**900102-000-00-KM-04, Principles of Intermediate Programming with Java, NQF Level 4, Credits 6**

**Learner Guide**

**MODULE Four (4)**

|  |  |
| --- | --- |
| **Module Code** | 900102-000-00-KM-04 |
| **NQF Level** | 4 |
| **Credits** | 6 |
| **Skills Programme ID Number** | SP- 220329 |
| **Curriculum Title** | Java Programmer |
| **Curriculum Code** | 900102-000-00-00 |

|  |  |
| --- | --- |
| **Name** |  |
| **Contact Address** |  |
| **Telephone (H)** |  |
| **Telephone (W)** |  |
| **Facsimile** |  |
| **Cellular** |  |
| **E-mail** |  |

**Note to the learner**

This Learner Guide provides a comprehensive overview of the module. It is designed to improve the skills and knowledge of learners, and thus enabling them to effectively and efficiently complete specific tasks.

**Purpose of the Module**

The main focus of the learning in this knowledge module is to build an understanding of the principles of intermediated programming with Java.

The learning will enable learners to demonstrate an understanding of:

* KM-04-KT01: Java Collections 50%
* KM-04-KT02: Introduction to Generics in Java Programming 10%
* KM-04-KT03: Functional programming in Java 30%
* KM-04-KT04: Concurrency with concurrent collections and atomic operations 10%

**Provider Accreditation Requirements for the Knowledge Module**

**Physical Requirements:**

* The provider must have lesson plans and structured learning material or provide learners with access to structured learning material that addresses all the topics in all the knowledge modules as well as the applied knowledge in the application.
* QCTO/ MICT SETA requirements

**Human Resource Requirements:**

* Qualification of lecturer (SME):
* NQF 5 qualified in industry recognised qualifications with 1 year experience in the IT industry o Cybersecurity vendor certification
* Assessors and moderators: accredited by the MICT SETA

**Legal Requirements:**

* Legal (product) licences to use the software for learning and training
* OHS compliance certificate

**Exemptions**

* RPL based

**Venue, Date and Time:**

Consult your facilitator should there be any changes to the venue, date and/or time.Refer to your timetable.

**Assessments**

**Integrated Formative Assessment:** The skills development provider will use the curriculum to guide them on the stipulated internal assessment criteria and weighting. They will also apply the scope of practical skills and applied knowledge as stipulated by the internal assessment criteria. This formative assessment leads to entrance into the integrated external summative assessment.

**Integrated Summative Assessment**: An external integrated summative assessment conducted through the relevant QCTO Assessment Quality Partner is required to issue this qualification. The external integrated summative assessment will focus on the exit level outcomes and associated assessment criteria.

**Skills Programme Purpose**

A Java Programmer will be able to implement solutions to solve real-life problems in an efficient manner, applying a knowledge and understanding of the principles of programming with Java and applicable tools. Tasks that the learner will be able to know, do and understand after achievement of the skills programme include:

* Create well-written and readable Java programs, using a disciplined coding style, including documentation and indentation standards.
* Use Git functionalities for working collaboratively in a team and execute version control.

**Skills Programme Rationale**

Realising the importance and future impact of the Fourth Industrial Revolution (4IR) on the economy of South Africa and its competitiveness, the Minister of Communications gazetted the Presidential Commission on the Fourth Industrial Revolution (PC4IR) on 9 April 2019. By March 2020 this Commission delivered a report with wide ranging recommendations for Human Capital Development that will drive the 4IR forward. It clearly indicated the speed at which companies will have to invest in big data analysis, web-enabled market investment and the use of cloud computing and machine learning.

Software development is central to these initiatives. Software developers are the creative minds behind computer programs. Some develop the applications that allow people to do specific tasks on a computer or another device. Others develop the underlying systems that run the devices or that control networks. The software developer is the important cog in designing advanced computerised technologies. South Africa has a scarcity of software developers and there is a clear need for a qualification focusing specifically on the training and education of software developers.

**Entry Requirements**

Grade 11 with Maths Lit and English.

Access to equipment, internet connectivity and how to work remotely

**EXIT LEVEL OUTCOMES**

**Exit Level Outcomes (ELO) 1**

Describe the basics of Java Programming

Associated Assessment Criteria (AACs)

* The fundamentals of the Java programming language are explained.
* The basic concepts and methods of object-oriented programming and object-oriented design are described.
* The development life-cycle as a means of creating applications is described.

**Exit Level Outcomes (ELO) 2**

Programme effectively using Java frameworks and functionalities

Associated Assessment Criteria (AACs)

* Java syntax is demonstrated, using the Java API.
* Well-written and readable Java programs are created, using a disciplined coding style, including documentation and indentation standards.
* Problems with application development are addressed by troubleshooting.

**Exit Level Outcomes (ELO) 3**

Work collaboratively in a team using GitHub platform

Associated Assessment Criteria (AACs)

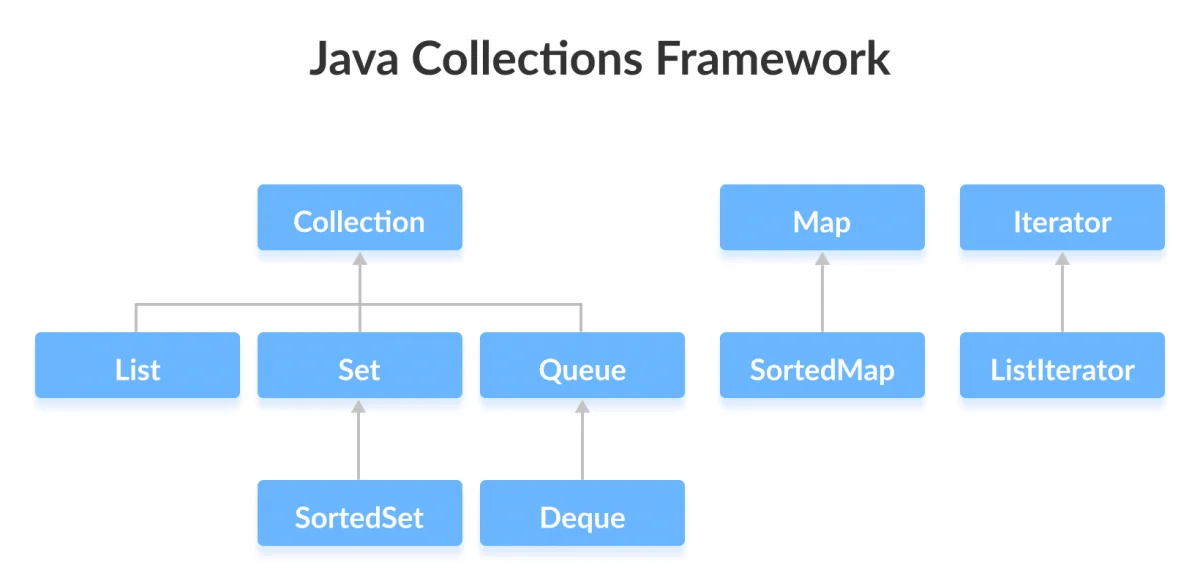
* An ability to work with GitHub is demonstrated.
* Working in a team collaboratively is achieved by using GitHub.
* Version control is exercised using GitHub. functionalities such as repositories, branches, commits and pull requests

**Session 1:** **KM-04-KT01: Java Collections**

Topic elements to be covered include:

* KT0101 Need for Collections
* KT0102 Java collection framework Hierarchy
* KT0103 List Interface
  + - Position/order is advisable
    - Immutability and Introduction of Implementations – Array
    - Implementations - ArrayList vs LinkedList
    - Implementations - ArrayList vs Vector
    - Methods to add, remove and change elements and lists
* KT0104 List and ArrayList
  + - Iterating around elements
    - Choosing iteration approach for printing and display
    - Type safety and removing integers
    - Sorting - introduction to collections sort static
    - Sorting - implementing comparable interface
* KT0105 Data Structures
  + - Array, LinkedList and Hashing
    - Tree - Sorted Order
* KT0106 Set Interface
  + - Hands-on - HashSet, LinkedHashSet and TreeSet
    - Find Unique Characters in a List
    - TreeSet - Methods from NavigableSet - floor, lower, upper, subSet, head
    - Queue Interface - Process Elements in Order
* KT0107 Introduction to PriorityQueue - Basic Methods and Customized Priority
* KT0108 Map Interface
  + - Key and Value
    - Implementations - HashMap, HashTable, LinkedHashMap
    - Basic Operations
    - Comparison - HashMap vs LinkedHashMap vs TreeMap
    - Count occurrences of characters and words
    - TreeMap - Methods from NavigableMap - floorKey, higherKey, firstEntry

**KT0101 Need for Collections**

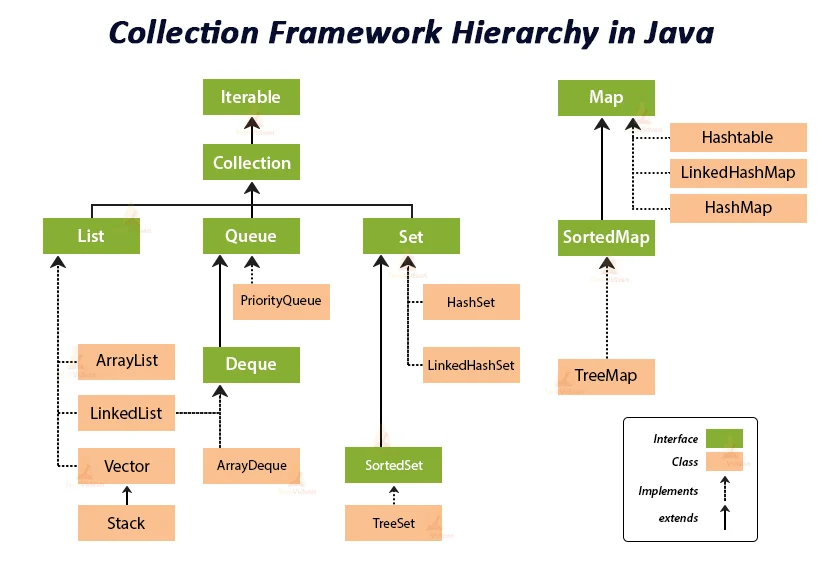


Collections in Java are fundamental data structures that are used to store and manipulate groups of objects. They provide a way to manage, store, and organize data efficiently. The need for collections in Java arises from several factors:

* **Dynamic Data Storage:** In many real-world applications, the amount of data to be stored is not known at compile time. Collections provide dynamic data storage that can grow or shrink as needed. This flexibility is crucial when dealing with changing data sets.
* **Ease of Use:** Collections offer a set of high-level, easy-to-use data structures that abstract away many low-level details. They simplify common operations such as adding, removing, and searching for elements in a collection.
* **Code Reusability:** Collections allow you to create reusable and generic data structures that can work with various data types. This promotes code reusability and reduces the need to write custom data structures for different scenarios.
* **Efficient Data Retrieval:** Collections are designed to provide efficient access to elements, whether you want to retrieve an element by its index, search for a specific element, or iterate through the collection.
* **Standardization:** Java's Collections Framework defines a set of standard interfaces and classes for working with collections. This standardization ensures consistency and interoperability between different collection types.
* **Memory Management:** Collections manage memory efficiently by allocating memory as needed and automatically resizing when the collection grows. This helps in optimizing memory usage.
* **Type Safety:** Modern collections, when used with generics, provide type safety, preventing the mixing of different data types within a collection. This helps catch type-related errors at compile time rather than at runtime.
* **Concurrency Support:** Java provides concurrent collection classes that allow multiple threads to work with collections safely. This is crucial in multithreaded and parallel programming scenarios.
* **Algorithms and Operations:** Collections come with a set of built-in algorithms and operations, such as sorting, searching, filtering, and mapping, which simplify data manipulation.
* **Compatibility with Libraries:** Collections are compatible with many Java libraries and APIs, making it easier to integrate data structures into various parts of an application.

Overall, collections are a fundamental part of Java's utility in managing and processing data efficiently. They play a central role in many Java applications, ranging from simple data storage to complex data processing tasks. Java's Collections Framework provides a rich set of collection classes and interfaces to cater to various needs, making it a valuable tool for Java developers.

**KT0102 Java collection framework Hierarchy**



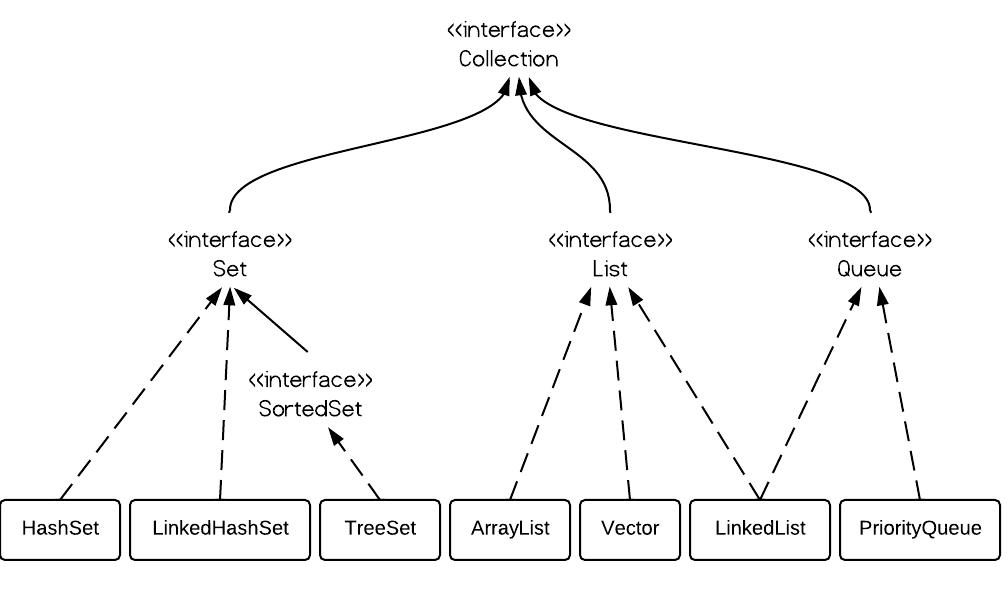
The Java Collections Framework (JCF) is a comprehensive set of classes and interfaces in Java that provides a standardized way to work with collections of objects. It is designed to be easy to use and efficient in terms of both time and space complexity. The framework is organized into a hierarchy, with key interfaces and classes at the top-level hierarchy and more specialized collections extending from them. Here's an overview of the Java Collections Framework hierarchy:

1. **java.util.Collection (Interface):**
   * This is the root interface of the Collections Framework hierarchy. It represents a group of objects known as elements.
   * Subinterfaces of Collection include List, Set, and Queue.
2. **java.util.List (Interface):**
   * Extends Collection and represents an ordered collection of elements. Lists allow duplicate elements and maintain the order of insertion.
   * Common implementations of List include ArrayList, LinkedList, and Vector.
3. **java.util.Set (Interface):**
   * Extends Collection and represents an unordered collection of elements with no duplicates. Sets do not maintain insertion order.
   * Common implementations of Set include HashSet, LinkedHashSet, and TreeSet.
4. **java.util.SortedSet (Interface):**
   * Extends Set and represents a sorted set of elements. SortedSets are always in ascending order based on their natural order or a specified comparator.
   * Common implementation of SortedSet is TreeSet.
5. **java.util.Queue (Interface):**
   * Extends Collection and represents a queue (FIFO - First-In-First-Out) data structure. Elements are added to the end (tail) and removed from the beginning (head).
   * Common implementations of Queue include LinkedList and PriorityQueue.
6. **java.util.Map (Interface):**
   * Represents a collection of key-value pairs. Each key is associated with a value, and keys are unique.
   * Subinterfaces of Map include SortedMap.
7. **java.util.SortedMap (Interface):**
   * Extends Map and represents a sorted map of key-value pairs. SortedMaps are always sorted based on keys' natural order or a specified comparator.
   * Common implementation of SortedMap is TreeMap.
8. **java.util.AbstractCollection (Class):**
   * Provides a skeletal implementation of the Collection interface to minimize the effort required to implement custom collections.
9. **java.util.AbstractList (Class):**
   * Extends AbstractCollection and provides a skeletal implementation of the List interface, simplifying the creation of custom lists.
10. **java.util.AbstractSet (Class):**
    * Extends AbstractCollection and provides a skeletal implementation of the Set interface for creating custom sets.
11. **java.util.AbstractMap (Class):**
    * Provides a skeletal implementation of the Map interface, which can be extended to create custom map implementations.
12. **java.util.HashMap (Class):**
    * Implements the Map interface using a hash table for key-value pairs. It allows fast access to elements based on their keys.
13. **java.util.LinkedHashMap (Class):**
    * Extends HashMap and maintains the insertion order of key-value pairs.
14. **java.util.TreeMap (Class):**
    * Implements SortedMap and uses a Red-Black tree to store key-value pairs in a sorted order.
15. **java.util.ArrayList (Class):**
    * Implements List using a dynamically resizable array.
16. **java.util.LinkedList (Class):**
    * Implements List and Queue using a doubly-linked list.
17. **java.util.HashSet (Class):**
    * Implements Set using a hash table for storage.
18. **java.util.LinkedHashSet (Class):**
    * Extends HashSet and maintains the insertion order of elements.
19. **java.util.TreeSet (Class):**
    * Implements SortedSet using a Red-Black tree for sorted storage.
20. **java.util.PriorityQueue (Class):**
    * Implements Queue using a priority heap.

This hierarchy provides a rich set of classes and interfaces for working with collections in Java, offering flexibility and efficiency in various data manipulation tasks. Developers can choose the appropriate collection type based on the specific requirements of their applications.

**KT0103 List Interface**

* **Position/order is advisable**
* **Immutability and Introduction of Implementations – Array**
* **Implementations - ArrayList vs LinkedList**
* **Implementations - ArrayList vs Vector**
* **Methods to add, remove and change elements and lists**



The **List** interface in Java is part of the Java Collections Framework and represents an ordered collection of elements. Lists allow duplicate elements, and they maintain the order of insertion. Here, we'll explore various aspects of the **List** interface, including its characteristics, immutability, and some common implementations such as **ArrayList**, **LinkedList**, and **Vector**. We'll also discuss methods for adding, removing, and modifying elements within lists.

**Characteristics of the List Interface:**

1. **Ordered:** Lists maintain the order of elements as they are inserted. You can access elements by their index.
2. **Allows Duplicates:** Lists can contain duplicate elements. Each element is identified by its index.
3. **Dynamic Size:** Lists can grow or shrink dynamically as elements are added or removed.

**Immutability of List Interface:**

The **List** interface itself does not guarantee immutability. It provides methods to add, remove, and modify elements, so you can change the content of a list if needed. However, if you want to create an immutable list, you can use **Collections.unmodifiableList()** to wrap an existing list and make it unmodifiable.

List<String> mutableList = new ArrayList<>(); mutableList.add("Apple"); mutableList.add("Banana"); List<String> immutableList = Collections.unmodifiableList(mutableList); // Attempting to modify the immutable list will result in an exception immutableList.add("Cherry"); // This will throw UnsupportedOperationException

**Implementations of List:**

1. **ArrayList:**
   * Implemented using a dynamically resizable array.
   * Provides fast access to elements by index.
   * Suitable for scenarios where random access and fast iteration are required.
   * Adding or removing elements from the middle of the list is less efficient than appending to the end.
2. **LinkedList:**
   * Implemented using a doubly-linked list.
   * Provides efficient insertion and removal of elements at both the beginning and end of the list.
   * Suitable for scenarios where frequent insertions and removals are required, especially in the middle of the list.
   * Iterating through the list may be slower than **ArrayList**.
3. **Vector:**
   * Similar to **ArrayList** but synchronized, making it thread-safe.
   * Synchronization comes at the cost of performance, so it is less efficient than **ArrayList** for single-threaded scenarios.
   * Generally not recommended in modern Java development; prefer **ArrayList** with external synchronization if needed.

**Common List Operations:**

Here are some common methods for adding, removing, and changing elements in lists:

* **Adding Elements:**
  + **add(E e)**: Appends the specified element to the end of the list.
  + **add(int index, E element)**: Inserts the specified element at the specified position in the list.
* **Removing Elements:**
  + **remove(Object o)**: Removes the first occurrence of the specified element from the list.
  + **remove(int index)**: Removes the element at the specified position in the list.
  + **clear()**: Removes all elements from the list.
* **Changing Elements:**
  + **set(int index, E element)**: Replaces the element at the specified position in the list with the specified element.
* **Other Operations:**
  + **size()**: Returns the number of elements in the list.
  + **get(int index)**: Returns the element at the specified position in the list.

These methods are part of the **List** interface and are implemented by its various concrete classes, including **ArrayList** and **LinkedList**. The choice of which list implementation to use depends on the specific requirements and performance characteristics of your application.

**KT0104 List and ArrayList**

* **Iterating around elements**
* **Choosing iteration approach for printing and display**
* **Type safety and removing integers**
* **Sorting - introduction to collections sort static**
* **Sorting - implementing comparable interface**

It seems like you're interested in working with lists in Java, particularly **List** and **ArrayList**. I'll provide you with information on the topics you've mentioned:

1. **Iterating around elements:**
   * You can iterate through elements in a **List** or **ArrayList** using various approaches, such as a traditional for loop, enhanced for loop (for-each loop), or using iterators.
   * Example using an enhanced for loop:

List<String> myList = new ArrayList<>(); myList.add("Item 1"); myList.add("Item 2"); for (String item : myList) { System.out.println(item); }

1. **Choosing iteration approach for printing and display:**
   * The choice of iteration approach depends on your specific use case. Use an enhanced for loop when you want a simple way to iterate through all elements. Use iterators when you need more control or want to remove elements while iterating.
2. **Type safety and removing integers:**
   * To ensure type safety, you can specify the type of elements a list can contain by using generics. For example:

List<Integer> integerList = new ArrayList<>(); integerList.add(1); integerList.add(2);

* + To remove integers from the list, you can use the **remove** method:

integerList.remove(Integer.valueOf(2)); // Removes the integer 2 from the list

1. **Sorting - introduction to Collections.sort static method:**
   * You can use the **Collections.sort** method to sort a list of elements. This method takes a list and sorts it in ascending order based on the natural order of elements.

List<Integer> numbers = new ArrayList<>(); numbers.add(5); numbers.add(2); numbers.add(8); Collections.sort(numbers);

1. **Sorting - implementing Comparable interface:**
   * To sort custom objects or non-comparable objects in a list, you can implement the **Comparable** interface and override the **compareTo** method.

public class Student implements Comparable<Student> { private String name; private int id; // Constructor, getters, and setters @Override public int compareTo(Student other) { // Implement comparison logic here (e.g., compare by ID) return Integer.compare(this.id, other.id); } } // Sorting a list of Student objects List<Student> students = new ArrayList<>(); Collections.sort(students);

These are some basic concepts related to working with **List** and **ArrayList** in Java, iterating through elements, ensuring type safety, and sorting elements. Depending on your specific use case, you can tailor these concepts to meet your requirements.

**KT0105 Data Structures**

* **Array, LinkedList and Hashing**
* **Tree - Sorted Order**

It looks like you're interested in data structures in Java, particularly arrays, linked lists, hashing, and sorted trees. Let's explore these topics:

1. **Array, LinkedList, and Hashing:**
   * **Array:** An array is a fixed-size data structure that can store elements of the same type. In Java, arrays have a fixed length, and you need to specify the size when creating them. Accessing elements in an array is fast (O(1)), but inserting or removing elements may require shifting elements, resulting in O(n) time complexity.

int[] arr = new int[5]; arr[0] = 1; arr[1] = 2; // ...

* + **LinkedList:** A linked list is a dynamic data structure where elements (nodes) are connected in a linear order. In Java, you can use the **LinkedList** class from the **java.util** package. Insertion and removal of elements are efficient (O(1) if you have a reference to the node), but accessing elements requires traversing the list (O(n) in the worst case).

LinkedList<String> linkedList = new LinkedList<>(); linkedList.add("Node 1"); linkedList.add("Node 2"); // ...

* + **Hashing:** Hashing is a technique used to map keys to values, allowing for efficient retrieval and storage of data. In Java, you can use HashMap, HashSet, or other hash-based data structures from the **java.util** package.

HashMap<String, Integer> hashMap = new HashMap<>(); hashMap.put("Key1", 1); hashMap.put("Key2", 2); // ...

1. **Tree - Sorted Order:**
   * A sorted tree, commonly referred to as a binary search tree (BST), is a hierarchical data structure in which each node has at most two children. In a BST, the left subtree contains nodes with values less than the parent node, and the right subtree contains nodes with values greater than the parent node.
   * Java provides the **TreeSet** and **TreeMap** classes in the **java.util** package, which are based on balanced binary search trees (Red-Black Trees). These data structures maintain elements in sorted order based on their natural order (or a custom comparator).

TreeSet<Integer> treeSet = new TreeSet<>(); treeSet.add(3); treeSet.add(1); treeSet.add(2); // Elements will be stored and retrieved in sorted order (1, 2, 3)

* + To implement a custom sorting order, you can provide a comparator when constructing a **TreeSet** or **TreeMap**.

TreeSet<String> customOrderSet = new TreeSet<>(new MyCustomComparator()); customOrderSet.add("Apple"); customOrderSet.add("Banana"); customOrderSet.add("Cherry");

These are fundamental data structures and concepts in Java for working with collections of data efficiently. Depending on your use case, you can choose the appropriate data structure to suit your needs.

**KT0106 Set Interface**

* **Hands-on - HashSet, LinkedHashSet and TreeSet**
* **Find Unique Characters in a List**
* **TreeSet - Methods from NavigableSet - floor, lower, upper, subSet, head**
* **Queue Interface - Process Elements in Order**

It appears you're interested in the Java **Set** interface and related data structures like **HashSet**, **LinkedHashSet**, and **TreeSet**, as well as some practical tasks you can perform with these data structures. Additionally, you're interested in understanding methods from the **NavigableSet** interface, as well as working with the **Queue** interface. Let's dive into these topics:

1. **Hands-on with HashSet, LinkedHashSet, and TreeSet:**
   * **HashSet**: A **HashSet** is an unordered collection that does not allow duplicate elements. It uses a hash table for efficient storage and retrieval.
   * **LinkedHashSet**: A **LinkedHashSet** is similar to a **HashSet**, but it maintains the order in which elements were inserted.
   * **TreeSet**: A **TreeSet** is an ordered collection that stores elements in sorted order. It uses a Red-Black Tree for efficient sorting and retrieval.

HashSet<String> hashSet = new HashSet<>(); LinkedHashSet<String> linkedHashSet = new LinkedHashSet<>(); TreeSet<String> treeSet = new TreeSet<>();

1. **Finding Unique Characters in a List:**
   * To find unique characters in a list, you can use a **Set** to store the characters while iterating through the list. Since **Set** does not allow duplicates, you can add each character to the **Set**. After processing the entire list, the **Set** will only contain unique characters.

List<Character> charList = Arrays.asList('a', 'b', 'a', 'c', 'b'); Set<Character> uniqueChars = new HashSet<>(); for (Character c : charList) { uniqueChars.add(c); }

1. **TreeSet - Methods from NavigableSet (e.g., floor, lower, upper, subSet, head):**
   * **NavigableSet** is an interface that extends the **SortedSet** interface and provides additional methods for navigating and manipulating the set. Here are some examples:

TreeSet<Integer> treeSet = new TreeSet<>(Arrays.asList(1, 2, 3, 4, 5)); Integer floorValue = treeSet.floor(3); // Returns 3 or the greatest element less than or equal to 3. Integer lowerValue = treeSet.lower(3); // Returns 2 or the greatest element strictly less than 3. Integer ceilingValue = treeSet.ceiling(3); // Returns 3 or the smallest element greater than or equal to 3. Integer higherValue = treeSet.higher(3); // Returns 4 or the smallest element strictly greater than 3. // SubSet - Returns a subset of elements within a specified range. NavigableSet<Integer> subSet = treeSet.subSet(2, true, 4, true); // [2, 3, 4] (inclusive) // HeadSet - Returns a set of all elements less than the specified value. NavigableSet<Integer> headSet = treeSet.headSet(4, true); // [1, 2, 3, 4] (inclusive)

1. **Queue Interface - Processing Elements in Order:**
   * The **Queue** interface in Java represents a collection designed for processing elements in a specific order, such as FIFO (First-In-First-Out) or priority order.
   * You can use classes like **LinkedList** or **PriorityQueue** that implement the **Queue** interface to perform queue-related operations like **add**, **poll**, **peek**, etc.

Queue<String> queue = new LinkedList<>(); queue.add("Item 1"); queue.add("Item 2"); String item = queue.poll(); // Removes and returns the first item ("Item 1")

These concepts should help you understand and work with the **Set**, **NavigableSet**, and **Queue** interfaces in Java effectively.

**KT0107 Introduction to PriorityQueue - Basic Methods and Customized Priority**

In Java, **PriorityQueue** is an implementation of the **Queue** interface that allows you to maintain elements in a queue based on their priority. Elements with higher priority are dequeued before elements with lower priority. Here's an introduction to **PriorityQueue**, covering its basic methods and how to customize the priority of elements:

1. **Creating a PriorityQueue:**

You can create a **PriorityQueue** by specifying the element type and, optionally, providing a custom comparator to define the priority order.

PriorityQueue<Integer> priorityQueue = new PriorityQueue<>();

1. **Basic Methods:**

**PriorityQueue** provides several essential methods for working with elements:

* + **add(E e)** or **offer(E e)**: Adds an element to the priority queue.

priorityQueue.offer(5); priorityQueue.offer(2);

* + **poll()**: Retrieves and removes the element with the highest priority (the head of the queue).

Integer highestPriority = priorityQueue.poll(); // Retrieves and removes the highest-priority element.

* + **peek()**: Retrieves the element with the highest priority without removing it.

Integer highestPriority = priorityQueue.peek(); // Retrieves the highest-priority element without removing it.

* + **size()**: Returns the number of elements in the priority queue.

int queueSize = priorityQueue.size();

1. **Customizing Priority:**

By default, **PriorityQueue** orders elements in their natural order (e.g., numbers in ascending order). However, you can customize the priority by providing a custom comparator during **PriorityQueue** creation. The comparator determines how elements are prioritized.

PriorityQueue<Integer> customPriorityQueue = new PriorityQueue<>(Comparator.reverseOrder());

In this example, we've used **Comparator.reverseOrder()** to reverse the natural order and prioritize elements in descending order.

You can also create a custom comparator for more complex priority rules. For example, to prioritize elements based on their length (for strings):

PriorityQueue<String> lengthPriorityQueue = new PriorityQueue<>(Comparator.comparing(String::length));

Here, we're using a comparator that compares strings based on their lengths.

Remember that you need to ensure the elements you add to the **PriorityQueue** are comparable or that you provide a suitable comparator. Otherwise, you may encounter **ClassCastException** at runtime.

1. **Custom Priority Objects:**

You can use custom objects in a **PriorityQueue** by implementing the **Comparable** interface or by providing a custom comparator. This allows you to define how objects are prioritized.

For example, if you have a **Task** class with a custom priority field:

class Task implements Comparable<Task> { private String name; private int priority; // Constructor, getters, setters @Override public int compareTo(Task other) { return Integer.compare(this.priority, other.priority); } }

You can create a **PriorityQueue** of tasks and have them prioritized based on their custom priority values:

PriorityQueue<Task> taskQueue = new PriorityQueue<>(); taskQueue.add(new Task("Task 1", 2)); taskQueue.add(new Task("Task 2", 1)); Task highestPriorityTask = taskQueue.poll(); // Retrieves and removes the highest-priority task based on priority value.

Here, tasks are ordered by their priority values due to the custom **compareTo** method.

These are the fundamental concepts of working with **PriorityQueue** in Java. It's a versatile data structure for managing elements based on their priority, and you can customize it to suit various use cases by providing appropriate comparators or implementing the **Comparable** interface for custom objects.

**KT0108 Map Interface**

* **Key and Value**
* **Implementations - HashMap, HashTable, LinkedHashMap**
* **Basic Operations**
* **Comparison - HashMap vs LinkedHashMap vs TreeMap**
* **Count occurrences of characters and words**
* **TreeMap - Methods from NavigableMap - floorKey, higherKey, firstEntry**

The **Map** interface in Java represents a collection of key-value pairs. It allows you to store, retrieve, and manipulate data where each element is associated with a unique key. Here's an introduction to the **Map** interface, its implementations (such as **HashMap**, **Hashtable**, and **LinkedHashMap**), basic operations, and comparisons between these implementations:

1. **Key and Value:**

In a **Map**, each entry consists of a key and a corresponding value. The key is used to access the associated value. Keys are unique within the map, meaning that you cannot have duplicate keys.

1. **Implementations - HashMap, Hashtable, LinkedHashMap:**
   * **HashMap**: **HashMap** is an implementation of the **Map** interface that provides efficient key-value storage and retrieval. It does not guarantee any specific order of elements.
   * **Hashtable**: **Hashtable** is a legacy implementation of **Map**, similar to **HashMap**. It is synchronized, making it thread-safe but potentially slower in multi-threaded environments.
   * **LinkedHashMap**: **LinkedHashMap** extends **HashMap** and maintains the order of elements based on their insertion order.

Map<String, Integer> hashMap = new HashMap<>(); Map<String, Integer> hashtable = new Hashtable<>(); Map<String, Integer> linkedHashMap = new LinkedHashMap<>();

1. **Basic Operations:**

Common operations you can perform on a **Map** include:

* + **put(K key, V value)**: Add a key-value pair to the map.
  + **get(K key)**: Retrieve the value associated with a key.
  + **containsKey(K key)**: Check if a key exists in the map.
  + **containsValue(V value)**: Check if a value exists in the map.
  + **remove(K key)**: Remove a key-value pair from the map.
  + **size()**: Get the number of key-value pairs in the map.

1. **Comparison - HashMap vs. LinkedHashMap vs. TreeMap:**
   * **HashMap**: Offers fast and efficient retrieval of elements. Does not guarantee any specific order of elements.
   * **LinkedHashMap**: Maintains the order of elements based on insertion order. Slower than **HashMap** but useful when order matters.
   * **TreeMap**: Maintains elements in sorted order based on keys. Slower for insertion and retrieval compared to the other two, but provides sorted access.
2. **Count Occurrences of Characters and Words:**

You can use a **HashMap** to count the occurrences of characters or words in a text:

String text = "Hello, World!"; Map<Character, Integer> charCountMap = new HashMap<>(); for (char c : text.toCharArray()) { charCountMap.put(c, charCountMap.getOrDefault(c, 0) + 1); }

For counting words, split the text into words and use a **HashMap**:

String[] words = text.split("\\s+"); Map<String, Integer> wordCountMap = new HashMap<>(); for (String word : words) { wordCountMap.put(word, wordCountMap.getOrDefault(word, 0) + 1); }

1. **TreeMap - Methods from NavigableMap (floorKey, higherKey, firstEntry):**

**TreeMap** implements the **NavigableMap** interface, which provides methods for navigation and retrieval:

* + **floorKey(K key)**: Returns the greatest key less than or equal to the given key.
  + **higherKey(K key)**: Returns the least key strictly greater than the given key.
  + **firstEntry()**: Returns the first key-value mapping.

TreeMap<Integer, String> treeMap = new TreeMap<>(); treeMap.put(1, "One"); treeMap.put(2, "Two"); treeMap.put(4, "Four"); Integer floorKey = treeMap.floorKey(3); // Returns 2 Integer higherKey = treeMap.higherKey(2); // Returns 4 Map.Entry<Integer, String> firstEntry = treeMap.firstEntry(); // Returns (1, "One")

These are the fundamentals of the **Map** interface and its common implementations (**HashMap**, **Hashtable**, and **LinkedHashMap**). Understanding these concepts will enable you to work with key-value pairs efficiently and perform various operations on map data structures.

**Internal Assessment Criteria and Weight**

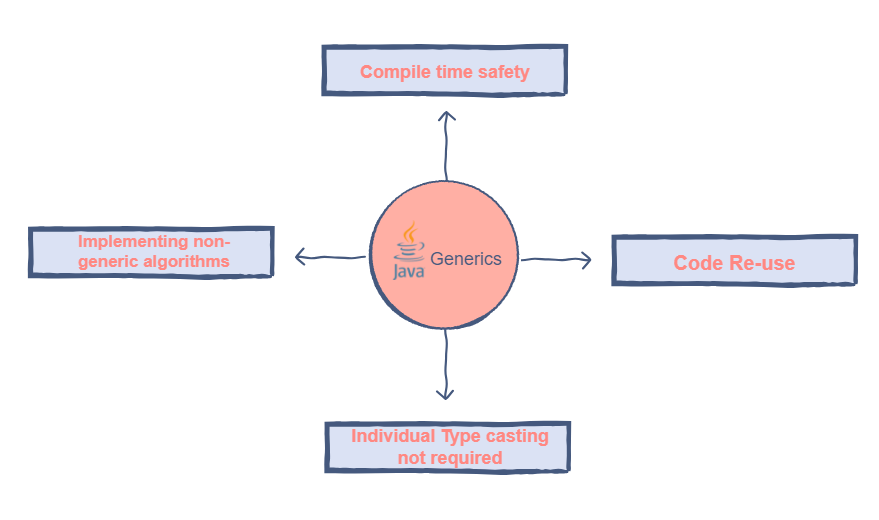
1. IAC0101 Definitions, functions and features of Collections in Java are stated.

**Session 2:** **KM-04-KT02: Introduction to Generics in Java Programming**

Topic elements to be covered include:

* KT0201 Definition
* KT0202 Purpose
* KT0203 Features
* KT0204 Uses
* KT0205 Benefits
* KT0206 Restrictions with extends and generic methods
* KT0207 Symbols

**KT0201 Definition**

****

Generics in Java programming provide a way to create classes, interfaces, and methods that operate on typed parameters, allowing you to write code that can work with different data types while maintaining type safety. The primary purpose of generics is to enable you to write reusable and type-safe code.

Here's a breakdown of key concepts related to generics in Java:

1. **Type Parameter (T):** In generic classes, interfaces, and methods, you define one or more type parameters within angle brackets (**<>**). These type parameters act as placeholders for actual data types. Commonly used type parameter names include **T** for a type, **E** for an element, and **K** and **V** for key and value in map-like structures.

public class MyGenericClass<T> { // Class code that can use the type parameter T }

1. **Type Argument:** When you create an instance of a generic class or call a generic method, you specify the actual data type that the type parameter should represent. These actual data types are called type arguments.

MyGenericClass<String> myStringInstance = new MyGenericClass<>(); MyGenericClass<Integer> myIntegerInstance = new MyGenericClass<>();

In the above examples, **String** and **Integer** are type arguments.

1. **Type Safety:** Generics provide compile-time type checking, which means that the Java compiler verifies that the types used in generic code are consistent and compatible. This helps catch type-related errors at compile time rather than at runtime.
2. **Reusability:** Generics promote code reusability by allowing you to write generic code that can work with different data types. This reduces the need for code duplication and makes your code more maintainable.
3. **Collections Framework:** One of the most common uses of generics in Java is within the Java Collections Framework. Generics allow you to create type-safe collections like **List<T>**, **Set<T>**, and **Map<K, V>**, where **T**, **K**, and **V** are type parameters representing the element or key-value types.

List<String> stringList = new ArrayList<>(); Map<Integer, String> idToNameMap = new HashMap<>();

1. **Method Generics:** In addition to generic classes and interfaces, you can also define generic methods that allow type parameters to be declared and used within the method.

public <T> T myGenericMethod(T value) { // Method code that can use the type parameter T return value; }

1. **Wildcards:** Generics support the use of wildcard types (**?**) to represent unknown types or to provide flexibility when working with generic collections.

public void processList(List<?> list) { // This method can accept a List of any type. }

In summary, generics in Java programming provide a mechanism for creating flexible, reusable, and type-safe code that can work with different data types. They are a fundamental feature of the Java language, widely used in libraries, frameworks, and applications to enhance code quality and maintainability.

**KT0202 Purpose**

Generics in Java programming serve several important purposes, enhancing the flexibility, type safety, and reusability of code. Here are the primary purposes of using generics in Java:

* **Type Safety:** One of the main purposes of generics is to provide type safety to Java code. Generics enable the compiler to perform type checking at compile time, ensuring that the correct data types are used consistently throughout the code. This prevents runtime type-related errors, such as ClassCastException, by catching them at compile time.
* **Code Reusability:** Generics promote code reusability by allowing you to write generic code that can work with a wide range of data types. Instead of writing separate code for each data type, you can create a single generic class, interface, or method that adapts to different types. This reduces code duplication and simplifies maintenance.
* **Flexibility:** Generics provide flexibility by allowing you to create classes, interfaces, and methods that can work with different types while maintaining type safety. This flexibility makes it easier to design libraries and APIs that are more adaptable to various use cases.
* **Collections Framework:** Generics are extensively used in the Java Collections Framework. They allow you to create type-safe collections (e.g., **List<T>**, **Set<T>**, **Map<K, V>**) that store and manipulate elements of specific types. This ensures that you can work with collections of any data type without casting and with compile-time type checking.
* **Custom Data Structures:** Generics enable the creation of custom data structures that can hold elements of any type while maintaining type safety. For example, you can create a generic stack, queue, or linked list that works with different data types.
* **Eliminating Casting:** Before generics, developers had to use casting extensively when working with collections, which could lead to runtime errors. Generics eliminate the need for casting because the compiler ensures that you work with the correct types.
* **Improved Readability:** Generics make code more readable and self-documenting. When you see a method or class with a generic type parameter, it's clear what kind of data it operates on, enhancing code understanding.
* **Compiler Warnings:** Without generics, using raw types (e.g., **List** instead of **List<T>**) would result in compiler warnings about unchecked operations. Generics help eliminate these warnings, indicating potential type-related issues.
* **API Design:** Generics play a crucial role in designing flexible and robust APIs. They allow API creators to define methods and classes that can be used with different types while ensuring type safety for API consumers.

In summary, the purpose of generics in Java programming is to enhance the safety, reusability, and flexibility of code. They are an essential feature of the language and are widely used in libraries, frameworks, and applications to write more robust and maintainable software. Generics enable developers to create versatile and type-safe code that adapts to various data types, reducing the likelihood of errors and simplifying code maintenance.

**KT0203 Features**

Generics in Java programming provide several important features that enhance code flexibility, type safety, and reusability. These features make generics a fundamental and powerful aspect of the Java language. Here are the key features of generics in Java:

1. **Type Safety:**
   * Generics provide compile-time type checking, ensuring that you use the correct data types consistently throughout your code. This prevents runtime type-related errors, such as ClassCastException.
   * Type safety is a fundamental feature of generics, making it safer to work with collections and algorithms that operate on generic types.
2. **Code Reusability:**
   * Generics enable you to write generic code that can work with different data types. You can create a single generic class, interface, or method that adapts to various types, reducing code duplication.
   * Code reusability simplifies maintenance and promotes efficient development.
3. **Flexibility:**
   * Generics allow you to create classes, interfaces, and methods that can operate on different types while maintaining type safety. This flexibility makes it easier to design libraries and APIs that can adapt to various use cases.
   * You can write more adaptable and extensible code with generics.
4. **Parameterized Types:**
   * Generics introduce parameterized types, where you define type parameters within angle brackets (**<>**) when creating generic classes or methods.
   * Example: **List<T>** where **T** is the type parameter representing the element type.
5. **Wildcards:**
   * Generics support wildcard types (**?**) to represent unknown types or provide flexibility when working with generic collections.
   * Wildcards are used in method signatures to accept arguments of unspecified types.
   * Example: **public void processList(List<?> list)**.
6. **Upper Bounded Wildcards:**
   * You can specify upper bounds for wildcards to restrict the types that can be used as arguments.
   * Example: **public void processList(List<? extends Number> list)** accepts a list of any type that is a subtype of **Number**.
7. **Lower Bounded Wildcards:**
   * Lower bounded wildcards allow you to specify a lower bound for wildcards, restricting the types that can be used as arguments.
   * Example: **public void addNumbers(List<? super Integer> list)** accepts a list of any type that is a supertype of **Integer**.
8. **Generic Methods:**
   * Generics can be used in methods, allowing you to declare and use type parameters within methods independently of class-level type parameters.
   * Example: **public <T> T myGenericMethod(T value)**.
9. **Type Erasure:**
   * Generics use type erasure, which means that type parameters are erased (replaced with their bounds) during compilation to maintain backward compatibility with pre-generic Java code.
   * Type erasure ensures that generics do not impact runtime performance.
10. **Bounded Type Parameters:**
    * You can specify bounds for type parameters, restricting the types that can be used.
    * Example: **public <T extends Number> void performOperation(T value)** ensures that **T** is a subtype of **Number**.
11. **Generic Interfaces and Classes:**
    * Generics can be applied to interfaces and classes, allowing you to create parameterized interfaces and classes that work with generic types.
    * Example: **public interface MyGenericInterface<T> { /\* ... \*/ }**.
12. **Generic Inheritance:**
    * Generics support inheritance, allowing generic classes and interfaces to extend or implement other generic or non-generic types.
13. **Diamond Operator (<>):**
    * The diamond operator simplifies the instantiation of generic classes by allowing you to omit type arguments when the compiler can infer them from the context.
    * Example: **List<String> myList = new ArrayList<>();**.

Generics in Java provide a rich set of features that enable you to write type-safe and reusable code, adapt to different data types, and design flexible APIs and data structures. These features are widely used in the Java ecosystem to create robust and efficient software.

**KT0204 Uses**

Generics in Java programming are widely used to enhance code flexibility, type safety, and reusability. They offer numerous benefits, and you can find them employed in various contexts throughout Java applications and libraries. Here are some common uses of generics in Java programming:

1. **Collections Framework:**
   * Generics are extensively used in the Java Collections Framework (**java.util** package) to create type-safe collections. For example, **List<T>**, **Set<T>**, and **Map<K, V>** allow you to store and manipulate elements of specific types without casting.
   * Example: **List<String>**, **Set<Integer>**, **Map<String, Double>**.
2. **Custom Data Structures:**
   * Generics enable the creation of custom data structures that can hold elements of any type while maintaining type safety. Developers can create generic stacks, queues, linked lists, and more.
   * Example: **Stack<T>**, **Queue<E>**, **LinkedList<E>**.
3. **Method Parameterization:**
   * Generics are used to parameterize methods, allowing developers to create methods that work with various types of input data while ensuring type safety.
   * Example: **public <T> T findMax(List<T> list)**.
4. **Generics in Classes:**
   * Developers use generics in class definitions to create generic classes that can work with different types.
   * Example: **public class Box<T> { /\* ... \*/ }**.
5. **API Design:**
   * Generics play a crucial role in designing flexible and reusable APIs. They enable API creators to define methods and classes that can be used with different data types while ensuring type safety for API consumers.
   * Example: The **java.util.function** package provides generic functional interfaces like **Predicate<T>**, **Function<T, R>**, and **Consumer<T>**.
6. **Collections Algorithms:**
   * Generics are used in collection algorithms like **Collections.sort** and **Collections.max** to perform operations on collections with different element types.
   * Example: Sorting a list of **Student** objects by their grades.
7. **Generics in I/O:**
   * Generics are used when reading and writing data from/to streams, files, and databases. This allows developers to handle data of different types while maintaining type safety.
   * Example: Using **BufferedReader<T>** to read lines from a file and parse them into specific types.
8. **Event Handling:**
   * Generics are used in event-driven programming to create type-safe event listeners and handlers. This allows event-driven systems to handle various types of events while ensuring that the appropriate handlers are invoked.
   * Example: Implementing a generic event listener to handle events of different types.
9. **Concurrency and Thread Safety:**
   * Generics can be used to create thread-safe data structures and containers that work with different data types. This is important in multi-threaded applications where data integrity and type safety are crucial.
   * Example: Creating a generic thread-safe cache.
10. **Dependency Injection:**
    * Generics are used in dependency injection frameworks (e.g., Spring) to inject dependencies of specific types into components. This allows for flexible configuration while maintaining type safety.
    * Example: Injecting a **List<Customer>** as a dependency.
11. **Custom Containers:**
    * Developers use generics to create custom container classes and data structures, such as trees, graphs, and caches, that can store and manipulate data of different types while ensuring type safety.
    * Example: Implementing a generic binary search tree.
12. **Reflection and Introspection:**
    * Generics are used in reflection and introspection to discover and manipulate the generic type information of classes and objects at runtime.
    * Example: Analyzing the generic type parameters of a class using reflection.

These are just some of the many uses of generics in Java programming. Generics are a powerful feature that simplifies code, improves type safety, and enhances code reusability in a wide range of applications and libraries.

**KT0205 Benefits**

Generics in Java programming offer several significant benefits that contribute to code quality, reusability, and type safety. Understanding and utilizing generics can lead to more efficient and robust Java applications. Here are the key benefits of using generics in Java programming:

1. **Type Safety:**
   * Generics provide compile-time type checking, ensuring that the correct data types are used consistently throughout the code. This prevents runtime type-related errors, such as **ClassCastException**, making code more reliable and less error-prone.
2. **Code Reusability:**
   * Generics enable you to write generic code that can work with different data types. Instead of writing separate code for each data type, you can create a single generic class, interface, or method that adapts to various types. This reduces code duplication and simplifies maintenance.
3. **Flexibility:**
   * Generics allow you to create classes, interfaces, and methods that can operate on different types while maintaining type safety. This flexibility makes it easier to design libraries and APIs that can adapt to various use cases.
4. **Improved Readability:**
   * Code that uses generics is more self-documenting and easier to understand. When you see a method or class with a generic type parameter, it's clear what kind of data it operates on, enhancing code understanding.
5. **Efficient Data Structures:**
   * Generics enable the creation of type-safe data structures and collections, such as lists, sets, and maps, that can store and manipulate elements of specific types without the need for casting. This improves the efficiency and readability of code.
6. **Compile-Time Checking:**
   * Generics catch type-related errors at compile time, which is preferable to finding such errors at runtime. This helps ensure that code behaves as expected and prevents unexpected crashes in production.
7. **API Design:**
   * Generics play a crucial role in designing flexible and robust APIs. They allow API creators to define methods and classes that can be used with different data types while ensuring type safety for API consumers.
8. **Reduced Casting:**
   * Generics eliminate the need for excessive casting when working with collections and data structures. This results in cleaner and more concise code, reducing the likelihood of casting-related errors.
9. **Better Abstraction:**
   * Generics enable developers to create more abstract and reusable code. By parameterizing types, you can write code that focuses on the algorithm or logic rather than the specifics of data types.
10. **API Evolution:**
    * Generics make it easier to evolve APIs over time. You can introduce new data types and features without breaking existing code that relies on generic interfaces and classes.
11. **Safety in Concurrent Programming:**
    * Generics facilitate the development of thread-safe data structures and containers. With type-safe collections, developers can avoid concurrency issues that may arise from sharing data among threads.
12. **Reduced Casting and Errors:**
    * By providing type information at compile time, generics eliminate the need for casting data types when using collections and algorithms. This not only reduces code clutter but also minimizes the risk of runtime errors.
13. **Cleaner Code:**
    * Generics lead to cleaner, more concise, and more maintainable code. They enable developers to express the intent of their code without the noise of casting and type conversions.
14. **Compatibility with Legacy Code:**
    * Generics are designed with type erasure, allowing them to work seamlessly with pre-generic Java code. This backward compatibility ensures that older libraries and codebases can still interoperate with newer generic code.
15. **Improved Debugging:**
    * The compile-time type checking provided by generics can help developers catch and fix type-related issues during development, reducing debugging efforts and improving code quality.

Overall, the benefits of using generics in Java programming are substantial, making code safer, more reusable, and easier to maintain. Generics are a fundamental feature of the language and are widely employed in libraries, frameworks, and applications to create robust and efficient software.

**KT0206 Restrictions with extends and generic methods**

In Java generics, the use of the **extends** keyword in type parameter declarations and generic methods allows for certain restrictions and behaviors that are important to understand. Here, we'll discuss the restrictions associated with **extends** in type parameters and some aspects of generic methods:

1. **Restrictions with extends in Type Parameters:**

When using **extends** in type parameters, you indicate that the type parameter must be a subtype (or the same type) of the specified bound. Here are some key restrictions and behaviors:

* + **Upper Bounded Wildcards (? extends T):**
    - You can use **? extends T** to create upper-bounded wildcards. This allows you to accept arguments of any type that is a subtype of **T** or **T** itself.
    - For example, if you have a method that takes a **List** of elements and you want to allow lists of any type that is a subtype of **Number**, you can use **List<? extends Number>** as the parameter type.
  + **Use in Method Signatures:**
    - You can use **extends** in method signatures, enabling you to create methods that accept arguments of a bounded type.
    - Example: **public <T extends Number> void processNumbers(List<T> numbers)** accepts a list of any type that is a subtype of **Number**.
  + **Read-Only Access:**
    - With upper-bounded wildcards, you can read elements from a collection but cannot add elements to it. This ensures that you don't violate type safety.
    - Example: You can retrieve elements from a **List<? extends Number>**, but you cannot add new elements to it.
  + **Lack of Precision:**
    - Using **extends** with wildcards provides flexibility but can lead to a loss of precision. You can't assume much about the exact type of elements you are working with within the method.
    - Example: If you have **List<? extends Number>**, you can't invoke methods specific to **Integer** or **Double** on the elements without casting.

1. **Generic Methods:**

In addition to using **extends** with type parameters, you can create generic methods that have their own type parameters. Some points to note:

* + **Type Inference:**
    - Java's type inference mechanism allows the compiler to automatically infer the type parameters of generic methods in most cases. You don't always need to specify type arguments explicitly.
    - Example: **public <T> T findMax(List<T> elements)** can be called without specifying **T** if the compiler can infer it from the method arguments.
  + **Generic and Non-Generic Methods Overloading:**
    - You can overload methods with generic and non-generic versions, and the compiler will resolve the correct method to call based on the argument types.
    - Example: You can have both **public <T> T findMax(List<T> elements)** and **public int findMax(List<Integer> elements)** in the same class.
  + **Wildcards in Method Parameters:**
    - You can use wildcards (**?**) in method parameters to create more flexible generic methods. Wildcards allow you to accept arguments of unspecified types.
    - Example: **public void processList(List<?> list)** accepts a list of any type.
  + **Generic Method Return Types:**
    - Generic methods can have return types that depend on the type parameters declared in the method signature.
    - Example: **public <T> List<T> filter(List<T> elements, Predicate<T> predicate)** returns a list of the same type as the input list.

In summary, when using **extends** in type parameters and creating generic methods:

* Upper-bounded wildcards (**? extends T**) allow you to work with subtypes of a specific type but have limited write access.
* Generic methods provide flexibility in working with different types and can be overloaded alongside non-generic methods.
* Type inference and wildcards make code more flexible and expressive.

However, the flexibility provided by **extends** and generic methods can sometimes lead to less precise code, and you should be aware of the trade-offs when using them in your Java applications.

**KT0207 Symbols**

Generics in Java programming use various symbols and syntax to denote type parameters, wildcards, and other related concepts. Understanding these symbols is crucial for working effectively with generics. Here are the key symbols and syntax used in Java generics:

1. **Type Parameter Angle Brackets (<>):**
   * Type parameters are enclosed in angle brackets when declaring generic classes, interfaces, and methods.
   * Example: **class MyGenericClass<T>**, **interface MyGenericInterface<E>**, **public <T> void myGenericMethod(T value)**.
2. **Wildcards (?):**
   * Wildcards are used to represent unknown types or provide flexibility when working with generic types.
   * Example: **List<?>**, **public void processList(List<? extends Number> list)**.
3. **Upper Bounded Wildcard (? extends T):**
   * Denotes that a wildcard can accept types that are subtypes of **T** or **T** itself.
   * Example: **List<? extends Number>** can accept **List<Integer>**, **List<Double>**, or other subtypes of **Number**.
4. **Lower Bounded Wildcard (? super T):**
   * Denotes that a wildcard can accept types that are supertypes of **T**.
   * Example: **List<? super Integer>** can accept **List<Number>**, **List<Object>**, or other supertypes of **Integer**.
5. **Generic Class Declaration (<T>):**
   * Indicates that a class is generic and can accept a type parameter **T**.
   * Example: **class MyGenericClass<T>**.
6. **Generic Interface Declaration (<E>):**
   * Indicates that an interface is generic and can accept a type parameter **E**.
   * Example: **interface MyGenericInterface<E>**.
7. **Generic Method Declaration (<T>):**
   * Denotes that a method is generic and can accept a type parameter **T**.
   * Example: **public <T> void myGenericMethod(T value)**.
8. **Type Erasure (T after compilation):**
   * After compilation, type parameters are erased (replaced with their bounds) due to type erasure. However, they are retained as metadata for reflection purposes.
   * Example: **List<String>** becomes **List** after type erasure.
9. **Diamond Operator (<>):**
   * Simplifies the instantiation of generic classes by allowing you to omit type arguments when the compiler can infer them from the context.
   * Example: **List<String> myList = new ArrayList<>();**.
10. **Generic Inheritance (<T> in superclass and subclass):**
    * Indicates that a generic class or interface is extended or implemented with the same or different type parameters.
    * Example: **class MyChildClass<T> extends MyGenericClass<T>**.
11. **Generic and Non-Generic Method Overloading:**
    * Allows both generic and non-generic methods with the same name in a class.
    * Example: **public <T> T findMax(List<T> elements)** and **public int findMax(List<Integer> elements)**.
12. **Type Argument (T, E, etc.):**
    * Represents the actual data type provided as an argument to a generic class, interface, or method.
    * Example: **MyGenericClass<String>**, **MyGenericInterface<Integer>**, **myGenericMethod(42)**.

These symbols and syntax elements are fundamental to working with generics in Java. They enable you to create flexible and type-safe code that can adapt to various data types and use cases while maintaining compile-time type checking.

**Internal Assessment Criteria and Weight**

1. IAC0201Definitions, functions, features, uses and benefits of Java generics are stated

**Session 3:** **KM-04-KT03: Functional programming in Java**

Topic elements to be covered include:

* KT0301 Functional Programming:
  + - The value of functions
    - First Example with Function as Parameter
    - Exercise - Loop a List of Numbers
    - Filtering - Exercises to print odd and even n
    - Collect - Sum of Numbers in a List
    - Functional Programming vs Structural Programming - A Quick Comparison
    - Functional Programming Terminology - Lambda Expression, Stream and Operations
* KT0302 Stream Intermediate Operations:
  + - Sort, Distinct, Filter and Map
* KT0303 Stream Terminal Operations
  + - 1 - max operation with Comparator
    - 2 - min, collect to List
* KT0304 Optional class in Java - An Introduction
* KT0305 Behind the Screens with Functional Interfaces
  + - Implement Predicate Interfaces
    - Implement Consumer Interfaces
    - Implement Function Interfaces
* KT0306 Simplify Functional Programming code with Method References
* KT0307Functions are First Class Citizens

**KT0301 Functional Programming:**

* **The value of functions**
* **First Example with Function as Parameter**
* **Exercise - Loop a List of Numbers**
* **Filtering - Exercises to print odd and even n**
* **Collect - Sum of Numbers in a List**
* **Functional Programming vs Structural Programming - A Quick Comparison**
* **Functional Programming Terminology - Lambda Expression, Stream and Operations**

Functional programming is a programming paradigm that focuses on treating computation as the evaluation of mathematical functions and avoiding changing state and mutable data. In Java, functional programming is supported through features introduced in Java 8, such as lambdas, streams, and functional interfaces. Let's explore some key concepts and examples related to functional programming in Java:

1. **The Value of Functions:**
   * In functional programming, functions are treated as first-class citizens. This means functions can be assigned to variables, passed as arguments to other functions, and returned as results from functions.
   * Functions are considered pure if they have no side effects and produce the same output for the same input, which leads to predictability and easier debugging.
2. **First Example with Function as Parameter:**
   * In Java, you can pass functions as parameters to methods. For example, you can use the **forEach** method with a lambda expression to iterate over a list of elements and perform an action for each element.
   * Example:

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5); numbers.forEach(number -> System.out.println(number));

1. **Exercise - Loop a List of Numbers:**
   * You can create a method that accepts a list and a function as parameters and applies the function to each element in the list.
   * Example:

public static void loopList(List<Integer> list, Consumer<Integer> action) { for (Integer item : list) { action.accept(item); } }

1. **Filtering - Exercises to Print Odd and Even Numbers:**
   * Functional programming encourages filtering data without modifying it. You can use Java streams and lambda expressions to filter a list of numbers and print odd or even numbers.
   * Example:

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6); numbers.stream() .filter(number -> number % 2 == 0) // Filter even numbers .forEach(System.out::println);

1. **Collect - Sum of Numbers in a List:**
   * You can use the **collect** operation in Java streams to accumulate elements into a result. For example, you can calculate the sum of numbers in a list.
   * Example:

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5); int sum = numbers.stream() .reduce(0, Integer::sum); // Accumulate the sum System.out.println("Sum of numbers: " + sum);

1. **Functional Programming vs. Structural Programming - A Quick Comparison:**
   * In structural programming, code is organized around data and functions that operate on data. In contrast, functional programming focuses on using functions as first-class entities and avoiding mutable data.
   * Functional programming often leads to cleaner and more concise code and is well-suited for parallel processing.
2. **Functional Programming Terminology - Lambda Expression, Stream, and Operations:**
   * **Lambda Expression:** A lambda expression is an anonymous function that can be used to create instances of functional interfaces (interfaces with a single abstract method).
   * **Stream:** A stream is a sequence of elements that can be processed sequentially or in parallel. Streams support various operations, including map, filter, reduce, and collect.
   * **Operations:** In functional programming, operations like map, filter, and reduce are used to manipulate and transform data within streams.

Functional programming in Java empowers developers to write more expressive and maintainable code by focusing on functions, immutability, and predictable behavior. Java's support for lambda expressions and streams makes it easier to embrace functional programming principles and leverage the benefits it offers.

**KT0302 Stream Intermediate Operations:**

* **Sort, Distinct, Filter and Map**

Stream intermediate operations are operations that are performed on a stream of data, and they produce another stream as their result. These operations are typically applied in a chain and are executed lazily, which means they don't process data until a terminal operation is encountered. In this context, we'll explore several common stream intermediate operations: **sorted**, **distinct**, **filter**, and **map**.

1. **sorted Operation:**
   * The **sorted** operation is used to sort the elements of a stream based on their natural order or a provided comparator.
   * It returns a new stream containing the sorted elements.
   * Example using natural order:

List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David", "Eve"); List<String> sortedNames = names.stream() .sorted() .collect(Collectors.toList());

* + Example using a custom comparator:

List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David", "Eve"); List<String> sortedNames = names.stream() .sorted((a, b) -> b.compareTo(a)) // Reverse order .collect(Collectors.toList());

1. **distinct Operation:**
   * The **distinct** operation is used to eliminate duplicate elements from a stream.
   * It returns a new stream with unique elements in the same order as they were encountered in the original stream.
   * Example:

List<Integer> numbers = Arrays.asList(1, 2, 2, 3, 3, 4, 5, 5); List<Integer> distinctNumbers = numbers.stream() .distinct() .collect(Collectors.toList());

1. **filter Operation:**
   * The **filter** operation is used to select elements from a stream based on a given predicate (a boolean condition).
   * It returns a new stream containing only the elements that satisfy the predicate.
   * Example:

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10); List<Integer> evenNumbers = numbers.stream() .filter(n -> n % 2 == 0) .collect(Collectors.toList());

1. **map Operation:**
   * The **map** operation is used to transform elements of a stream from one type to another by applying a function to each element.
   * It returns a new stream containing the transformed elements.
   * Example:

List<String> names = Arrays.asList("Alice", "Bob", "Charlie"); List<Integer> nameLengths = names.stream() .map(String::length) .collect(Collectors.toList());

In summary, stream intermediate operations like **sorted**, **distinct**, **filter**, and **map** allow you to process and manipulate data in a declarative and functional style. These operations are performed on streams of data and help you filter, transform, and modify elements as needed before reaching a terminal operation like **collect**, **forEach**, or **reduce**. By using streams and these intermediate operations, you can write more expressive and concise code for data processing in Java.

**KT0303 Stream Terminal Operations**

* **1 - max operation with Comparator**
* **2 - min, collect to List**

Stream terminal operations are operations that process a stream and produce a result or a side effect. Unlike intermediate operations, terminal operations trigger the actual processing of elements in a stream. In this context, we'll explore two common stream terminal operations: **max** with a **Comparator** and **min** followed by collecting to a list using **collect**.

1. **max Operation with Comparator:**
   * The **max** terminal operation is used to find the maximum element in a stream based on a provided **Comparator**.
   * It returns an **Optional<T>** that may or may not contain the maximum element.
   * Example:

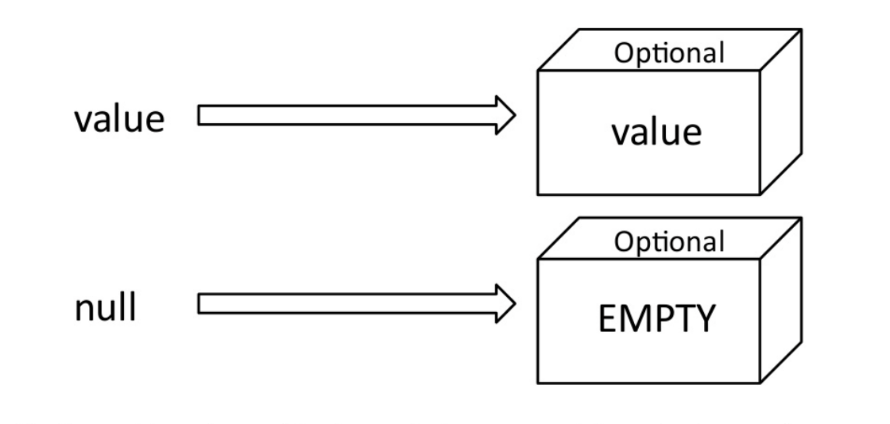
List<Integer> numbers = Arrays.asList(5, 2, 9, 1, 7, 3); Optional<Integer> maxNumber = numbers.stream() .max(Comparator.naturalOrder()); if (maxNumber.isPresent()) { System.out.println("Max number: " + maxNumber.get()); } else { System.out.println("No maximum element found."); }

1. **min Operation Followed by Collecting to a List:**
   * The **min** terminal operation is used to find the minimum element in a stream based on a provided **Comparator**.
   * It returns an **Optional<T>** that may or may not contain the minimum element.
   * After finding the minimum element, you can collect it to a list using the **collect** operation.
   * Example:

List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David"); Optional<String> shortestName = names.stream() .min(Comparator.comparing(String::length)); if (shortestName.isPresent()) { System.out.println("Shortest name: " + shortestName.get()); } else { System.out.println("No minimum element found."); }

These terminal operations allow you to find the maximum and minimum elements in a stream based on a provided **Comparator**. Additionally, you can collect the result into an **Optional** and further process it as needed. Terminal operations like **max** and **min** are essential for retrieving summary information from a stream and making decisions based on the elements it contains.

**KT0304 Optional class in Java - An Introduction**



The **Optional** class in Java is introduced in Java 8 to handle the absence of values more effectively and to reduce the occurrence of null pointer exceptions. It is part of the **java.util** package and provides a container object that may or may not contain a non-null value. The primary purpose of **Optional** is to help write more robust and expressive code when dealing with potentially missing or nullable values.

Here's an introduction to the **Optional** class and its key concepts:

1. **Why Optional?**
   * In traditional Java programming, when a method might return a value that could be null, developers often had to explicitly check for null and handle it, which could lead to cumbersome and error-prone code.
   * **Optional** aims to eliminate the use of null values and encourage better error handling and code readability.
2. **Creating an Optional Object:**
   * You can create an **Optional** object using various factory methods, such as **Optional.of(value)**, **Optional.ofNullable(value)**, and **Optional.empty()**.
   * Example:

Optional<String> optionalValue = Optional.of("Hello, World!");

1. **Accessing the Value:**
   * You can retrieve the value from an **Optional** using the **get()** method, but it should be used with caution. If the **Optional** is empty, it throws a **NoSuchElementException**.
   * Example:

String value = optionalValue.get();

1. **Checking for Presence:**
   * You can use methods like **isPresent()** to check whether an **Optional** contains a value or **isEmpty()** to check if it's empty.
   * Example:

if (optionalValue.isPresent()) { System.out.println("Value is present: " + optionalValue.get()); } else { System.out.println("Value is absent."); }

1. **Conditional Execution with ifPresent:**
   * The **ifPresent** method allows you to execute a specific action if the **Optional** contains a value. This approach is safer than using **get()** directly.
   * Example:

optionalValue.ifPresent(val -> System.out.println("Value is present: " + val));

1. **Default Values with orElse and orElseGet:**
   * You can provide a default value to be used when the **Optional** is empty using the **orElse** or **orElseGet** methods.
   * Example:

String result = optionalValue.orElse("Default Value"); String resultFromSupplier = optionalValue.orElseGet(() -> "Default Value from Supplier");

1. **Mapping with map and flatMap:**
   * You can apply functions to the value inside an **Optional** using **map** and **flatMap**. These methods return a new **Optional** with the transformed value.
   * Example:

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

1. **Filtering with filter:**
   * The **filter** method allows you to conditionally keep the value in an **Optional** if it satisfies a certain condition.
   * Example:

Optional<String> filteredValue = optionalValue.filter(val -> val.startsWith("H"));

1. **Chaining Operations:**
   * You can chain **Optional** operations together to build expressive and safe code.
   * Example:

String result = optionalValue.map(String::toUpperCase) .orElse("No Value") .substring(0, 5);

1. **Handling Exception with orElseThrow:**
   * If you prefer to throw a specific exception when the **Optional** is empty, you can use the **orElseThrow** method.
   * Example:

String value = optionalValue.orElseThrow(() -> new NoSuchElementException("Value not found"));

The **Optional** class is a valuable tool for improving code safety and readability when dealing with potentially missing or nullable values. It encourages developers to handle null values more explicitly, reducing the risk of null pointer exceptions and making code more robust. However, it's essential to use **Optional** judiciously and not overcomplicate simple scenarios with unnecessary use of it.

**KT0305 Behind the Screens with Functional Interfaces**

* **Implement Predicate Interfaces**
* **Implement Consumer Interfaces**
* **Implement Function Interfaces**

Functional interfaces are a fundamental concept in Java that are closely related to functional programming. They represent single abstract methods (SAMs) and are used to define the types of lambda expressions and method references. In this context, we'll explore three common functional interfaces: **Predicate**, **Consumer**, and **Function**, and we'll implement them.

1. **Implementing Predicate Interfaces:**
   * The **Predicate** functional interface represents a function that takes an input and returns a boolean result. It is often used for filtering and testing conditions.
   * To implement a **Predicate**, you need to provide an implementation for the **test** method, which takes an argument and returns a boolean.
   * Example:

import java.util.function.Predicate; public class PredicateExample { public static void main(String[] args) { Predicate<Integer> isEven = num -> num % 2 == 0; System.out.println(isEven.test(4)); // Output: true System.out.println(isEven.test(7)); // Output: false } }

1. **Implementing Consumer Interfaces:**
   * The **Consumer** functional interface represents a function that takes an input and performs some action but doesn't return any result.
   * To implement a **Consumer**, you need to provide an implementation for the **accept** method, which takes an argument and performs an action.
   * Example:

import java.util.function.Consumer; public class ConsumerExample { public static void main(String[] args) { Consumer<String> printUpperCase = str -> System.out.println(str.toUpperCase()); printUpperCase.accept("hello"); // Output: "HELLO" } }

1. **Implementing Function Interfaces:**
   * The **Function** functional interface represents a function that takes an input and returns an output.
   * To implement a **Function**, you need to provide an implementation for the **apply** method, which takes an argument and returns a result.
   * Example:

import java.util.function.Function; public class FunctionExample { public static void main(String[] args) { Function<Integer, String> intToString = num -> String.valueOf(num); String result = intToString.apply(42); System.out.println(result); // Output: "42" } }

These functional interfaces are part of the **java.util.function** package and are commonly used in functional programming and when working with streams in Java. They allow you to encapsulate behavior as lambda expressions and method references, making your code more concise and expressive. By implementing these interfaces, you can pass behavior as data, leading to more flexible and reusable code.

**KT0306 Simplify Functional Programming code with Method References**

Method references in Java are a concise way to refer to methods or constructors and are often used in functional programming, particularly when working with streams, lambdas, and functional interfaces. They simplify code by providing a way to pass behavior as a reference to a method rather than defining a lambda expression. There are four types of method references in Java:

1. **Reference to a Static Method:** It refers to a static method of a class.

// Lambda expression Function<String, Integer> parseToInt = str -> Integer.parseInt(str); // Method reference Function<String, Integer> parseToIntReference = Integer::parseInt;

1. **Reference to an Instance Method of a Particular Object:** It refers to an instance method of a particular object.

// Lambda expression Predicate<String> startsWithA = str -> str.startsWith("A"); // Method reference String prefix = "A"; Predicate<String> startsWithAReference = prefix::startsWith;

1. **Reference to an Instance Method of an Arbitrary Object of a Particular Type:** It refers to an instance method of an arbitrary object of a particular type (typically used with streams).

// Lambda expression List<String> names = Arrays.asList("Alice", "Bob", "Charlie"); names.forEach(name -> System.out.println(name.toUpperCase())); // Method reference names.forEach(System.out::println);

1. **Reference to a Constructor:** It refers to a constructor.

// Lambda expression Supplier<List<String>> listSupplier = () -> new ArrayList<>(); // Method reference Supplier<List<String>> listSupplierReference = ArrayList::new;

Method references make the code more readable and concise, especially when you're using predefined methods or constructors. They can often be a more elegant alternative to lambda expressions, especially when the lambda expression simply calls an existing method. When choosing between lambda expressions and method references, consider which one improves code clarity and maintainability in your specific use case.

**KT0307Functions are First Class Citizens**

In programming languages, the concept of "functions as first-class citizens" refers to the ability to treat functions (or methods) like any other data type, such as integers, strings, or objects. In languages that support first-class functions, functions can be assigned to variables, passed as arguments to other functions, returned as values from functions, and stored in data structures like arrays or lists. This concept is a fundamental aspect of functional programming and is also present in many modern programming languages, including Java.

Here's how functions are treated as first-class citizens in Java and the benefits this concept brings:

1. **Functions as Values:**
   * In Java, functions are represented by objects that implement functional interfaces. These interfaces have a single abstract method (SAM), allowing them to be used as function types.
   * For example, **java.util.function.Function** is a functional interface that represents a function that takes one argument and produces a result. You can assign instances of this interface to variables.

Function<Integer, String> intToString = num -> String.valueOf(num);

1. **Passing Functions as Arguments:**
   * Functions can be passed as arguments to other functions or methods. This enables you to implement higher-order functions, which are functions that operate on other functions.

void process(Function<Integer, String> func, int value) { String result = func.apply(value); // Do something with the result }

1. **Returning Functions from Functions:**
   * Functions can be returned as values from other functions. This is useful for creating functions dynamically based on certain conditions or configurations.

Function<Integer, String> getFormatter(boolean useUpperCase) { if (useUpperCase) { return num -> String.valueOf(num).toUpperCase(); } else { return num -> String.valueOf(num); } }

1. **Storing Functions in Data Structures:**
   * You can store functions in data structures like arrays, lists, or maps, just like you would with other data types.

List<Function<Integer, String>> formatters = new ArrayList<>(); formatters.add(num -> String.valueOf(num)); formatters.add(num -> String.valueOf(num).toUpperCase());

1. **Lambda Expressions and Method References:**
   * Lambda expressions and method references provide a concise way to define and pass functions in Java. They are commonly used in functional programming and make the code more readable.

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5); numbers.forEach(System.out::println); // Method reference

The ability to treat functions as first-class citizens in Java allows for more flexible and expressive code. It promotes functional programming principles, such as higher-order functions, immutability, and code that's easier to reason about. This concept is a core element of modern programming languages and plays a significant role in enabling functional programming paradigms in Java.

**Internal Assessment Criteria and Weight**

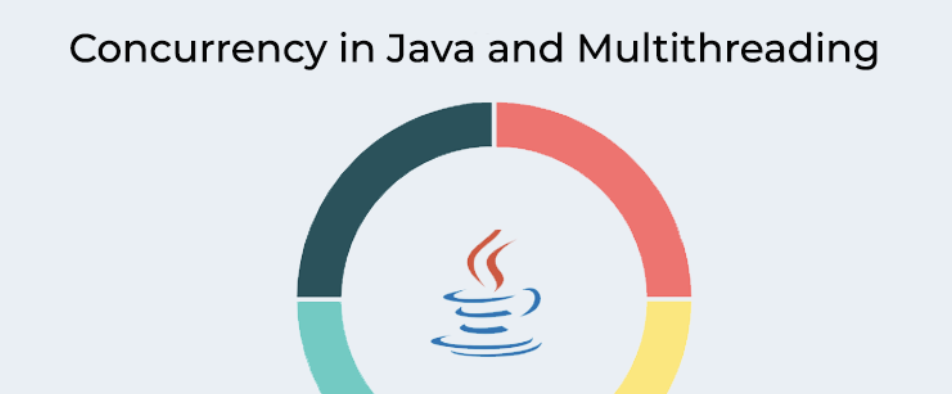
1. IAC0301 Definitions, functions, features, uses and benefits of functional programming in Java are stated.

**Session 4:** **KM-04-KT04: Concurrency with concurrent collections and atomic operations 10%**

Topic elements to be covered include:

* KT0401 Synchronized
* KT0402 Problem with synchronized
* KT0403 Atomic classes
* KT0404 Need for Concurrent Map
* KT0405 Concurrent HashMap uses different locks for different regions
* KT0406 Copy On Write concurrent collections

**KT0401 Synchronized**



Concurrency in Java is the ability of a program to manage and execute multiple tasks simultaneously. It's crucial for building efficient and responsive applications, especially in multi-threaded environments. Java provides various mechanisms and classes to support concurrency, including synchronized blocks, concurrent collections, and atomic operations. In this context, we'll focus on the concept of synchronization and concurrent collections:

1. **Synchronization in Java:**
   * Synchronization is a technique used to control access to shared resources in a multi-threaded environment. It prevents multiple threads from accessing critical sections of code simultaneously, which can lead to data corruption or inconsistent state.
   * The **synchronized** keyword is used to create synchronized blocks or methods. When a thread enters a synchronized block, it acquires the lock associated with the synchronized object, allowing only one thread to execute the synchronized code at a time.
   * Example of a synchronized block:

synchronized (sharedObject) { // Critical section of code // Only one thread can execute this at a time }

1. **Concurrent Collections:**
   * Java provides a set of concurrent collection classes in the **java.util.concurrent** package, which are designed to be thread-safe and efficient in multi-threaded scenarios.
   * Common concurrent collection classes include **ConcurrentHashMap**, **ConcurrentLinkedQueue**, and **CopyOnWriteArrayList**. These classes allow multiple threads to read and write concurrently without explicit synchronization.
   * Example of using **ConcurrentHashMap**:

ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>(); map.put("key1", 1); map.put("key2", 2); int value = map.get("key1"); // Thread-safe reading

1. **Atomic Operations:**
   * Atomic operations are operations that are guaranteed to be executed without interruption by other threads. Java provides atomic classes in the **java.util.concurrent.atomic** package for performing atomic operations on variables.
   * Common atomic classes include **AtomicInteger**, **AtomicLong**, and **AtomicReference**. These classes ensure that operations like increments, updates, and comparisons are thread-safe.
   * Example of using **AtomicInteger**:

AtomicInteger atomicInt = new AtomicInteger(0); int newValue = atomicInt.incrementAndGet(); // Thread-safe increment

1. **Thread Safety and Concurrent Programming:**
   * While synchronized blocks, concurrent collections, and atomic operations provide tools for concurrent programming, it's essential to understand the principles of thread safety and synchronization to design reliable concurrent applications.
   * Thread safety involves ensuring that shared resources are accessed and modified safely by multiple threads.
   * Proper synchronization, choosing the right data structures, and using atomic operations are key to avoiding data races and ensuring correctness in concurrent programs.
2. **Java's Concurrency Utilities:**
   * In addition to synchronized blocks, concurrent collections, and atomic operations, Java provides higher-level concurrency utilities like **ExecutorService** for managing thread pools, **Future** and **CompletableFuture** for asynchronous programming, and synchronization primitives like **CountDownLatch** and **Semaphore**.

Effective concurrent programming in Java requires a good understanding of these concepts and the careful design of multi-threaded applications to avoid common pitfalls like race conditions and deadlocks. The choice of synchronization mechanisms and concurrent data structures depends on the specific requirements and complexity of the application.

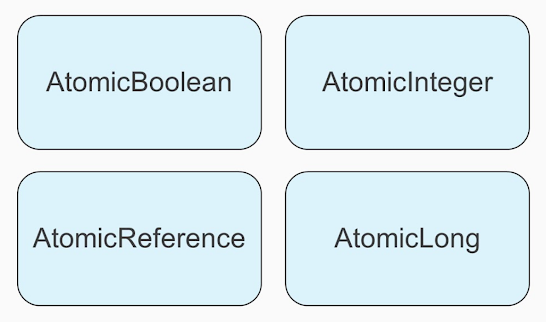
**KT0402 Problem with synchronized**

While synchronized blocks, concurrent collections, and atomic operations provide tools to handle concurrency in Java, they are not without their challenges and potential problems. It's important to understand these issues to write robust and efficient concurrent code. Here are some common problems and challenges associated with synchronized blocks, concurrent collections, and atomic operations:

1. **Deadlocks:**
   * Deadlocks occur when two or more threads wait indefinitely for each other to release resources they need. This can happen when synchronized blocks are not used consistently or when resources are acquired in a different order by different threads.
   * Avoiding deadlocks requires careful design and proper locking strategies. Techniques like using a fixed order for acquiring locks and using timeouts can help prevent deadlocks.
2. **Performance Overhead:**
   * Synchronized blocks and methods introduce performance overhead, as they can serialize access to code blocks even when no contention exists. This can lead to reduced parallelism and slower execution.
   * Concurrent collections and atomic operations are designed to mitigate some of this overhead, but they are not always suitable for all scenarios. Choosing the right synchronization mechanism is crucial for optimizing performance.
3. **Complexity and Maintainability:**
   * Synchronized blocks and methods can make code more complex and harder to maintain, especially in large and intricate multi-threaded applications. Debugging synchronization-related issues can be challenging.
   * Proper documentation and consistent coding conventions can help improve code readability and maintainability.
4. **Race Conditions:**
   * A race condition occurs when multiple threads access shared data concurrently without proper synchronization. This can result in unpredictable behavior and data corruption.
   * Concurrent collections and atomic operations are designed to minimize race conditions, but they do not eliminate them entirely. Proper synchronization must still be applied where necessary.
5. **Lock Contention:**
   * In highly contended scenarios, where many threads compete for the same locks, performance can degrade significantly due to lock contention.
   * Techniques like lock splitting (using multiple locks instead of a single global lock) and lock-free data structures can help alleviate lock contention.
6. **Complexity of Concurrent Collections:**
   * While concurrent collections provide thread-safe data structures, they may not always offer the same rich set of operations and features as their non-concurrent counterparts.
   * Developers need to carefully evaluate the trade-offs between thread-safety and functionality when choosing between concurrent and non-concurrent collections.
7. **Amdahl's Law:**
   * Amdahl's Law states that the speedup of a program from parallelization is limited by the fraction of the program that cannot be parallelized.
   * This law underscores the importance of identifying and optimizing the critical sections of code to achieve meaningful performance improvements in concurrent applications.

To address these problems and challenges, it's important to have a deep understanding of concurrency concepts, design patterns, and best practices. Additionally, tools like thread profilers and concurrency testing frameworks can help identify and diagnose concurrency-related issues in your Java applications.

**KT0403 Atomic classes**



Java provides a set of atomic classes in the **java.util.concurrent.atomic** package that are designed to perform atomic operations on variables without the need for explicit synchronization. These classes are crucial for writing safe and efficient multi-threaded code. Here are some common atomic classes and their use cases:

1. **AtomicInteger and AtomicLong:**
   * **AtomicInteger** and **AtomicLong** provide atomic operations for **int** and **long** variables, respectively.
   * Common operations include **get**, **set**, **incrementAndGet**, **decrementAndGet**, **addAndGet**, and more.
   * Example using **AtomicInteger**:

AtomicInteger atomicInt = new AtomicInteger(0); int newValue = atomicInt.incrementAndGet(); // Thread-safe increment

1. **AtomicReference:**
   * **AtomicReference** allows you to atomically update a reference to an object.
   * It provides methods like **get**, **set**, **compareAndSet**, and **getAndSet**.
   * Example using **AtomicReference**:

AtomicReference<String> atomicRef = new AtomicReference<>("Initial Value"); boolean updated = atomicRef.compareAndSet("Initial Value", "New Value"); // Thread-safe update

1. **AtomicBoolean:**
   * **AtomicBoolean** is used for atomic operations on **boolean** variables.
   * It provides methods like **get**, **set**, **getAndSet**, and **compareAndSet**.
   * Example using **AtomicBoolean**:

AtomicBoolean atomicBool = new AtomicBoolean(true); boolean updated = atomicBool.compareAndSet(true, false); // Thread-safe update

1. **AtomicIntegerArray and AtomicLongArray:**
   * These classes provide atomic operations for arrays of **int** and **long** values, respectively.
   * Operations include **get**, **set**, **getAndSet**, **getAndIncrement**, **compareAndSet**, and more.
   * Example using **AtomicIntegerArray**:

AtomicIntegerArray atomicIntArray = new AtomicIntegerArray(3); atomicIntArray.set(0, 42); // Thread-safe array element modification

1. **AtomicReferenceArray:**
   * **AtomicReferenceArray** is used for atomic operations on arrays of references.
   * It provides methods for atomic updates of array elements.
   * Example using **AtomicReferenceArray**:

AtomicReferenceArray<String> atomicRefArray = new AtomicReferenceArray<>(3); atomicRefArray.set(0, "Value"); // Thread-safe array element modification

1. **AtomicStampedReference:**
   * **AtomicStampedReference** extends **AtomicReference** and associates a stamp (integer) with each reference. This allows for atomic conditional updates based on both reference and stamp values.
   * It provides methods like **get**, **compareAndSet**, and **attemptStamp**.
   * Example using **AtomicStampedReference**:

AtomicStampedReference<String> atomicStampedRef = new AtomicStampedReference<>("Value", 0); boolean updated = atomicStampedRef.compareAndSet("Value", "New Value", 0, 1); // Thread-safe update with stamp

These atomic classes are essential for managing shared state in multi-threaded applications without the need for explicit synchronization using **synchronized** blocks or methods. They help prevent data races and ensure that operations on variables are atomic, making your code more thread-safe and efficient. When using these classes, it's important to carefully choose the appropriate one based on the data type you are working with and the specific use case.

**KT0404 Need for Concurrent Map**

A **ConcurrentMap** is a specialized interface in Java that extends the **java.util.Map** interface and is designed to support concurrent access by multiple threads. It provides a thread-safe way to manage key-value pairs, allowing multiple threads to read and write to the map concurrently without the need for explicit synchronization. The need for a **ConcurrentMap** arises in scenarios where thread safety and high concurrent performance are crucial.

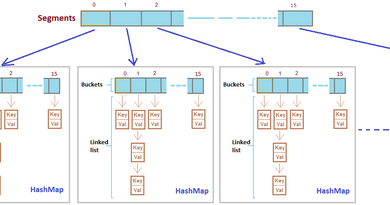
Here are some key reasons why you might need a **ConcurrentMap** in concurrent programming:

1. **Thread Safety:**
   * In multi-threaded applications, when multiple threads access and modify a regular **HashMap** or other non-concurrent map concurrently, it can lead to data corruption and unpredictable behavior.
   * A **ConcurrentMap** ensures thread safety by providing built-in mechanisms for safe concurrent access. This means that multiple threads can read and write to the map simultaneously without causing data races or inconsistencies.
2. **High Concurrency:**
   * Some applications require high levels of concurrency, where many threads are concurrently accessing and modifying data. In such cases, using a regular map with explicit synchronization (e.g., **synchronized** blocks) can lead to performance bottlenecks and reduced parallelism.
   * **ConcurrentMap** implementations are optimized for high concurrency scenarios, allowing multiple threads to operate efficiently.
3. **Atomic Operations:**
   * **ConcurrentMap** implementations often provide atomic operations for common operations like **putIfAbsent**, **replace**, and **remove**, which can be used to ensure that certain operations are performed atomically without the need for explicit synchronization.
   * Atomic operations simplify concurrent programming and help prevent data races.
4. **Scalability:**
   * Scalability is a critical factor in concurrent applications. Regular maps may not scale well when the number of threads or the rate of concurrent access increases.
   * **ConcurrentMap** implementations are designed to be highly scalable, ensuring that the performance of concurrent operations remains consistent even with a large number of threads.
5. **Performance Optimization:**
   * **ConcurrentMap** implementations are often optimized for specific use cases. For example, **ConcurrentHashMap** is optimized for read-heavy workloads and offers good performance for both reading and writing operations in a highly concurrent environment.
   * Using the appropriate **ConcurrentMap** implementation can lead to better performance and reduced contention among threads.
6. **Reduced Locking Overhead:**
   * Traditional synchronization mechanisms like **synchronized** blocks or methods can introduce significant locking overhead and contention, especially when multiple threads are accessing different parts of the map.
   * **ConcurrentMap** implementations use fine-grained locking strategies, which reduce contention and improve performance.

Common **ConcurrentMap** implementations in Java include **ConcurrentHashMap** and **ConcurrentSkipListMap**. These implementations provide different trade-offs in terms of performance and functionality, allowing you to choose the one that best fits your specific concurrency requirements.

In summary, the need for a **ConcurrentMap** arises when you want to ensure thread safety, achieve high concurrency, and optimize performance in multi-threaded applications. It provides a convenient and efficient way to manage key-value pairs in scenarios where concurrent access is a requirement.

**KT0405 Concurrent HashMap uses different locks for different regions**



The statement that "ConcurrentHashMap uses different locks for different regions" is not accurate. ConcurrentHashMap uses a segmented or partitioned approach to manage concurrency, where the data structure is divided into multiple segments or partitions, and each segment is protected by a separate lock. However, it's essential to clarify the terminology to avoid any confusion.

Here's how ConcurrentHashMap works with its segmented approach:

* **Segments:** ConcurrentHashMap is internally divided into a fixed number of segments. Each segment is essentially a mini-map, which can be thought of as a separate HashMap with its own lock.
* **Locking:** When you perform operations on a ConcurrentHashMap, such as **put**, **get**, or **remove**, the appropriate segment is determined based on the hash code of the key. Only the segment associated with that key is locked, allowing other threads to operate on different segments concurrently without contention.
* **Concurrency:** Multiple threads can read and write to different segments concurrently without blocking each other. This is a key feature of ConcurrentHashMap that improves concurrency and performance in multi-threaded scenarios.
* **Lock Granularity:** The granularity of locks in ConcurrentHashMap is finer than using a single global lock for the entire map. This means that while one thread is modifying data in one segment, other threads can simultaneously read or modify data in different segments, reducing contention and improving overall concurrency.

Here's a simplified example to illustrate the concept:

ConcurrentHashMap<String, Integer> concurrentMap = new ConcurrentHashMap<>(); // Thread 1 concurrentMap.put("A", 1); // Locks segment 1 // Thread 2 concurrentMap.put("B", 2); // Locks segment 2 // Thread 3 int valueA = concurrentMap.get("A"); // Reads from segment 1 // Thread 4 int valueB = concurrentMap.get("B"); // Reads from segment 2

In this example, Thread 1 and Thread 2 operate on different segments concurrently, while Thread 3 and Thread 4 also access different segments concurrently. This minimizes contention and allows for efficient concurrent access.

So, while ConcurrentHashMap does use separate locks for different segments, it doesn't lock different "regions" in the traditional sense. Instead, it divides the map into segments and locks each segment independently, promoting concurrency and performance.

**KT0406 Copy**

Copy-On-Write (COW) collections are a type of concurrent collection in Java that provide a high level of thread safety and are particularly useful in scenarios where reads vastly outnumber writes. These collections ensure that reads can occur concurrently with writes without the need for locks or synchronization, making them efficient for certain use cases. The key feature of COW collections is that they create a new copy of the underlying data structure whenever a modification (such as an add, set, or remove operation) is performed. Here are some important points about Copy-On-Write collections:

1. **Immutability and Thread Safety:**
   * COW collections are inherently immutable from the perspective of readers. When you read from a COW collection, you are always reading from a consistent snapshot of the data, which doesn't change during the read operation. This ensures thread safety for reading operations.
2. **Write Operations Create Copies:**
   * When you perform a write operation (e.g., adding or modifying an element), COW collections create a new copy of the entire data structure with the modification applied. The original collection remains unchanged. This approach ensures that ongoing read operations are not affected by writes.
3. **Cost of Writes:**
   * While COW collections are efficient for reading, write operations can be relatively expensive because they involve copying the entire data structure. As a result, COW collections are not suitable for scenarios where writes are frequent or where low-latency write operations are required.
4. **Use Cases:**
   * COW collections are commonly used in scenarios where reads are much more frequent than writes and where read consistency is critical. Examples include configuration data, application settings, and caches.
   * They are particularly useful in scenarios where the cost of copying the data is outweighed by the benefits of read concurrency and consistency.
5. **Examples:**
   * **CopyOnWriteArrayList**: This is a COW implementation of a list. It provides thread-safe and efficient reading of lists where writes are infrequent.
   * **CopyOnWriteArraySet**: This is a COW implementation of a set. It offers similar characteristics to **CopyOnWriteArrayList** but for sets.
   * **CopyOnWriteHashMap** (not part of the standard Java library): This is a custom implementation of a COW map, which follows similar principles.

Here's a simplified example of using **CopyOnWriteArrayList**:

import java.util.List; import java.util.concurrent.CopyOnWriteArrayList; public class CopyOnWriteArrayListExample { public static void main(String[] args) { List<String> list = new CopyOnWriteArrayList<>(); // Initial data insertion list.add("A"); list.add("B"); list.add("C"); // Concurrent read (no locking) for (String item : list) { System.out.println(item); } // Write operation (creates a new copy) list.add("D"); // Concurrent read (still no locking) for (String item : list) { System.out.println(item); } } }

In this example, you can see how **CopyOnWriteArrayList** allows for concurrent reads while still supporting write operations. Keep in mind that while COW collections are efficient for certain use cases, they are not suitable for scenarios with frequent writes, as the cost of copying the data on each write can become prohibitive.

**Internal Assessment Criteria and Weight**

1. IAC0401 Definitions, functions, features, uses and benefits of concurrency in Java are stated.

**References**

***Books:***

1. ***"Effective Java" by Joshua Bloch:*** *This book is considered a must-read for any Java developer looking to write more robust and efficient code. It covers best practices, design patterns, and common pitfalls in Java.*
2. ***"Java Concurrency in Practice" by Brian Goetz et al.:*** *For those interested in mastering Java concurrency and multithreading, this book provides an in-depth guide to writing concurrent programs correctly and efficiently.*
3. ***"Clean Code: A Handbook of Agile Software Craftsmanship" by Robert C. Martin:*** *While not Java-specific, this book focuses on writing clean and maintainable code, which is valuable for any intermediate or advanced Java programmer.*
4. ***"Java: The Complete Reference" by Herbert Schildt:*** *This comprehensive book covers a wide range of Java topics, making it a good reference for intermediate programmers looking to explore advanced features.*

***Online Courses:***

1. ***Coursera's Java Programming and Software Engineering Fundamentals (Duke University):*** *This specialization includes several courses, including "Java Programming and Software Engineering Fundamentals," "Java Programming: Arrays, Lists, and Structured Data," and more. It provides a structured path for advancing your Java skills.*
2. ***edX's Java Fundamentals for Android Development (University of Maryland):*** *If you're interested in Android app development, this course covers Java fundamentals with a focus on Android programming.*
3. ***Udemy's Java Programming Masterclass for Software Developers:*** *Taught by Tim Buchalka, this course covers Java from the basics to advanced topics and includes hands-on projects.*

***Websites and Tutorials:***

1. ***Oracle's Java Tutorials:*** *The official Java tutorials provided by Oracle are an excellent resource for learning Java in-depth. They cover a wide range of topics and include code examples.*
2. ***Baeldung (***[*https://www.baeldung.com/*](https://www.baeldung.com/)***):*** *Baeldung offers in-depth tutorials and articles on various Java-related topics, including Spring Framework, Hibernate, and more.*
3. ***GeeksforGeeks (***[*https://www.geeksforgeeks.org/java/*](https://www.geeksforgeeks.org/java/)***):*** *GeeksforGeeks provides a wealth of Java tutorials and practice problems, making it a valuable resource for learning and practicing Java.*
4. ***Java Code Geeks (***[*https://www.javacodegeeks.com/*](https://www.javacodegeeks.com/)***):*** *This website offers Java tutorials, articles, and examples, with a focus on both core Java and advanced topics.*

***Programming Challenges:***

1. ***LeetCode (***[*https://leetcode.com/*](https://leetcode.com/)***):*** *LeetCode provides coding challenges in Java and other languages. It's an excellent platform for improving your problem-solving and coding skills.*
2. ***HackerRank (***[*https://www.hackerrank.com/*](https://www.hackerrank.com/)***):*** *HackerRank offers Java challenges and competitions that allow you to practice your coding skills and algorithms.*