**1. Declaring and Initializing Collections**

**List**

A List is an ordered collection that allows duplicate elements.

* **Declaration**:
* List<String> list;
* **Initialization**:
* list = new ArrayList<>(); // Using ArrayList
* list = new LinkedList<>(); // Using LinkedList
* **With elements** (using Arrays.asList()):
* List<String> list = new ArrayList<>(Arrays.asList("Apple", "Banana", "Cherry"));

**Set**

A Set is an unordered collection that does not allow duplicate elements.

* **Declaration**:
* Set<String> set;
* **Initialization**:
* set = new HashSet<>(); // Using HashSet
* set = new TreeSet<>(); // Using TreeSet
* **With elements**:
* Set<String> set = new HashSet<>(Arrays.asList("Apple", "Banana", "Cherry"));

**Map**

A Map stores key-value pairs and does not allow duplicate keys.

* **Declaration**:
* Map<Integer, String> map;
* **Initialization**:
* map = new HashMap<>(); // Using HashMap
* map = new TreeMap<>(); // Using TreeMap
* **With elements**:
* Map<Integer, String> map = new HashMap<>();
* map.put(1, "Apple");
* map.put(2, "Banana");
* map.put(3, "Cherry");

**2. List Interface Methods**

* **add()**  
  Adds an element to the list.
* list.add("Apple");
* **get()**  
  Retrieves the element at a specified index.
* String fruit = list.get(0);
* **set()**  
  Replaces the element at a specified index.
* list.set(0, "Orange");
* **remove()**  
  Removes an element by index or by value.
* list.remove(0); // by index
* list.remove("Orange"); // by value
* **clear()**  
  Removes all elements from the list.
* list.clear();
* **contains()**  
  Checks if an element exists in the list.
* list.contains("Apple");
* **size()**  
  Returns the number of elements in the list.
* int size = list.size();
* **indexOf()**  
  Finds the index of the first occurrence of an element.
* int index = list.indexOf("Apple");
* **isEmpty()**  
  Checks if the list is empty.
* boolean empty = list.isEmpty();
* **subList()**  
  Returns a sublist from fromIndex to toIndex.
* List<String> sublist = list.subList(0, 2);

**3. Set Interface Methods**

* **add()**  
  Adds an element to the set.
* set.add("Apple");
* **remove()**  
  Removes an element from the set.
* set.remove("Apple");
* **contains()**  
  Checks if an element is present in the set.
* set.contains("Apple");
* **size()**  
  Returns the number of elements in the set.
* int size = set.size();
* **isEmpty()**  
  Checks if the set is empty.
* boolean empty = set.isEmpty();
* **clear()**  
  Removes all elements from the set.
* set.clear();
* **addAll()**  
  Adds all elements from another collection.
* set.addAll(anotherSet);

**4. Map Interface Methods**

* **put()**  
  Adds a key-value pair to the map.
* map.put(1, "Apple");
* **get()**  
  Retrieves the value for a specific key.
* String value = map.get(1);
* **remove()**  
  Removes a key-value pair by key.
* map.remove(1);
* **containsKey()**  
  Checks if a specific key is in the map.
* map.containsKey(1);
* **containsValue()**  
  Checks if a specific value is in the map.
* map.containsValue("Apple");
* **size()**  
  Returns the number of entries in the map.
* int size = map.size();
* **isEmpty()**  
  Checks if the map is empty.
* boolean empty = map.isEmpty();
* **clear()**  
  Removes all entries from the map.
* map.clear();
* **keySet()**  
  Returns a set of keys.
* Set<Integer> keys = map.keySet();
* **values()**  
  Returns a collection of values.
* Collection<String> values = map.values();

**5. Collections Utility Methods**

* **sort()**  
  Sorts a list in ascending order.
* Collections.sort(list);
* **reverse()**  
  Reverses the elements in a list.
* Collections.reverse(list);
* **shuffle()**  
  Randomly shuffles the elements in a list.
* Collections.shuffle(list);
* **min()**  
  Returns the minimum element in a collection.
* String min = Collections.min(list);
* **max()**  
  Returns the maximum element in a collection.
* String max = Collections.max(list);
* **binarySearch()**  
  Searches for a key in a sorted list (binary search).
* int index = Collections.binarySearch(list, "Apple");
* **copy()**  
  Copies elements from one list to another.
* Collections.copy(destList, srcList);
* **frequency()**  
  Returns the number of occurrences of an element in a collection.
* int frequency = Collections.frequency(list, "Apple");
* **fill()**  
  Fills a list with the specified value.
* Collections.fill(list, "Orange");
* **unmodifiableList()**  
  Returns an unmodifiable view of the list.
* List<String> unmodifiableList = Collections.unmodifiableList(list);
* **synchronizedList()**  
  Returns a synchronized (thread-safe) list.
* List<String> synchronizedList = Collections.synchronizedList(list);
* **singletonList()**  
  Returns a singleton list containing one element.
* List<String> singletonList = Collections.singletonList("Apple");
* **emptyList()**  
  Returns an empty list.
* List<String> emptyList = Collections.emptyList();

**6. Common Java Collection Implementations**

* **ArrayList**  
  A dynamic array that supports random access and resizing.
* List<String> list = new ArrayList<>();
* **LinkedList**  
  A doubly linked list that supports efficient insertions and deletions.
* List<String> list = new LinkedList<>();
* **HashSet**  
  A set backed by a hash table, which does not allow duplicate elements and has no specific order.
* Set<String> set = new HashSet<>();
* **TreeSet**  
  A sorted set backed by a red-black tree.
* Set<String> set = new TreeSet<>();
* **HashMap**  
  A map backed by a hash table, which stores key-value pairs with no specific order.
* Map<Integer, String> map = new HashMap<>();
* **TreeMap**  
  A sorted map that uses a red-black tree to store key-value pairs.
* Map<Integer, String> map = new TreeMap<>();

This **Java Collections Cheat Sheet** now includes initialization and declaration examples for commonly used collection types (List, Set, Map) along with their frequent methods and utility functions from the Collections class. You can use this as a quick reference for your Java programming tasks.

Here’s a quick comparison of **List**, **Set**, and **Map** in terms of one key difference:

* **List**:
  + **Order is maintained**. Lists store elements in a specific order, and you can access elements by their index.
* **Set**:
  + **No duplicates allowed**. Sets only store unique elements and do not maintain order (unless using a specialized Set, like LinkedHashSet in Java).
* **Map**:
  + **Stores key-value pairs**. A Map is a collection of key-value pairs where each key is unique, and you retrieve values using keys.

Sure! Here’s a concise two-point comparison for each implementation of List, Set, and Map:

**List Implementations:**

1. **ArrayList**:
   * **Fast random access** (get(index) is O(1)).
   * **Slow insertion/removal** at the beginning or middle (O(n)).
2. **LinkedList**:
   * **Fast insertions/removals** at the beginning or end (O(1)).
   * **Slow random access** (get(index) is O(n)).
3. **Vector**:
   * **Thread-safe** (synchronized).
   * **Slower** than ArrayList due to synchronization overhead.

**Set Implementations:**

1. **HashSet**:
   * **No ordering** (elements are unordered).
   * **Fast operations** (O(1) average time for add(), remove(), and contains()).
2. **LinkedHashSet**:
   * **Maintains insertion order**.
   * **Slightly slower** than HashSet due to linked list overhead.
3. **TreeSet**:
   * **Maintains elements in sorted order** (natural ordering or custom comparator).
   * **Slower operations** (O(log n)) due to red-black tree structure.

**Map Implementations:**

1. **HashMap**:
   * **No ordering** of keys (unordered).
   * **Fast operations** (O(1) average time for put(), get(), and remove()).
2. **LinkedHashMap**:
   * **Maintains insertion order** of keys.
   * **Slightly slower** than HashMap due to additional linked list overhead.
3. **TreeMap**:
   * **Stores keys in sorted order** (natural order or custom comparator).
   * **Slower operations** (O(log n)) due to red-black tree structure.

**What is a Stream?**

* A **Stream** in Java is a sequence of elements that can be processed in parallel or sequentially.
* It provides a functional approach to handle collections, such as List, Set, or Map.
* Introduced in **Java 8**, Streams allow you to perform complex operations like filtering, mapping, and reducing in a more declarative way.

**Key Characteristics of Streams:**

1. **No Storage**: A Stream doesn’t store data. It operates on data from a source (like a collection or array).
2. **Functional Operations**: Streams support **functional-style operations** (like map, filter, reduce), making it more readable and concise.
3. **Lazy Evaluation**: Intermediate operations on streams are **lazy**, meaning they only execute when a terminal operation is called.
4. **Can be Parallel**: Streams can be processed in parallel to improve performance, especially with large datasets.

**Types of Stream Operations:**

1. **Intermediate Operations**:
   * **Definition**: Operations that transform the stream into another stream. They are **lazy** and only executed when a terminal operation is triggered.
   * **Examples**:
     + filter(): Filters elements based on a condition.
     + map(): Transforms each element.
     + sorted(): Sorts elements.
   * **Example**:
   * List<String> list = Arrays.asList("apple", "banana", "cherry");
   * list.stream()
   * .filter(s -> s.startsWith("a")) // Intermediate operation
   * .map(String::toUpperCase) // Intermediate operation
   * .forEach(System.out::println); // Terminal operation
2. **Terminal Operations**:
   * **Definition**: Operations that produce a result and **terminate** the stream. Once a terminal operation is invoked, the stream is consumed and can no longer be used.
   * **Examples**:
     + collect(): Collects the stream into a different collection (e.g., List, Set).
     + forEach(): Performs an action on each element.
     + reduce(): Combines elements into a single result.
     + count(): Counts elements.
   * **Example**:
   * long count = list.stream()
   * .filter(s -> s.startsWith("a"))
   * .count(); // Terminal operation, counts elements
   * System.out.println(count); // Output: 1

**Stream Pipeline Example:**

A stream operation is often chained together as a **pipeline**:

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);

long count = numbers.stream()

.filter(n -> n % 2 == 0) // Intermediate operation

.map(n -> n \* 2) // Intermediate operation

.count(); // Terminal operation

System.out.println(count); // Output: 3

**Common Stream Methods:**

* **filter(Predicate)**: Filters elements based on a condition.
* **map(Function)**: Applies a function to each element.
* **collect(Collectors)**: Collects the result into a collection (e.g., List, Set).
* **forEach(Consumer)**: Performs an action for each element.
* **reduce(BinaryOperator)**: Reduces the stream to a single value.

**Parallel Streams:**

* You can use parallel streams to process elements in parallel, making it possible to speed up large datasets.
* **Example**:
* list.parallelStream().forEach(System.out::println);

**Key Points to Remember:**

1. **Streams are not data structures**; they represent a sequence of data.
2. **Intermediate operations** are lazy; they don’t execute until a **terminal operation** is called.
3. Streams provide a **functional** approach to processing collections.

In Java, a **stream pipeline** is a series of **intermediate** and **terminal operations** chained together to process the elements of the stream.