WEEK 5 COURSE CONTENT: INTRODUCTION TO JAVA COLLECTIONS

TOPICS:

- **Introduction to Java Collections**
- **Working with Lists, Sets, and Maps**
- **Iteration and Filtering using Streams**
- **Understanding the Collections API**
- **Comparators and Sorting Collections**
- **Using Java 8 Features in Collections**

LEARNING OUTCOME:

By the end of the module, student should be able to;

- **Understand the concept of collections**
- **Be able to create different type of collection and manipulate their contents**
- **Manipulate Lists, Sets and Maps**
- **Understand the strength and limitation of collections**
- **Use streams to handle collections**
- **Understand basic lambda and its usage in streams**

DAY 1

INTRODUCTION TO JAVA COLLECTION

Introduction to Java Collections

Collections in Java are fundamental data structures that allow you to store, manipulate, and organize data efficiently. They provide a way to manage groups of objects as a single unit, making it easier to work with large amounts of data. Collections are an essential part of the Java programming language, and understanding them is crucial for developing robust applications.

Advantages of Using Collections over Arrays:

Using collections in Java offers several advantages over arrays:

- Dynamic Size: Unlike arrays, collections can grow or shrink dynamically based on the number of elements they contain. This flexibility eliminates the need to manually manage the size of the collection, allowing for easier and more efficient data management.
- Enhanced Functionality: Collections provide a rich set of methods and operations that simplify data manipulation. You can easily add or remove elements, search for specific items, sort the collection, and perform other common tasks without having to write complex code manually.
- Type Safety: Collections provide type safety through the use of generics. Generics allow you to specify the type of elements stored in a collection, ensuring that only compatible objects can be added. This helps catch errors at compile-time and makes the code more robust.

Primary Collection Types in Java:

Java provides three primary collection types: lists, sets, and maps.

- Lists: A list is an ordered collection of elements where each element has an associated index. Lists allow duplicate elements and provide methods to add, retrieve, update, and remove elements by their index or value. Examples of list implementations in Java include ArrayList and LinkedList.
- Sets: A set is a collection that does not allow duplicate elements. It provides methods to add, remove, and check for the existence of elements. Sets do not have a specific order, and their primary use is to store a unique set of values. Examples of set implementations in Java include HashSet and TreeSet.
- Maps: A map is a collection that stores key-value pairs. Each key is associated with a value, similar to a dictionary or an associative array in other programming languages. Maps provide methods to add, retrieve, update, and remove elements based on their keys. Examples of map implementations in Java include HashMap and TreeMap.

WORKING WITH LISTS, SETS AND MAPS

To work with lists, sets, and maps in Java, you can utilize the Java Collections API, which provides a set of interfaces and classes for managing collections. Here's an overview of common operations

Creating a Collection

```
List<String> list = new ArrayList<>();
Set<Integer> set = new HashSet<>();
Map<String, Integer> map = new HashMap<>();
```

Adding Elements

```
list.add("Apple");
set.add(10);
map.put("Key", 42);
```

Accessing Elements

```
String element = list.get(0);
boolean contains = set.contains(10);
int value = map.get("Key");
```

Modifying Elements

```
list.set(0, "Banana");
set.remove(10);
map.replace("Key", 42, 43);
```

Iterating Over Elements

```
for (String element : list) {
   System.out.println(element);
}
```

```
for (int element : set) {
   System.out.println(element);
}

for (Map.Entry<String, Integer> entry : map.entrySet()) {
   System.out.println(entry.getKey() + ": " + entry.getValue());
}
```

These are just a few examples of the operations you can perform on lists, sets, and maps using the Java Collections API. Practicing coding exercises that involve these collection types will help you become more comfortable with their usage and understand their capabilities better.

DAY 2

ITERATION AND FILTERING USING STREAMS

Java streams provide a powerful way to process collections efficiently by enabling functional-style operations on data. They allow you to perform various operations such as filtering, mapping, and reducing on collections in a concise and expressive manner.

Benefits of Java Streams:

- Efficient Processing: Streams enable parallel processing of data, automatically dividing the work among multiple threads. This can significantly improve the performance of operations on large collections.
- Simplified Code: Streams promote a declarative programming style, where you specify what you want to achieve rather than how to achieve it. This leads to more concise and readable code.
- Method Chaining: Streams allow you to chain multiple operations together, creating a pipeline of transformations on the data. This eliminates the need for intermediate collections and reduces the amount of code you have to write.

Working with Streams:

To work with streams, you can follow these steps:

- Obtain a Stream: You can obtain a stream from a collection or an array using the **stream()** or **parallelStream()** methods, respectively.

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

Intermediate Operations: Intermediate operations transform the data within the stream. These operations are lazy, meaning they do not execute until a terminal operation is invoked. Some commonly used intermediate operations are:

- **filter(Predicate<T> predicate)**: Filters the elements based on a condition specified by the predicate.

- **map(Function<T, R> mapper)**: Transforms each element in the stream based on the given mapper function.
- **distinct()**: Removes duplicate elements from the stream.
- **sorted()**: Sorts the elements in natural order.
- **limit(long maxSize)**: Truncates the stream to a specified maximum size.

```
stream.filter(n -> n % 2 == 0)
          .map(n -> n * n)
          .distinct()
          .sorted()
          .limit(3)
          .forEach(System.out::println);
```

Terminal Operations: Terminal operations produce a result or a side-effect. Once a terminal operation is invoked, the stream is consumed and cannot be reused. Common terminal operations include:

- **forEach(Consumer<T> action)**: Performs an action for each element in the stream.
- **collect(Collector<T, A, R> collector)**: Aggregates the elements into a collection or a single value.
- **count()**: Returns the number of elements in the stream.
- **anyMatch(Predicate<T> predicate)**: Checks if any element satisfies the given predicate.
- **allMatch(Predicate<T> predicate)**: Checks if all elements satisfy the given predicate.
- **noneMatch(Predicate<T> predicate)**: Checks if no elements satisfy the given predicate.

```
List<Integer> squares = stream.filter(n -> n % 2 == 0)
              .map(n -> n * n)
              .collect(Collectors.toList());
              long count = stream.count();
              boolean anyMatch = stream.anyMatch(n -> n > 10);
```

Short-Circuiting Operations: Short-circuiting operations are terminal operations that may not process the entire stream. They provide early termination based on a condition. Some commonly used short-circuiting operations are:

- **findFirst()**: Returns the first element in the stream.
- **findAny()**: Returns any element in the stream.
- **min(Comparator<T> comparator)**: Returns the minimum element based on the comparator.
- **max(Comparator<T> comparator)**: Returns the maximum element based on the comparator.
- **limit(long maxSize)**: Truncates the stream to a specified maximum size.

```
Optional<Integer> firstElement = stream.filter(n -> n > 5)
                        .findFirst();
Optional<Integer> anyElement = stream.findAny();
Optional<Integer> minElement = stream.min(Comparator.naturalOrder());
```

Streams in Java provide a versatile and expressive way to perform iteration and filtering operations on data. They enable you to write code that is concise, readable, and efficient. By leveraging the power of streams and their various methods, you can simplify complex data processing tasks and make your code more maintainable.

Collections API

The Java Collections API provides a set of interfaces and classes for handling collections of objects. It offers a wide range of data structures, such as lists, sets, and maps, along with their corresponding implementations. Let's explore the main collection interfaces and some of their implementations, discussing their characteristics, performance implications, strengths, and use cases.

List Interface: The List interface represents an ordered collection that allows duplicate elements. Some commonly used implementations of List are ArrayList and LinkedList.

- **ArrayList**: ArrayList is backed by an array and provides fast random access to elements. It is efficient for accessing elements by index but slower for inserting or deleting elements in the middle, as it requires shifting subsequent elements. It's suitable for scenarios where frequent element access is required and modifications are less common.
- **Strengths**:
  - Provides fast random access to elements by index.
  - Efficient for iterating over elements.
  - Dynamic resizing of the underlying array when needed.
- **Limitations:**
  - Slower performance for inserting or deleting elements in the middle, as it requires shifting subsequent elements.
  - Resizing the array can be costly if done frequently.
  - Not suitable for scenarios where frequent modifications in the middle of the list are required.

- **LinkedList**: LinkedList is implemented as a doubly linked list and provides efficient insertions and deletions at both ends. However, random access is slower compared to ArrayList. LinkedList is useful when there's a need for frequent element insertion or deletion at the beginning or end of the list.
- **Strengths:**
  - Efficient for frequent insertion or deletion of elements at the beginning or end of the list.
  - No need for resizing since elements are stored as nodes.
  - Good for implementing queues and stacks.
- **Limitations**:
  - Slower random access to elements since sequential traversal is required.
  - Higher memory overhead due to the storage of node references.
  - Not suitable for scenarios where frequent random access to elements is required.

Set Interface: The Set interface represents an unordered collection that does not allow duplicate elements. HashSet and TreeSet are popular implementations of Set.

- **HashSet:** HashSet uses a hash table for storing elements, providing constant-time performance for basic operations such as add, remove, and contains. However, it does not maintain the insertion order. HashSet is ideal when uniqueness and fast lookup operations are important.
- **Strengths**:

    - Provides constant-time performance for adding, removing, and checking the presence of elements.
    - Ideal for ensuring uniqueness of elements in a collection.
    - Efficient for large data sets due to the underlying hash table implementation.
- **Limitations**:
    - Does not guarantee the order of elements.
    - No direct access to elements by index.
    - No support for sorting elements.

- **TreeSet**: TreeSet is implemented using a self-balancing binary search tree (usually a Red-Black tree). It maintains elements in sorted order. TreeSet provides efficient operations for add, remove, and search, but with a slightly slower performance compared to HashSet. It's suitable when elements need to be ordered or sorted.
- **Strengths**:
    - Maintains elements in sorted order based on natural ordering or a custom comparator.
    - Supports operations like finding the smallest/largest element and range queries.
    - Suitable for scenarios where elements need to be ordered or sorted.
- **Limitations**:
    - Slower performance compared to HashSet due to the underlying Red-Black tree structure.
    - Higher memory overhead for storing the tree structure.
    - Not suitable for scenarios where constant-time operations like add and remove are critical.

Map Interface: The Map interface represents a mapping of unique keys to values. Some commonly used implementations of Map are HashMap and TreeMap.

- **HashMap**: HashMap stores key-value pairs and provides constant-time performance for basic operations like put, get, and remove. It does not guarantee the order of elements. HashMap is widely used when quick access to values based on unique keys is required.
- **Strengths**:
    - Provides constant-time performance for adding, retrieving, and removing key-value pairs.
    - Efficient for large data sets due to the underlying hash table implementation.
    - Allows null keys and values.
- **Limitations**:
    - Does not guarantee the order of key-value pairs.
    - Not suitable for scenarios where elements need to be ordered based on keys.
    - Multiple threads accessing and modifying a HashMap concurrently can result in an inconsistent state.

- **TreeMap**: TreeMap is implemented as a Red-Black tree, maintaining the elements in sorted order based on their keys. It provides log(n) time complexity for most operations. TreeMap is useful when elements need to be sorted based on their keys.

- **Strengths**:
    - Maintains key-value pairs in sorted order based on natural ordering or a custom comparator.
    - Supports operations like finding the smallest/largest key and range queries.
    - Suitable for scenarios where elements need to be ordered based on keys.
- **Limitations**:
    - Slower performance compared to HashMap due to the underlying Red-Black tree structure.
    - Higher memory overhead for storing the tree structure.
    - Not suitable for scenarios where constant-time operations like add and remove are critical.

DAY 3

COMPARATORS AND SORTING OF COLLECTIONS

In the world of programming, sorting is an essential operation when working with collections of objects. While many collection classes in Java provide a natural ordering based on the elements' inherent properties, there are scenarios where we need to define custom sorting criteria. This is where comparators come into play. In this article, we will explore the concept of comparators, their role in sorting collections, how to implement custom comparators, and provide coding examples to practice sorting collections.

Understanding Comparators: A comparator in Java is an interface that allows custom sorting of objects based on specific criteria. It defines a method called **compare** that compares two objects and returns an integer value representing their relative order. A positive value indicates that the first object is greater, a negative value indicates the second object is greater, and zero indicates equality.

Implementing Custom Comparators: To implement a custom comparator, we need to create a class that implements the **Comparator** interface. The class should override the **compare** method, defining the custom sorting logic. Let's consider an example where we have a class called **Person** with properties like name, age, and salary. We want to sort a collection of **Person** objects based on their salary in descending order.

```
public class PersonComparator implements Comparator<Person> {
            @Override
            public int compare(Person person1, Person person2) {
               return person2.getSalary() - person1.getSalary();
            }
         }
```

In the above code snippet, we create a **PersonComparator** class that implements the **Comparator<Person>** interface. We override the **compare** method to compare two **Person** objects based on their salary, returning the difference between the second person's salary and the first person's salary.

Sorting Collections using Comparators: Once we have a custom comparator, we can use it to sort collections using various methods provided by the **Collections** class. The most common method is **Collections.sort()**, which sorts the collection in-place using the specified comparator.

```
List<Person> persons = new ArrayList<>();

        persons.add(new Person("Alice", 25, 5000));
        persons.add(new Person("Bob", 30, 7000));
        persons.add(new Person("Charlie", 35, 4000));

        Comparator<Person> salaryComparator = new PersonComparator();
        Collections.sort(persons, salaryComparator);

        for (Person person : persons) {
            System.out.println(person.getName() + ": " + person.getSalary());
        }
```

In the above example, we create a list of **Person** objects and populate it. We instantiate the **PersonComparator** and pass it to the **Collections.sort()** method along with the list. The list is then sorted based on the salary in descending order. Finally, we iterate over the sorted list and print the name and salary of each person.

Coding Examples and Exercises: To practice sorting collections using comparators, here are a few coding examples and exercises:

- Sort a list of strings in alphabetical order.

```
List<String> strings = new ArrayList<>();
        strings.add("Zebra");
        strings.add("Lion");
        strings.add("Elephant");

        Collections.sort(strings);
        System.out.println(strings);
```

- Sort a list of integers in ascending order.

```
List<Integer> numbers = new ArrayList<>();
        numbers.add(5);
        numbers.add(2);
        numbers.add(7);

        Collections.sort(numbers);
        System.out.println(numbers);
```

- Sort a list of custom objects based on a specific property, such as age or name.

```
List<Person> persons = new ArrayList<>();
        persons.add(new Person("Alice", 25, 5000));
        persons.add(new Person("Bob", 30, 7000));
        persons.add(new Person("Charlie", 35, 4000));

        Comparator<Person> ageComparator = Comparator.comparingInt(Person::getAge);
        Collections.sort(persons, ageComparator);
        System.out.println(persons);
```

By working through these examples and exercises, you can gain hands-on experience with sorting collections using comparators, expanding your understanding of the topic and building confidence in implementing custom sorting logic.

Conclusion: Comparators play a vital role in sorting collections based on custom criteria. They allow us to define our own ordering logic for objects, providing flexibility and control over how collections are sorted. By implementing custom comparators and using them with various sorting methods, we can tailor the sorting behavior to meet our specific requirements. Through coding examples and exercises, we have explored the practical aspect of sorting collections using comparators. Armed with this knowledge, you can now confidently handle sorting operations on collections in your Java applications.

Java 8 Features in Collections

Lambdas: Lambdas in Java are a concise way to express anonymous functions. They provide a compact syntax for defining behavior that can be passed around as data. Lambdas are particularly useful when working with functional interfaces.

Functional Interfaces: A functional interface is an interface that has exactly one abstract method. Java 8 introduced the **@FunctionalInterface** annotation to mark interfaces explicitly as functional interfaces. Functional interfaces are key to leveraging the power of lambdas.

- Runnable interface: Represents a task that can be executed concurrently.

```
@FunctionalInterface
                public interface Runnable {
                  void run();
                }
```

- Supplier: Represents a supplier of results.

```
@FunctionalInterface
                public interface Supplier<T> {
                  T get();
                }
```

- Consumer: Represents an operation that accepts a single input and returns no result.

```
@FunctionalInterface
                    public interface Consumer<T> {
                       void accept(T t);
                    }
```

- Function: Represents a function that takes an argument of type T and produces a result of type R.

```
@FunctionalInterface
                    public interface Function<T, R> {
                       R apply(T t);
                    }
```

- Predicate: Represents a predicate (boolean-valued function) of one argument.

```
@FunctionalInterface
                    public interface Predicate<T> {
                       boolean test(T t);
                    }
```

Here are examples of using Java 8 features in collection operations, including filtering, mapping, and reducing:

Filtering: Filtering allows you to select elements from a collection based on a specific condition. In this example, we'll filter a list of strings to include only strings with a length greater than 3.

```
List<String> words = Arrays.asList("Java", "Python", "C++", "JavaScript", "Ruby");

                    List<String> filteredWords = words.stream()
                       .filter(word -> word.length() > 3)
                       .collect(Collectors.toList());

                    System.out.println(filteredWords); // Output: [Java, Python, JavaScript]
```

The **filter** method is used to apply the condition **word -> word.length() > 3** as a predicate to the stream. It selects only those strings with a length greater than 3.

Mapping: Mapping allows you to transform elements in a collection into another form. In this example, we'll convert a list of names to uppercase.

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

                    List<String> upperCaseNames = names.stream()
                       .map(String::toUpperCase)
                       .collect(Collectors.toList());

                    System.out.println(upperCaseNames); // Output: [ALICE, BOB, CHARLIE]
```

These examples demonstrate how Java 8 features can be used to simplify collection operations. Filtering allows you to selectively choose elements based on conditions, mapping helps transform elements into a different representation, and reducing enables you to combine elements into a single result.

DAY 4

- Introduction to Java Collection
    - Working with Lists, Sets, Maps
- Iterating and Filtering using Streams
- Lambda statements
- Functional interfaces

Exercise

1. Convert this array = {Mary, Maryam, Mitchell, Mimi, Margaret, Emily, Monster} into a Java List.

- Filter above list to print only words with 5 characters and below
- Convert all the words to UpperCase and print out the content
- Filter and Collect only the words that start with ‘Ma’.

2. Given this array {1,2,3,4,5....20}

- Using streams, print out only the even numbers
- Sum up all the odd numbers in the filtered stream of odd numbers
- Collect the even numbers in a list named evenNumbers.
- Map the even numbers to be a square of the number itself.