads file contents as byte array
- **Example:** `byte[] bytes = Files.readAllBytes(path);`
- **When to Use:** For small files; loads entire file into memory

`readAllLines(Path path, Charset cs)` **- Files.readAllLines(Path path, Charset cs)**

- **Description:** Reads file as list of lines

- **Example:** `List<String> lines = Files.readAllLines(path, StandardCharsets.UTF_8);`
- **When to Use:** For small files; loads entire file into memory

`lines(Path path, Charset cs)` **- Files.lines(Path path, Charset cs)**

- **Description:** Returns stream of lines
- **Example:** `try (Stream<String> lines = Files.lines(path, StandardCharsets.UTF_8)) { ... }`
- **When to Use:** For large files; lazy loading; must be closed

`write(Path path, byte[] bytes, OpenOption... options)` **- Files.write(Path path, byte[] bytes, OpenOption... options)**

- **Description:** Writes byte array to file
- **Example:** `Files.write(path, bytes, StandardOpenOption.CREATE);`
- **When to Use:** For small files; use options to control behavior

`exists(Path path, LinkOption... options)` **- Files.exists(Path path, LinkOption... options)**

- **Description:** Checks if file exists
- **Example:** `boolean exists = Files.exists(path);`
- **When to Use:** Use before operations that throw if file doesn't exist

`list(Path dir)` **- Files.list(Path dir)**

- **Description:** Lists directory contents
- **Example:** `try (Stream<Path> stream = Files.list(dir)) { ... }`
- **When to Use:** Returns stream; must be closed

`walk(Path start, int maxDepth, FileVisitOption... options)` **- Files.walk(Path start, int maxDepth, FileVisitOption... options)**

- **Description:** Walks directory tree
- **Example:** `try (Stream<Path> stream = Files.walk(dir, 3)) { ... }`
- **When to Use:** Returns stream; must be closed

📚 Part 3: Java Collections Cheatsheet

📊 Core Interfaces & Implementations

Collection Hierarchy

```
Collection (Interface)
├── List (Interface)
```

```
│   ├── ArrayList (Implementation)
│   ├── LinkedList (Implementation)
│   ├── Vector (Legacy Implementation)
│   │   └── Stack (Legacy Implementation)
├── Set (Interface)
│   ├── HashSet (Implementation)
│   │   └── LinkedHashSet (Implementation)
│   ├── TreeSet (Implementation)
│   ├── EnumSet (Implementation)
├── Queue (Interface)
│   ├── PriorityQueue (Implementation)
│   ├── LinkedList (Implementation)
│   ├── ArrayDeque (Implementation)
│   └── BlockingQueue (Interface)
│       ├── ArrayBlockingQueue (Implementation)
│       ├── LinkedBlockingQueue (Implementation)
│       └── PriorityBlockingQueue (Implementation)
├── Deque (Interface)
│   ├── ArrayDeque (Implementation)
│   ├── LinkedList (Implementation)
│   └── BlockingDeque (Interface)
│       └── LinkedBlockingDeque (Implementation)
```

Map Hierarchy (Separate from Collection)

```
Map (Interface)
├── HashMap (Implementation)
│   └── LinkedHashMap (Implementation)
├── TreeMap (Implementation)
├── EnumMap (Implementation)
├── WeakHashMap (Implementation)
├── IdentityHashMap (Implementation)
├── Hashtable (Legacy Implementation)
│   └── Properties (Legacy Implementation)
├── ConcurrentMap (Interface)
│   └── ConcurrentHashMap (Implementation)
```

📝 List Implementations Comparison

Feature	ArrayList	LinkedList	Vector	CopyOnWriteArr
Internal Structure	Dynamic array	Doubly-linked list	Dynamic array	Immutable array copy-on-write
Random Access	O(1)	O(n)	O(1)	O(1)
Insert/Delete at End	O(1) amortized	O(1)	O(1) amortized	O(n)
Insert/Delete in Middle	O(n)	O(1) after finding position	O(n)	O(n)
Memory Overhead	Low	High (pointers)	Low	High (copy on w
Thread Safety	No	No	Yes (synchronized)	Yes (immutable snapshots)
Iteration Performance	Fast, cache-friendly	Slower, cache-unfriendly	Fast	Snapshot view (never fail)
Best For	Random access, fixed size or growth at end	Frequent insertions/deletions at arbitrary positions	Legacy code (use ArrayList instead)	Concurrent read heavy workload infrequent modifications
Avoid For	Frequent insertions/deletions in middle	Random access by index	New code (outdated)	Write-heavy workloads

📝 Set Implementations Comparison

Feature	HashSet	LinkedHashSet	TreeSet	EnumSet
Internal Structure	Hash table (HashMap)	Hash table with linked list	Red-black tree	Bit vectors
Order	No guaranteed order	Insertion order	Sorted order	Enum declaration order
Performance (add/remove/contains)	O(1) average	O(1) average	O(log n)	O(1)
Null Elements	One null allowed	One null allowed	No nulls	No nulls (enum values)
Memory Efficiency	Medium	Low	Medium	Very high (bit vector)
Thread Safety	No	No	No	No

Feature	HashSet	LinkedHashSet	TreeSet	EnumSet
Best For	General purpose, speed	Order preservation + speed	Sorted data, range queries	Sets of enum constants
Avoid For	Order requirements	No special advantages	Random access, speed-critical	Non-enum elements

📝 Map Implementations Comparison

Feature	HashMap	LinkedHashMap	TreeMap	EnumMap	Conc
Internal Structure	Hash table	Hash table + doubly-linked list	Red-black tree	Array	Segm table
Order	No guaranteed order	Insertion order or access order	Sorted by key	Enum declaration order	No gu
Performance (get/put)	O(1) average	O(1) average	O(log n)	O(1)	O(1) a highe
Null Keys/Values	One null key, multiple null values	One null key, multiple null values	No null keys, multiple null values	No null keys, multiple null values	No nu value
Memory Overhead	Medium	High	Medium	Very low	Medi
Thread Safety	No	No	No	No	Yes
Special Features	Basic map	LRU cache support	Range queries, ceiling/floor ops	Specialized for enum keys	Conc opera putIf
Best For	General purpose	Order tracking, LRU caches	Sorted data, range queries	Maps with enum keys	Conc
Avoid For	Order requirements	No special advantages	Random access speed	Non-enum keys	Singl (over

📝 Queue and Deque Implementations Comparison

Feature	ArrayDeque	LinkedList	PriorityQueue	ArrayBlockingQueue	L
Internal Structure	Circular array	Doubly-linked list	Heap (array-based)	Array with locks	L l

Feature	ArrayDeque	LinkedList	PriorityQueue	ArrayBlockingQueue	L
Order	FIFO	FIFO	Natural order or comparator	FIFO	F
Null Elements	No	Yes	No	No	N
Bounded Capacity	No	No	No	Yes, fixed at creation	C
Thread Safety	No	No	No	Yes	Y
Blocking Operations	No	No	No	Yes	Y
Performance	Fast at both ends	Fast at both ends	O(log n) for inserts, O(1) for peek	Contention when full/empty	L a
Best For	Stack or queue in single thread	Deque with null elements	Priority scheduling	Fixed-size producer/consumer	U p
Avoid For	Concurrent access	Random access	FIFO requirements	Unpredictable capacity needs	N

🔒 Concurrent Collections Comparison

Collection	Key Features	Performance Characteristics	Use Cases
ConcurrentHashMap	Segmented locking, atomic operations, no locks for reads	High throughput, good scalability	Thread-safe maps with high concurrency
CopyOnWriteArrayList	Thread-safe reads without locking, copies array on modification	Fast reads, expensive writes	Read-heavy, rarely modified collections
CopyOnWriteArraySet	Set backed by CopyOnWriteArrayList	Fast reads, expensive writes	Read-heavy sets with thread safety
ConcurrentSkipListMap	Concurrent sorted map implementation	O(log n) operations, lock-free	Concurrent access to sorted map
ConcurrentSkipListSet	Concurrent sorted set implementation	O(log n) operations, lock-free	Concurrent access to sorted set
ArrayBlockingQueue	Bounded blocking queue backed by array	Predictable performance	Producer-consumer with fixed capacity
LinkedBlockingQueue	Optionally bounded blocking queue	Less contention than array version	Producer-consumer with

Collection	Key Features	Performance Characteristics	Use Cases
			unbounded or very large capacity
PriorityBlockingQueue	Blocking priority queue	O(log n) inserts, O(1) peek	Concurrent priority-based processing
DelayQueue	Queue that releases elements after a delay	Based on PriorityQueue	Scheduled tasks, rate limiting
SynchronousQueue	Queue with no capacity	Direct handoff	Direct producer-consumer handoffs
LinkedTransferQueue	Combines features of SynchronousQueue and LinkedBlockingQueue	High throughput	Producer-consumer with optional synchronous transfer

🔒 Immutable Collections

Method	Description	Example	Java Versi
List.of()	Creates immutable list	`List<String> list = List.of("a", "b", "c");`	Java 9+
Set.of()	Creates immutable set	`Set<String> set = Set.of("a", "b", "c");`	Java 9+
Map.of()	Creates immutable map (up to 10 entries)	`Map<String, Integer> map = Map.of("a", 1, "b", 2);`	Java 9+
Map.ofEntries()	Creates immutable map with arbitrary entries	`Map<String, Integer> map = Map.ofEntries(Map.entry("a", 1), Map.entry("b", 2));`	Java 9+
List.copyOf()	Creates immutable copy of collection	`List<String> copy = List.copyOf(originalList);`	Java 10+
Set.copyOf()	Creates immutable copy of collection	`Set<String> copy = Set.copyOf(originalSet);`	Java 10+
Map.copyOf()	Creates immutable	`Map<String, Integer> copy = Map.copyOf(originalMap);`	Java 10+

Method	Description	Example	Java Versi
	copy of map		
Collections.unmodifiableList()	Returns unmodifiable view of list	`List<String> unmodifiable = Collections.unmodifiableList(list);`	Java 1.2+
Collections.unmodifiableSet()	Returns unmodifiable view of set	`Set<String> unmodifiable = Collections.unmodifiableSet(set);`	Java 1.2+
Collections.unmodifiableMap()	Returns unmodifiable view of map	`Map<String, Integer> unmodifiable = Collections.unmodifiableMap(map);`	Java 1.2+

🛠️ Utility Methods (Collections Class)

Method	Description	Example	Performance
sort(List<T> list)	Sorts list in natural order	`Collections.sort(list);`	O(n log n)
sort(List<T> list, Comparator<? super T> c)	Sorts list using comparator	`Collections.sort(list, Comparator.reverseOrder());`	O(n log n)
binarySearch(List<? extends Comparable<? super T>> list, T key)	Searches sorted list	`int index = Collections.binarySearch(list, "key");`	O(log n)
reverse(List<?> list)	Reverses list order	`Collections.reverse(list);`	O(n)
shuffle(List<?> list)	Randomly permutes list	`Collections.shuffle(list);`	O(n)
fill(List<? super T> list, T obj)	Replaces all elements with obj	`Collections.fill(list, "default");`	O(n)
copy(List<? super T> dest, List<? extends T> src)	Copies src to dest	`Collections.copy(dest, src);`	O(n)
min(Collection<? extends T> coll)	Returns minimum element	`T min = Collections.min(collection);`	O(n)
max(Collection<? extends T> coll)	Returns maximum element	`T max = Collections.max(collection);`	O(n)
rotate(List<?> list, int distance)	Rotates list by distance	`Collections.rotate(list, 2);`	O(n)
replaceAll(List<T> list, T oldVal, T newVal)	Replaces all occurrences	`Collections.replaceAll(list, "old", "new");`	O(n)
frequency(Collection<?> c, Object o)	Counts occurrences	`int count = Collections.frequency(collection,`	O(n)

Method	Description	Example	Performance
		`"item");`	
disjoint(Collection<?> c1, Collection<?> c2)	Checks if collections are disjoint	`boolean disjoint = Collections.disjoint(set1, set2);`	O(n*m) or O(n+m)
addAll(Collection<? super T> c, T... elements)	Adds all elements	`Collections.addAll(collection, "a", "b", "c");`	O(n)
singleton(T o)	Returns immutable set with one element	`Set<String> single = Collections.singleton("item");`	O(1)
singletonList(T o)	Returns immutable list with one element	`List<String> single = Collections.singletonList("item");`	O(1)
singletonMap(K key, V value)	Returns immutable map with one entry	`Map<String, Integer> single = Collections.singletonMap("key", 1);`	O(1)
emptyList()	Returns immutable empty list	`List<String> empty = Collections.emptyList();`	O(1)
emptySet()	Returns immutable empty set	`Set<String> empty = Collections.emptySet();`	O(1)
emptyMap()	Returns immutable empty map	`Map<String, Integer> empty = Collections.emptyMap();`	O(1)

🔒 Thread Safety & Mutability

Making Collections Thread-Safe

Synchronized Wrappers:

```
// For List
List<String> syncList = Collections.synchronizedList(new ArrayList<>());

// For Set
Set<String> syncSet = Collections.synchronizedSet(new HashSet<>());

// For Map
Map<String, Integer> syncMap = Collections.synchronizedMap(new HashMap<
>());
```

Concurrent Collections:

```
// Instead of synchronized HashMap
Map<String, Integer> concurrentMap = new ConcurrentHashMap<>();

// Instead of synchronized ArrayList
List<String> concurrentList = new CopyOnWriteArrayList<>();

// Instead of synchronized TreeMap
Map<String, Integer> concurrentSortedMap = new ConcurrentSkipListMap<>();
```

Synchronized Wrappers vs. Concurrent Collections

Feature	Synchronized Wrappers	Concurrent Collections
Implementation	Single lock for entire collection	Fine-grained locking or lock-free algorithms
Contention	High (one thread at a time)	Low (multiple threads can access different parts)
Iteration	Must manually synchronize iterators	Fail-safe iterators (snapshot or weakly consistent)
Performance	Lower throughput under contention	Higher throughput, better scalability
Atomic Operations	Must use external synchronization for compound operations	Built-in atomic compound operations (putIfAbsent, etc.)
Best For	Simple thread safety requirements, low contention	High-concurrency environments, scalability requirements

⏱️ Big-O Time Complexity

List Operations

Operation	ArrayList	LinkedList
add at end	O(1) amortized	O(1)
add at index	O(n)	O(n)
get	O(1)	O(n)
remove at end	O(1)	O(1)
remove at index	O(n)	O(n)
contains	O(n)	O(n)
size	O(1)	O(1)
isEmpty	O(1)	O(1)

Operation	ArrayList	LinkedList
iterator.next()	O(1)	O(1)

Set Operations

Operation	HashSet	LinkedHashSet	TreeSet
add	O(1) average	O(1) average	O(log n)
remove	O(1) average	O(1) average	O(log n)
contains	O(1) average	O(1) average	O(log n)
size	O(1)	O(1)	O(1)
isEmpty	O(1)	O(1)	O(1)
iterator.next()	O(1) amortized	O(1)	O(log n)
first/last	N/A	N/A	O(log n)
ceiling/floor	N/A	N/A	O(log n)

Map Operations

Operation	HashMap	LinkedHashMap	TreeMap
put	O(1) average	O(1) average	O(log n)
get	O(1) average	O(1) average	O(log n)
remove	O(1) average	O(1) average	O(log n)
containsKey	O(1) average	O(1) average	O(log n)
containsValue	O(n)	O(n)	O(n)
size	O(1)	O(1)	O(1)
isEmpty	O(1)	O(1)	O(1)
keySet/values/entrySet	O(1)	O(1)	O(1)
floorKey/ceilingKey	N/A	N/A	O(log n)

📋 When to Use / Avoid

List Implementation Selection Guide

Use ArrayList when:

- Random access by index is frequent
- Size changes infrequently or only grows at end
- Iteration performance is important
- Memory efficiency matters

Use LinkedList when:

- Frequent insertions/deletions at both ends
- Implementing both List and Queue interfaces
- Implementing a stack or queue

Use CopyOnWriteArrayList when:

- Reads vastly outnumber writes
- Need thread-safe iteration without explicit synchronization
- Need to prevent ConcurrentModificationException

Set Implementation Selection Guide

Use HashSet when:

- Fast lookup, insertion, deletion is primary concern
- Order doesn't matter
- Implementing simple set membership

Use LinkedHashSet when:

- Need insertion-order iteration
- Fast access with predictable iteration

Use TreeSet when:

- Elements must be sorted
- Need range queries (ceiling, floor, etc.)
- Need elements always in sorted order

Use EnumSet when:

- Set elements are all from a single enum type
- Memory efficiency is important
- Performance is critical

Map Implementation Selection Guide

Use HashMap when:

- Basic key-value storage
- Maximum performance is needed
- No special ordering requirements

Use LinkedHashMap when:

- Need insertion-order or access-order iteration
- Implementing LRU caches
- Predictable iteration order matters

Use TreeMap when:

- Keys must be sorted
- Need range operations (headMap, tailMap, etc.)
- Need to find closest matches (ceiling, floor, etc.)

Use ConcurrentHashMap when:

- High-concurrency environments
- Need atomic compound operations
- Need thread-safe without external locking

Use WeakHashMap when:

- Implementing caches where entries can be garbage collected
- Memory-sensitive caches with unpredictable lifetimes

Queue Implementation Selection Guide

Use ArrayDeque when:

- Implementing a stack or queue
- Need efficient operations at both ends
- General-purpose double-ended queue

Use PriorityQueue when:

- Need elements processed in priority order
- Implementing algorithms like Dijkstra's or Huffman coding
- Task scheduling based on priority

Use LinkedBlockingQueue when:

- Producer-consumer patterns
- Need thread-safe queue with blocking operations
- Potentially unbounded capacity

Use ArrayBlockingQueue when:

- Bounded producer-consumer scenarios
- Need thread-safe queue with blocking operations
- Fixed capacity is acceptable or required

Use SynchronousQueue when:

- Direct handoffs between threads
- No actual queueing needed
- Ensuring "one in, one out" processing

✈️ Happy Coding!
