collections

What is a Class?

In programming, a class is like a blueprint or a template for creating objects. It defines the characteristics (attributes) and behaviors (methods) that objects of that type will have. Think of it as a recipe for creating objects of a certain type.

Here are the key points about classes:

1. **Attributes (Fields):** A class can have attributes, also known as fields or member variables, which represent the state or data of an object. These attributes can be of various data types such as int, double, String, or even other custom types (including other classes).

For example, in a class representing a **Car**, attributes could include **make**, **model**, **year**, **color**, etc.

2. **Methods (Functions):** A class can also have methods, which represent the actions or behaviors that objects of that class can perform. Methods can manipulate the object's state (change attribute values) or perform other operations.

For instance, in the **Car** class, methods could include **start()**, **accelerate()**, **brake()**, etc.

3. **Encapsulation:** Classes in Java promote encapsulation, which means bundling the data (attributes) and the methods that operate on the data together within a single unit (the class). This helps in organizing and managing complexity by hiding the internal implementation details of an object.

4. **Constructor:** A constructor is a special type of method that is used to initialize an object of a class. It typically has the same name as the class and is invoked automatically when an object is created.

For example, a constructor for the **Car** class could initialize its attributes such as **make**, **model**, etc., when a new **Car** object is created.

5. **Inheritance:** Inheritance is a key feature of classes in object-oriented programming. It allows a class (called a subclass or derived class) to inherit attributes and methods from another class (called a superclass or base class). This promotes code reuse and allows for the creation of hierarchical relationships between classes.

For instance, you could have a **Vehicle** class as a superclass, and **Car**, **Truck**, and **Motorcycle** classes as subclasses that inherit from it.

6. **Polymorphism:** Polymorphism allows objects of different classes to be treated as objects of a common superclass type. This means that a method defined in a superclass can be overridden in a subclass to provide specific behavior. Polymorphism enables flexibility and extensibility in object-oriented design.

Overall, classes are fundamental building blocks in Java programming, allowing you to create custom types with their own attributes and behaviors, encapsulate data and methods, and organize code in a modular and reusable way.

```
// Define the Car class
public class Car {
    // Attributes (Fields)
    private String make;
    private String model;
    private int year;
    private String color;

    // Constructor
    public Car(String make, String model, int year, String color) {
        this.make = make;
        this.model = model;
        this.year = year;
        this.color = color;
    }

    // Methods
    public void start() {
        System.out.println("The " + year + " " + make + " " + model + " starts.");
```

```
    }

    public void accelerate() {
        System.out.println("The " + year + " " + make + " " + model + " accelerates.");
    }

    public void brake() {
        System.out.println("The " + year + " " + make + " " + model + " brakes.");
    }

    // Getters and setters (Encapsulation)
    public String getMake() {
        return make;
    }

    public void setMake(String make) {
        this.make = make;
    }

    public String getModel() {
        return model;
    }

    public void setModel(String model) {
        this.model = model;
    }
```

```
public int getYear() {
    return year;
}

public void setYear(int year) {
    this.year = year;
}

public String getColor() {
    return color;
}

public void setColor(String color) {
    this.color = color;
}

// Main method to demonstrate usage
public static void main(String[] args) {
    // Creating an object of the Car class
    Car myCar = new Car("Toyota", "Camry", 2022, "Red");

    // Using methods
    myCar.start();
    myCar.accelerate();
    myCar.brake();

    // Using encapsulation (getter and setter)
```

```
        System.out.println("My car's make: " + myCar.getMake());
        System.out.println("My car's model: " + myCar.getModel());
        System.out.println("My car's year: " + myCar.getYear());
        System.out.println("My car's color: " + myCar.getColor());

        // Testing inheritance and polymorphism (not implemented in this example)
    }
}
```

Explanation:

1. **Attributes:** The **Car** class has attributes **make**, **model**, **year**, and **color** to represent the characteristics of a car.
2. **Constructor:** The class has a constructor to initialize these attributes when a **Car** object is created.
3. **Methods:** It defines methods like **start()**, **accelerate()**, and **brake()** to represent the behaviors of a car.
4. **Encapsulation:** Getters and setters are provided to encapsulate access to the attributes, ensuring controlled access to the class's internal state.
5. **Main method:** A **main** method is included to demonstrate the usage of the **Car** class by creating an object and invoking its methods.

What is an Interface?

An interface in Java is like a blueprint of a class. It defines a set of methods that a class must implement if it wants to be considered as implementing that interface. Think of it as a contract that a class agrees to follow.

Here are some key points about interfaces:

1. **Method Signatures:** Interfaces contain method signatures but do not provide the implementation of these methods. This means they only declare what methods a class should have, but not how those methods should be implemented.
2. **Multiple Inheritance:** Unlike classes, which can only inherit from one superclass, a class can implement multiple interfaces. This allows for more flexibility in designing classes with different functionalities.
3. **Abstraction:** Interfaces promote abstraction by defining a common set of methods that different classes can implement in their own way. This allows for polymorphism, where objects of different classes can be treated interchangeably based on the interface they implement.
4. **Extending Interfaces:** Interfaces can also extend other interfaces, allowing for the creation of hierarchies of interfaces. This helps in organizing related functionalities and promoting code reusability.

Difference between Class and Interface:

1. **Implementation:** Classes provide the implementation of methods, while interfaces only declare the method signatures without providing any implementation.
2. **Inheritance:** A class can only extend one superclass, while it can implement multiple interfaces.
3. **Variables:** Classes can have instance variables (fields), constructors, and static methods, while interfaces cannot have instance variables or constructors, but they can have constant variables (variables with a **final** keyword) and static methods.
4. **Access Modifiers:** Methods in an interface are by default public and abstract (no need to specify explicitly), while classes can have different access modifiers for their methods and fields.

Defining and Using Interfaces:

To define an interface in Java, you use the **interface** keyword followed by the name of the interface and a list of method signatures enclosed in curly braces. Here's an example:

```
interface Animal {
    void eat();
    void sleep();
}
class Dog implements Animal {
    public void eat() {
        System.out.println("Dog is eating");
    }

    public void sleep() {
        System.out.println("Dog is sleeping");
    }
}
```

Collections:

In programming, a collection refers to a group of objects that are treated as a single unit. Imagine you have a box, and inside that box, you can put different things like toys, books, or anything else. Similarly, in programming, a collection is like that box where you can store different types of data like numbers, words, or even other collections.

Now, moving on to the **Collection Framework:**

The Collection Framework in Java is a set of classes and interfaces that provide us with different types of collections to work with. Think of it like a toolbox full of different tools for managing collections efficiently.

Here are the key points to understand about the Collection Framework:

1. **Interfaces:** The Collection Framework provides several interfaces such as List, Set, Queue, etc. These interfaces define certain behaviors and operations that collections of objects can perform. For example, the List interface allows you to store objects in a specific order and access them by their index.
2. **Classes:** Along with interfaces, the Collection Framework also provides concrete classes that implement these interfaces. For example, ArrayList and LinkedList are classes that implement the List interface, HashSet and TreeSet implement the Set interface, and so on.
3. **Common Operations:** The Collection Framework offers common operations such as adding, removing, and accessing elements from collections. These operations are standardized across different types of collections, making it easier to work with them.
4. **Generics:** Generics are an important feature of the Collection Framework. They allow you to specify the type of objects that a collection can hold. This helps in ensuring type-safety and avoids the need for explicit type casting.
5. **Iterators:** Iterators are objects that allow you to traverse through the elements of a collection sequentially. They provide methods like hasNext() to check if there are more elements, and next() to retrieve the next element in the collection.

6. **Utility Methods:** The Collection Framework includes utility classes like Collections, which provide useful methods for performing common tasks on collections. For example, sorting, searching, and shuffling elements in a collection.

Overall, the Collection Framework in Java is a powerful tool for managing and manipulating collections of objects in an efficient and flexible manner. Whether you're working with lists, sets, queues, or maps

Collections:

In the context of the Java Collections Framework, a collection refers to a group of objects, often referred to as elements, that are treated as a single unit. Collections provide a way to store, retrieve, manipulate, and process groups of objects efficiently. Examples of collections in the Java Collections Framework include ArrayList, LinkedList, HashSet, TreeSet, etc. These classes implement various interfaces from the Collections Framework.

Interfaces:

Interfaces in the Java Collections Framework define contracts that classes must adhere to if they want to provide certain collection functionalities. Interfaces do not provide any concrete implementations; instead, they specify a set of methods that implementing classes must implement. Interfaces in the Java Collections Framework include:

1. **Collection:** The root interface in the collections hierarchy. It defines the basic operations that all collections should support, such as adding, removing, and querying elements. Examples of subinterfaces of Collection include List, Set, Queue, and Deque.
2. **List:** Extends the Collection interface and represents an ordered collection of elements. It allows duplicate elements and provides methods to access elements by their index.
3. **Set:** Also extends the Collection interface but represents a collection of unique elements. It does not allow duplicate elements and provides methods for set operations such as union, intersection, and difference.
4. **Queue:** Extends the Collection interface and represents a collection designed for holding elements prior to processing. It typically follows the FIFO (first-in, first-out) order.

5. **Deque:** Extends the Queue interface and represents a double-ended queue, which supports insertion and removal of elements at both ends.
6. **Map:** Represents a mapping between keys and values. Unlike the other interfaces, Map does not extend the Collection interface because it represents a different kind of collection. Examples of implementing classes include HashMap, TreeMap, LinkedHashMap, etc.

ArrayList:

ArrayList is a part of the Collection Framework and provides dynamic arrays in Java. It allows us to store and manipulate a collection of elements dynamically. Here's a basic example along with an explanation:

```
import java.util.ArrayList;

public class ArrayListExample {
    public static void main(String[] args) {
        // Creating an ArrayList of Strings
        ArrayList<String> fruits = new ArrayList<>();

        // Adding elements to the ArrayList
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Orange");

        // Accessing elements from the ArrayList
        System.out.println("Fruits: " + fruits);

        // Removing an element from the ArrayList
        fruits.remove("Banana");
        System.out.println("Fruits after removing Banana: " + fruits);

        // Getting the size of the ArrayList
        System.out.println("Size of the ArrayList: " + fruits.size());

        // Checking if the ArrayList is empty
        System.out.println("Is the ArrayList empty? " + fruits.isEmpty());
    }
}
```

Explanation:

1. **Creating an ArrayList: We import the ArrayList class from the java.util package and create an ArrayList called fruits to store String elements.**
2. **Adding elements: We use the add() method to add elements ("Apple", "Banana", "Orange") to the ArrayList.**
3. **Accessing elements: We print the elements of the ArrayList using the toString() method, which is implicitly called when we concatenate the ArrayList with a String.**
4. **Removing an element: We use the remove() method to remove the element "Banana" from the ArrayList.**
5. **Getting the size: We use the size() method to get the number of elements in the ArrayList.**
6. **Checking if empty: We use the isEmpty() method to check if the ArrayList is empty.**

Integer.valueOf(20): This part creates an Integer object with the value 20. In Java, primitive types like int can be converted to their corresponding wrapper classes (in this case, Integer) using the valueOf() method. So, Integer.valueOf(20) creates an Integer object representing the integer value 20.

Write a program that creates an ArrayList of integers, adds some numbers to it, removes a number, and prints the size of the ArrayList.

LinkedList:

LinkedList is another implementation of the List interface provided by the Collection Framework. It represents a doubly-linked list in Java. Here's a basic example along with an explanation:

```
import java.util.LinkedList;

public class LinkedListExample {
    public static void main(String[] args) {
        // Creating a LinkedList of Strings
        LinkedList<String> colors = new LinkedList<>();

        // Adding elements to the LinkedList
        colors.add("Red");
        colors.add("Green");
        colors.add("Blue");

        // Accessing elements from the LinkedList
        System.out.println("Colors: " + colors);

        // Removing an element from the LinkedList
        colors.remove("Green");
        System.out.println("Colors after removing Green: " + colors);
```

```
        // Getting the size of the LinkedList
        System.out.println("Size of the LinkedList: " + colors.size());

        // Checking if the LinkedList is empty
        System.out.println("Is the LinkedList empty? " +
colors.isEmpty());
    }
}
```

LinkedList is similar to ArrayList as both are part of the List interface. LinkedList provides similar functionalities as ArrayList but with different underlying data structures.

Write a program that creates a LinkedList of characters, adds some characters to it, removes a character, and prints the size of the LinkedList.

Stack:

A stack is a linear data structure that follows the Last-In, First-Out (LIFO) principle, meaning that the element inserted last will be removed first. It resembles a stack of plates, where you can only add or remove the top plate.

Nature:

- **Stacks can be implemented using both classes and interfaces. In Java, Stack is a class that extends the Vector class, providing stack operations like push, pop, peek, and more.**
- **Additionally, developers can also implement a stack using an array or a linked list.**

Functionality:

- **The main functionality of a stack includes:**
  - **push(E item): Adds an element to the top of the stack.**
  - **pop(): Removes and returns the top element from the stack.**
  - **peek(): Returns the top element of the stack without removing it.**
  - **isEmpty(): Checks if the stack is empty.**
  - **search(Object o): Searches for an element in the stack and returns its position (index) from the top of the stack.**

Example:

```
import java.util.Stack;
public class StackExample {
    public static void main(String[] args) {
        // Creating a stack
        Stack<Integer> stack = new Stack<>();

        // Adding elements to the stack
        stack.push(10);
        stack.push(20);
        stack.push(30);

        // Removing and printing the top element
        System.out.println("Popped element: " + stack.pop());

        // Peeking at the top element
        System.out.println("Top element: " + stack.peek());

        // Checking if the stack is empty
        System.out.println("Is the stack empty? " + stack.isEmpty());
    }
}
```

Practice Questions:

1. **Write a program to reverse a string using a stack.**
2. **Implement a stack using an array and perform push and pop operations.**
3. **Write a program to check if a given string containing parentheses is balanced or not using a stack.**

Queue:

A queue is a linear data structure that follows the First-In, First-Out (FIFO) principle, meaning that the element inserted first will be removed first. It resembles a queue of people waiting for a service, where the person who arrived first is served first.

Nature:

- Like stacks, queues can be implemented using both classes and interfaces. In Java, Queue is an interface that defines the operations for a queue.
- Implementations of the Queue interface include LinkedList, PriorityQueue, and ArrayDeque.

Functionality:

- **The main functionality of a queue includes:**
  - **offer(E e): Adds an element to the rear of the queue.**
  - **poll(): Removes and returns the element from the front of the queue.**
  - **peek(): Returns the element from the front of the queue without removing it.**
  - **isEmpty(): Checks if the queue is empty.**
  - **size(): Returns the number of elements in the queue.**

Example:

```
import java.util.LinkedList;
import java.util.Queue;

public class QueueExample {
    public static void main(String[] args) {
        // Creating a queue
        Queue<String> queue = new LinkedList<>();

        // Adding elements to the queue
        queue.offer("Alice");
        queue.offer("Bob");
        queue.offer("Charlie");

        // Removing and printing the front element
        System.out.println("Removed element: " + queue.poll());

        // Peeking at the front element
        System.out.println("Front element: " + queue.peek());

        // Checking if the queue is empty
        System.out.println("Is the queue empty? " + queue.isEmpty());
    }
}
```