Object Oriented Programming (OOP)

Short Q's (Syllabus-wise)

Class: A class is a blueprint or a template for creating objects. It defines the properties (attributes) and behaviors (methods) that its objects will have. Classes are like molds from which objects are created.

```
class Car {
    // Attributes (fields)
    String make;
    String model;

    // Methods
    void start() {
        System.out.println("The car is starting.");
    }
}
```

Object: An object is an instance of a class, created from the class blueprint. It has its own set of attributes and can perform actions defined by the class methods.

```
public class Main {
    public static void main(String[] args) {
        // Creating objects (instances) of the Car class
        Car car1 = new Car();
        car1.make = "Toyota";
        car1.model = "Camry";

        Car car2 = new Car();
        car2.make = "Honda";
        car2.model = "Civic";

        // Calling methods on objects
        car1.start(); // Output: The car is starting.
        car2.start(); // Output: The car is starting.
    }
}
```

Abstraction: It is one of the four fundamental principles of object-oriented programming (OOP). It involves simplifying complex reality by modeling classes based on the essential properties and behaviors they possess while hiding unnecessary details. In other words, abstraction allows you to focus on what an object does, rather than how it does it.

```
abstract class Shape {
    // Common attribute
    protected double area;
```

```
    // Abstract method (no implementation)
    abstract void calculateArea();

    // Common method
    void draw() {
        System.out.println("Drawing a shape.");
    }
}

class Circle extends Shape {
    private double radius;

    Circle(double radius) {
        this.radius = radius;
    }

    // Implementation of abstract method
    void calculateArea() {
        this.area = Math.PI * radius * radius;
    }

    // Overriding the common method
    void draw() {
        System.out.println("Drawing a circle.");
        // Actual drawing code for a circle
    }
}

class Rectangle extends Shape {
    private double length;
    private double width;

    Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    // Implementation of abstract method
    void calculateArea() {
        this.area = length * width;
    }

    // Overriding the common method
    void draw() {
        System.out.println("Drawing a rectangle.");
        // Actual drawing code for a rectangle
    }
}
```

Encapsulation: It is one of the four fundamental principles of object-oriented programming (OOP). It involves bundling the data (attributes) and methods (functions) that operate on the data into a single unit called a class. Additionally, it

restricts direct access to some of the object's components and prevents the accidental modification of data. In essence, encapsulation provides data protection and a way to control access to an object's internal state.

```
public class BankAccount {
    private String accountNumber;  // Encapsulated data
    private double balance;        // Encapsulated data

    public BankAccount(String accountNumber, double initialBalance) {
        this.accountNumber = accountNumber;
        this.balance = initialBalance;
    }

    // Encapsulated methods to interact with the data
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }

    public void withdraw(double amount) {
        if (amount > 0 && balance >= amount) {
            balance -= amount;
        }
    }

    public double getBalance() {
        return balance;
    }

    public String getAccountNumber() {
        return accountNumber;
    }
}
```

For instance:

```
public class Main {
    public static void main(String[] args) {
        BankAccount account = new BankAccount("1234567890", 1000.0);

        account.deposit(500.0);
        account.withdraw(200.0);

        double balance = account.getBalance();
        String accountNumber = account.getAccountNumber();

        System.out.println("Account Number: " + accountNumber);
        System.out.println("Balance: $" + balance);
    }
}
```

Final class: A final class in Java is a class that cannot be extended or sub classed. Once a class is declared as final, it cannot be used as a superclass for any other class. This is typically done when you want to prevent any further modification or extension of a class, often for security, performance, or design reasons.

```
final class MyFinalClass {
    // Class members and methods
}

// This would result in a compilation error
class SubclassOfFinal extends MyFinalClass {
    // Subclass definition
}
```

Nested classes: In Java, you can define classes within other classes, creating a concept known as nested classes. Nested classes are divided into two main categories: **static nested classes** and **inner classes**. Let's define both with examples:

1. **Static nested classes:** A static nested class is a class that is defined within another class and is marked as static. It is associated with the outer class but does not have access to the instance variables or methods of the outer class. You can create an instance of a static nested class without creating an instance of the outer class.

```
public class OuterClass {
    static class StaticNestedClass {
        void display() {
            System.out.println("This is a static nested class.");
        }
    }
}
```

2. **Inner classes:** An inner class is a class that is defined within another class and is not marked as static. It has access to the instance variables and methods of the outer class, even the private ones. You typically need to create an instance of the outer class to access the inner class.

```
public class OuterClass {
    private int outerVariable = 10;

    class InnerClass {
        void display() {
            System.out.println("This is an inner class. Outer variable: " + outerVariable);
        }
    }
}
```

To use these nested classes:

```
public class Main {
    public static void main(String[] args) {
        OuterClass.StaticNestedClass staticNestedObj = new OuterClass.StaticNestedClass();
        staticNestedObj.display();

        OuterClass outerObj = new OuterClass();
        OuterClass.InnerClass innerObj = outerObj.new InnerClass();
```

```
        innerObj.display();
    }
}
```

Inheritance: It is one of the four fundamental principles of object-oriented programming (OOP). It allows a new class (subclass or derived class) to inherit properties and behaviors (attributes and methods) from an existing class (superclass or base class). Inheritance promotes code reuse and supports the creation of hierarchical relationships between classes.

```
class Vehicle {
    String brand;
    int year;

    Vehicle(String brand, int year) {
        this.brand = brand;
        this.year = year;
    }

    void start() {
        System.out.println("Starting the vehicle.");
    }
}

class Car extends Vehicle {
    int numberOfDoors;

    Car(String brand, int year, int numberOfDoors) {
        super(brand, year);
        this.numberOfDoors = numberOfDoors;
    }

    void accelerate() {
        System.out.println("Accelerating the car.");
    }
}

class Bicycle extends Vehicle {
    int numberOfGears;

    Bicycle(String brand, int year, int numberOfGears) {
        super(brand, year);
        this.numberOfGears = numberOfGears;
    }

    void pedal() {
        System.out.println("Pedaling the bicycle.");
    }
}
```

You can use these classes as follow:

```
public class Main {
    public static void main(String[] args) {
        Car myCar = new Car("Toyota", 2023, 4);
```

```
        Bicycle myBicycle = new Bicycle("Giant", 2023, 21);

        myCar.start();
        myCar.accelerate();
        System.out.println("Car brand: " + myCar.brand);
        System.out.println("Car year: " + myCar.year);

        myBicycle.start();
        myBicycle.pedal();
        System.out.println("Bicycle brand: " + myBicycle.brand);
        System.out.println("Bicycle year: " + myBicycle.year);
    }
}
```

Abstract classes: Abstract classes are classes in Java that cannot be instantiated directly but serve as blueprints for other classes. They can contain abstract methods (methods without implementations) and concrete methods (methods with implementations). Abstract classes are useful when you want to create a common interface for a group of related classes while enforcing that certain methods are implemented by their subclasses.

```
abstract class Shape {
    // Abstract method (no implementation)
    abstract double calculateArea();

    // Concrete method (with implementation)
    void displayArea() {
        System.out.println("Area: " + calculateArea());
    }
}

class Circle extends Shape {
    private double radius;

    Circle(double radius) {
        this.radius = radius;
    }

    // Implementing the abstract method
    double calculateArea() {
        return Math.PI * radius * radius;
    }
}

class Rectangle extends Shape {
    private double length;
    private double width;

    Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }
```

```
    // Implementing the abstract method
    double calculateArea() {
        return length * width;
    }
}
```

You can use these classes as follow:

```
public class Main {
    public static void main(String[] args) {
        Circle circle = new Circle(5.0);
        Rectangle rectangle = new Rectangle(4.0, 6.0);

        circle.displayArea();    // Output: Area: 78.53981633974483
        rectangle.displayArea(); // Output: Area: 24.0
    }
}
```

Concrete classes: Concrete classes in Java are regular classes that can be instantiated directly, and they can have both attributes (data) and methods (functions) with implementations. Unlike abstract classes, concrete classes provide complete implementations for all their methods and can be used to create objects.

```
public class Person {
    private String name;
    private int age;

    // Constructor
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Method with implementation
    public void greet() {
        System.out.println("Hello, my name is " + name + " and I am " + age + " years old.");
    }

    // Getter methods
    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}
```

For instance:

```
public class Main {
    public static void main(String[] args) {
        Person person1 = new Person("Alice", 30);
        Person person2 = new Person("Bob", 25);
```

```
        person1.greet(); // Output: Hello, my name is Alice and I am 30 years old.
        person2.greet(); // Output: Hello, my name is Bob and I am 25 years old.

        String name = person1.getName();
        int age = person2.getAge();

        System.out.println(name); // Output: Alice
        System.out.println(age);  // Output: 25
    }
}
```

Inheritance via Abstract classes: Inheritance via abstract classes is a fundamental concept in object-oriented programming (OOP). It allows you to create a hierarchy of classes where a subclass (derived class) inherits properties and behaviors (attributes and methods) from an abstract superclass (base class). Abstract classes serve as blueprints for other classes, and they may contain abstract methods (methods without implementations) that must be defined by their concrete subclasses.

```
abstract class Animal {
    String name;

    Animal(String name) {
        this.name = name;
    }

    abstract void makeSound(); // Abstract method
}

class Dog extends Animal {
    Dog(String name) {
        super(name);
    }

    void makeSound() {
        System.out.println(name + " barks.");
    }
}

class Cat extends Animal {
    Cat(String name) {
        super(name);
    }

    void makeSound() {
        System.out.println(name + " meows.");
    }
}
```

For instance:

```
public class Main {
```

```
    public static void main(String[] args) {
        Animal dog = new Dog("Buddy");
        Animal cat = new Cat("Whiskers");

        dog.makeSound(); // Output: Buddy barks.
        cat.makeSound(); // Output: Whiskers meows.
    }
}
```

Extending the hierarchy: Extending the hierarchy in the context of object-oriented programming (OOP) refers to the process of adding new classes to an existing class hierarchy. It allows you to expand the inheritance structure by creating subclasses that inherit properties and behaviors from existing classes, including other subclasses.

```
abstract class Animal {
    String name;

    Animal(String name) {
        this.name = name;
    }

    abstract void makeSound();
}

class Mammal extends Animal {
    Mammal(String name) {
        super(name);
    }

    void makeSound() {
        System.out.println(name + " makes a mammal sound.");
    }
}

class Bird extends Animal {
    Bird(String name) {
        super(name);
    }

    void makeSound() {
        System.out.println(name + " makes a bird sound.");
    }
}
```

Now, let's extend this hierarchy by adding a new class ‘‘Reptile’’:

```
class Reptile extends Animal {
    Reptile(String name) {
        super(name);
    }

    void makeSound() {
        System.out.println(name + " makes a reptile sound.");
```

```
    }
}
```

For instance:

```
public class Main {
    public static void main(String[] args) {
        Animal dog = new Mammal("Buddy");
        Animal sparrow = new Bird("Sparrow");
        Animal snake = new Reptile("Snake");

        dog.makeSound();    // Output: Buddy makes a mammal sound.
        sparrow.makeSound(); // Output: Sparrow makes a bird sound.
        snake.makeSound();   // Output: Snake makes a reptile sound.
    }
}
```

Castings in OOP (Java): Upcasting and downcasting are two casting operations used in object-oriented programming languages like Java to convert references between classes in an inheritance hierarchy.

1. **Upcasting:** Upcasting refers to the process of casting an object reference to a superclass type, moving up the inheritance hierarchy. It is considered safe because you are treating a derived class object as an instance of its base class. Upcasting is done implicitly, and you don't need to use explicit casting (e.g., "(Superclass) derivedObject") in most cases.

```
class Animal {
    void eat() {
        System.out.println("Animal is eating.");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("Dog is barking.");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog(); // Upcasting
        myDog.eat(); // Output: Animal is eating.
        // myDog.bark(); // This would result in a compilation error because bark()
is
         not accessible.
    }
}
```

2. **Downcasting:** Downcasting is the reverse operation of upcasting, where you explicitly cast a superclass reference back to a subclass type. It's potentially unsafe because not all objects referred to by a superclass reference will be of the subclass type. Therefore, you need to use explicit casting and be cautious to avoid "ClassCastException" errors.

```
class Animal {
    void eat() {
        System.out.println("Animal is eating.");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("Dog is barking.");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Dog();

        // Downcasting (explicit)
        if (myAnimal instanceof Dog) {
            Dog myDog = (Dog) myAnimal;
            myDog.bark(); // Output: Dog is barking.
        }
    }
}
```

Downcasting is often used when you have a reference to a superclass but need to access subclass-specific methods or attributes. However, it should be used carefully to avoid runtime errors.

Interfaces: Interfaces in Java are a way to define a contract or a set of abstract methods that must be implemented by any class that claims to adhere to that interface. They provide a way to achieve multiple inheritance in Java, as a class can implement multiple interfaces.

```
// Define the Shape interface
interface Shape {
    double calculateArea(); // Abstract method for calculating the area
    void display();         // Abstract method for displaying shape information
}
```

Now, let's create two classes, "Circle" and "Rectangle", that implement the Shape interface:

```
class Circle implements Shape {
    private double radius;

    Circle(double radius) {
        this.radius = radius;
    }
```

```
    public double calculateArea() {
        return Math.PI * radius * radius;
    }

    public void display() {
        System.out.println("Circle with radius " + radius);
    }
}

class Rectangle implements Shape {
    private double length;
    private double width;

    Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    public double calculateArea() {
        return length * width;
    }

    public void display() {
        System.out.println("Rectangle with length " + length + " and width " + width);
    }
}
```

Now, create instances of "Circle" and "Rectangle" and use them as Shape objects:

```
public class Main {
    public static void main(String[] args) {
        Shape circle = new Circle(5.0);
        Shape rectangle = new Rectangle(4.0, 6.0);

        System.out.println("Circle Area: " + circle.calculateArea());
  // Output: Circle Area: 78.53981633974483
        circle.display(); // Output: Circle with radius 5.0

        System.out.println("Rectangle Area: " + rectangle.calculateArea());
  // Output: Rectangle Area: 24.0
        rectangle.display(); // Output: Rectangle with length 4.0 and width 6.0
    }
}
```

Composition: Composition is a design principle in object-oriented programming (OOP) where one class (the composite or container class) contains an object of another class (the component or contained class) as one of its instance variables. It allows you to build more complex objects by combining simpler objects, promoting code reuse and maintainability.

```
class Engine {
```

```
    void start() {
        System.out.println("Engine started.");
    }
}

class Car {
    private Engine engine; // Composition: Car has an Engine

    Car() {
        this.engine = new Engine(); // Create an Engine object
    }

    void start() {
        System.out.println("Car started.");
        engine.start(); // Delegate starting to the Engine object
    }
}
```

Now, create a “Car” object and call its “start()” method:

```
public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.start();
    }
}
```

Composition is a powerful design principle in OOP that promotes code modularity, flexibility, and maintainability by creating complex objects through the combination of simpler, reusable components.

The has-a Relationship: The "has-a relationship" is a fundamental concept in object-oriented programming (OOP) that represents a form of composition. It signifies that a class or object contains or is composed of another class or object as one of its components or attributes. In other words, it indicates that one class has a relationship with another class by containing it as a part.

```
class Wheel {
    int size;

    Wheel(int size) {
        this.size = size;
    }

    void rotate() {
        System.out.println("Wheel is rotating.");
    }
}

class Car {
    private Wheel[] wheels; // Composition: Car has wheels

    Car(int numberOfWheels) {
```

```
        this.wheels = new Wheel[numberOfWheels]; // Create an array of wheels
        for (int i = 0; i < numberOfWheels; i++) {
            wheels[i] = new Wheel(18); // Create wheel objects
        }
    }

    void drive() {
        System.out.println("Car is moving.");
        for (Wheel wheel : wheels) {
            wheel.rotate(); // Rotate each wheel
        }
    }
}
```

Now, you can create a "Car" object and call its "drive()" method:

```
public class Main {
    public static void main(String[] args) {
        Car myCar = new Car(4); // A car with 4 wheels
        myCar.drive();
    }
}
```

The "has-a relationship" is a key concept in OOP and is used to model the relationship between classes where one class contains or is composed of another class. It promotes code modularity, reusability, and maintainability by creating objects through composition.

Polymorphism: Polymorphism is a fundamental concept in object-oriented programming (OOP) that allows objects of different classes to be treated as objects of a common superclass. It enables you to write code that can work with objects of multiple classes in a unified way. Polymorphism simplifies code and promotes flexibility and extensibility in software design.

```
class Shape {
    void draw() {
        System.out.println("Drawing a shape.");
    }
}

class Circle extends Shape {
    void draw() {
        System.out.println("Drawing a circle.");
    }
}

class Rectangle extends Shape {
    void draw() {
        System.out.println("Drawing a rectangle.");
    }
}
```

Now, create an array of shapes and call the "draw()" method on each shape without knowing their specific types:

```
public class Main {
    public static void main(String[] args) {
        Shape[] shapes = new Shape[3];
        shapes[0] = new Circle();
        shapes[1] = new Rectangle();
        shapes[2] = new Shape(); // A generic shape

        for (Shape shape : shapes) {
            shape.draw(); // Polymorphic method call
        }
    }
}
```

Dynamic (or Late) Binding: Dynamic binding, also known as late binding or runtime polymorphism, is a concept in object-oriented programming where the specific method implementation to be called is determined at runtime based on the actual type of the object, rather than at compile time based on the reference type. It allows you to write more flexible and extensible code.

```
class Shape {
    void draw() {
        System.out.println("Drawing a shape.");
    }
}

class Circle extends Shape {
    void draw() {
        System.out.println("Drawing a circle.");
    }
}

class Rectangle extends Shape {
    void draw() {
        System.out.println("Drawing a rectangle.");
    }
}
```

Now, let's use dynamic binding to call the “draw()” method on objects of different shapes:

```
public class Main {
    public static void main(String[] args) {
        Shape shape1 = new Circle();
        Shape shape2 = new Rectangle();

        shape1.draw(); // Dynamic binding: Draws a circle.
        shape2.draw(); // Dynamic binding: Draws a rectangle.
    }
}
```

The Relationship b/w interfaces & polymorphism: The relationship between interfaces and polymorphism is that interfaces enable and facilitate polymorphism in object-oriented programming (OOP). Polymorphism allows objects of different classes to be treated as objects of a common superclass or interface, making code more flexible and extensible.

Let's say we have an interface called "Drawable":

```
interface Drawable {
    void draw();
}
```

And we have two classes, "Circle" and "Rectangle", both of which implement the Drawable interface:

```
class Circle implements Drawable {
    void draw() {
        System.out.println("Drawing a circle.");
    }
}

class Rectangle implements Drawable {
    void draw() {
        System.out.println("Drawing a rectangle.");
    }
}
```

Now, create an array of "Drawable" objects and call the "draw()" method on each object without knowing their specific types:

```
public class Main {
    public static void main(String[] args) {
        Drawable[] drawables = new Drawable[2];
        drawables[0] = new Circle();
        drawables[1] = new Rectangle();

        for (Drawable drawable : drawables) {
            drawable.draw(); // Polymorphic method call
        }
    }
}
```

Interfaces play a crucial role in achieving polymorphism in Java by providing a common contract that multiple classes can adhere to. This promotes code reuse, flexibility, and the ability to work with different objects in a unified way.

Wrapper classes: Wrapper classes in Java are a set of classes that encapsulate primitive data types (like int, char, boolean, etc.) and provide utility methods for working with these values as objects. They are part of the Java API and are primarily used when you need to treat primitive types as objects, such as when working with collections or using certain Java libraries that require objects.

Consider the primitive data type "int". In Java, there is a corresponding wrapper class called "Integer":

```
int primitiveInt = 42; // Primitive int
```

```
Integer wrappedInt = new Integer(42); // Wrapped in Integer object
```

Wrapper classes provide several useful methods for converting, manipulating, and comparing primitive values as objects.

For instance, you can convert between primitives and their wrapper objects like this:

```
int unwrapped = wrappedInt.intValue(); // Convert Integer to int
Integer rewrapped = Integer.valueOf(unwrapped); // Convert int to Integer
```

Wrapper classes are commonly used when working with data structures like ArrayList, HashMap, or when you need to perform operations that require objects rather than primitive types. For example:

```
ArrayList<Integer> numbers = new ArrayList<Integer>();
numbers.add(5); // Autoboxing: converts int to Integer
numbers.add(10);

int sum = numbers.get(0) + numbers.get(1); // Unboxing: converts Integer to int
System.out.println("Sum: " + sum); // Output: Sum: 15
```

Boxing & Un-boxing: Boxing and unboxing are operations in Java that involve converting between primitive data types and their corresponding wrapper classes. These operations allow you to treat primitive types as objects and vice versa.

1. **Boxing (Autoboxing):** Boxing is the process of converting a primitive data type into its corresponding wrapper class object. Autoboxing is the automatic conversion performed by the Java compiler when needed.

   ```
   int primitiveInt = 42; // Primitive int
   Integer wrappedInt = primitiveInt; // Autoboxing: Converts int to Integer
   ```

2. **Un-boxing:** Unboxing is the process of extracting the primitive value from a wrapper class object. Unboxing can be done explicitly using methods like “intValue()”, or it can be done automatically (auto-unboxing) when needed.

   ```
   Integer wrappedInt = new Integer(42); // Integer object
   int unwrapped = wrappedInt.intValue(); // Unboxing: Converts Integer to int
   ```

Here's an example that demonstrates both boxing and unboxing in the context of a list of integers:

```
ArrayList<Integer> numbers = new ArrayList<Integer>();
numbers.add(5); // Autoboxing: Converts int to Integer
numbers.add(10);

int sum = numbers.get(0) + numbers.get(1); // Auto-unboxing: Converts Integer to int
System.out.println("Sum: " + sum); // Output: Sum: 15
```

Packages: Packages in Java are a way to organize and group related classes and interfaces. They provide a mechanism for managing and controlling access to classes, preventing naming conflicts, and creating a hierarchical structure for organizing your Java code.

1. **Defining Packages:**

   ```
   package zoo.animals;
   ```

```
public class Lion {
    // Lion class implementation
}
```

2. **Accessing Classes from Packages:**

```
import zoo.animals.Lion;

public class ZooManager {
    public static void main(String[] args) {
        Lion lion = new Lion();
        // Use the Lion class
    }
}
```

Using packages helps in:

- **Organization:** You can group related classes together in a structured manner.
- **Encapsulation:** Classes within a package can control access to their members (fields and methods) using access modifiers like private, protected, and public.
- **Avoiding Naming Conflicts:** If two classes have the same name but are in different packages, there won't be naming conflicts.
- **Reusability:** You can create libraries or reusable components by packaging related classes together.

Exceptions & Exception handling: Exceptions in Java are events that occur during the execution of a program, disrupting the normal flow of instructions. These events can be caused by various factors, such as invalid input, division by zero, or running out of memory. Exception handling is a mechanism that allows you to gracefully manage and recover from such unexpected events.

```
public class Main {
    public static void main(String[] args) {
        int numerator = 10;
        int denominator = 0; // Division by zero will throw an exception
        int result = numerator / denominator;
        System.out.println("Result: " + result);
    }
}
```

In this code, we attempt to divide “numerator” by “denominator”. However, when “denominator is 0”, it results in an “ArithmeticException”, which is an example of an exception in Java.

To handle this exception and prevent the program from crashing, you can use a “try-catch block”:

```
public class Main {
    public static void main(String[] args) {
        int numerator = 10;
        int denominator = 0;

        try {
            int result = numerator / denominator; // This may throw an exception
```

```
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.err.println("Error: Division by zero"); // Handle the exception
        }
    }
}
```

Exception handling allows you to gracefully handle unexpected situations in your code and take appropriate actions to recover or provide feedback to the user. It's an essential aspect of writing robust and reliable Java applications.

File systems & Paths: File systems and paths refer to the organization and structure of files and directories (folders) on a computer's storage medium, such as a hard drive or SSD. Paths are used to locate and navigate through the file system to access files and directories.

1. **File System:** A file system is a method used by an operating system to manage and store files and directories on a storage device. Common file systems in use include NTFS (used by Windows), ext4 (used by many Linux distributions), and HFS+ (used by macOS).
2. **Paths:** A path is a string representation of a file or directory's location within a file system. Paths can be either absolute or relative.

- **Absolute Path Example:**
  `C:\Users\John\Documents\example.txt`
- **Relative Path Example:**
  `../images/pic.jpg`

File & Directory handling & manipulation: File and directory handling and manipulation in programming refer to operations and techniques used to create, read, write, move, delete, and otherwise manage files and directories on a computer's file system. This is essential for working with data and organizing information within a program.

- **Creating a Directory:**

```
import java.io.File;

public class FileHandlingExample {
    public static void main(String[] args) {
        File directory = new File("my_directory");

        if (!directory.exists()) {
            boolean created = directory.mkdir();
            if (created) {
                System.out.println("Directory created successfully.");
            } else {
                System.err.println("Failed to create directory.");
            }
        }
    }
}
```

- **Creating & Writing to a File:**

```
import java.io.FileWriter;
import java.io.IOException;

public class FileHandlingExample {
    public static void main(String[] args) {
        try {
            FileWriter writer = new FileWriter("my_file.txt");
            writer.write("Hello, world!");
            writer.close();
            System.out.println("File created and written successfully.");
        } catch (IOException e) {
            System.err.println("Error writing to the file: " + e.getMessage());
        }
    }
}
```

- **Listing Files in a Directory:**

```
import java.io.File;

public class FileHandlingExample {
    public static void main(String[] args) {
        File directory = new File("my_directory");

        if (directory.exists() && directory.isDirectory()) {
            File[] files = directory.listFiles();
            if (files != null) {
                System.out.println("Files in the directory:");
                for (File file : files) {
                    System.out.println(file.getName());
                }
            }
        }
    }
}
```

Input/Output (I/O) Streams: Input/Output (I/O) Streams in programming are a mechanism for reading data from or writing data to various sources, such as files, network connections, or memory, in a sequential manner. Streams are a fundamental concept in handling data in many programming languages, including Java.

- **Reading from a File:**

```
import java.io.FileInputStream;
import java.io.IOException;

public class InputStreamExample {
    public static void main(String[] args) {
        try {
            FileInputStream inputStream = new FileInputStream("input.txt");
            int data;
```

```
            while ((data = inputStream.read()) != -1) {
                System.out.print((char) data); // Display the character
            }
            inputStream.close();
        } catch (IOException e) {
            System.err.println("Error reading from the file: " + e.getMessage());
        }
    }
}
```

- **Writing to a File:**

```
import java.io.FileOutputStream;
import java.io.IOException;

public class OutputStreamExample {
    public static void main(String[] args) {
        try {
            FileOutputStream outputStream = new FileOutputStream("output.txt");
            String text = "Hello, world!";
            byte[] bytes = text.getBytes(); // Convert the string to bytes
            outputStream.write(bytes);
            outputStream.close();
            System.out.println("Data written to the file.");
        } catch (IOException e) {
            System.err.println("Error writing to the file: " + e.getMessage());
        }
    }
}
```

Reading binary data: Reading binary data refers to the process of reading and interpreting data that is encoded in a binary format, which represents information using only two possible values (usually 0 and 1). Binary data can represent a wide range of information, including images, audio, video, executable files, and more. Reading binary data often involves reading bytes or bits from a data source and interpreting them according to a specific format or protocol.

```
import java.io.FileInputStream;
import java.io.IOException;

public class ReadBinaryDataExample {
    public static void main(String[] args) {
        try {
            FileInputStream fileInputStream = new FileInputStream("binary_data.dat");
            byte[] buffer = new byte[4]; // A buffer to read 4 bytes at a time (for an
            integer)

            while (fileInputStream.read(buffer) != -1) {
                int intValue = byteArrayToInt(buffer);
                System.out.println("Read integer value: " + intValue);
            }

            fileInputStream.close();
```

```
        } catch (IOException e) {
            System.err.println("Error reading binary data: " + e.getMessage());
        }
    }

    // Convert a byte array to an integer (assuming little-endian byte order)
    private static int byteArrayToInt(byte[] bytes) {
        return (bytes[0] & 0xFF) |
               ((bytes[1] & 0xFF) << 8) |
               ((bytes[2] & 0xFF) << 16) |
               ((bytes[3] & 0xFF) << 24);
    }
}
```

Reading binary data is a common task when working with various file formats, network protocols, or low-level data processing. It requires careful handling of bytes and often involves interpreting data based on specific conventions or standards.

Writing binary data: Writing binary data refers to the process of creating or encoding data in a binary format, which represents information using only two possible values (usually 0 and 1). Binary data can represent a wide range of information, including images, audio, video, executable files, and more. When writing binary data, you typically convert the data into a binary format that can be easily understood and interpreted by programs that read the data.

```
import java.io.FileOutputStream;
import java.io.IOException;

public class WriteBinaryDataExample {
    public static void main(String[] args) {
        int[] valuesToWrite = {42, 123, 987, 654};

        try {
            FileOutputStream fileOutputStream = new FileOutputStream("binary_data.dat");

            for (int value : valuesToWrite) {
                byte[] bytes = intToByteArray(value);
                fileOutputStream.write(bytes);
            }

            fileOutputStream.close();
            System.out.println("Binary data written to the file.");
        } catch (IOException e) {
            System.err.println("Error writing binary data: " + e.getMessage());
        }
    }

    // Convert an integer to a byte array (assuming little-endian byte order)
    private static byte[] intToByteArray(int value) {
        byte[] bytes = new byte[4];
        bytes[0] = (byte) (value & 0xFF);
        bytes[1] = (byte) ((value >> 8) & 0xFF);
        bytes[2] = (byte) ((value >> 16) & 0xFF);
```

```
        bytes[3] = (byte) ((value >> 24) & 0xFF);
        return bytes;
    }
}
```

Writing binary data is commonly done when creating binary files, saving binary representations of complex data structures, or sending data over network protocols that require binary encoding. It's important to ensure that the binary format used for writing and the format expected for reading are compatible and adhere to any specified conventions or standards.

Writing text (characters): Writing text (characters) involves creating or encoding textual data in a human-readable format, typically using characters from a character set like ASCII or Unicode. When writing text, you deal with strings of characters that can represent various forms of data, such as text documents, configuration files, or user inputs.

```
import java.io.FileWriter;
import java.io.IOException;

public class WriteTextExample {
    public static void main(String[] args) {
        String textToWrite = "Hello, world!\nThis is a text file.";
        String filePath = "text_file.txt";

        try {
            FileWriter fileWriter = new FileWriter(filePath);
            fileWriter.write(textToWrite);
            fileWriter.close();
            System.out.println("Text file created successfully.");
        } catch (IOException e) {
            System.err.println("Error writing text: " + e.getMessage());
        }
    }
}
```

Writing text is fundamental for tasks like creating and editing text files, logging information, saving user preferences, and generating reports. Text files are widely used for storing human-readable data.

Reading text (characters): Reading text (characters) involves the process of retrieving and interpreting textual data from a file or other data source. When reading text, you typically read sequences of characters, which can represent a wide range of information, such as text documents, configuration files, or user inputs.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class ReadTextExample {
    public static void main(String[] args) {
        String filePath = "text_file.txt";

        try {
            FileReader fileReader = new FileReader(filePath);
```

```
            BufferedReader bufferedReader = new BufferedReader(fileReader);

            String line;
            while ((line = bufferedReader.readLine()) != null) {
                System.out.println(line); // Display each line of text
            }

            bufferedReader.close();
        } catch (IOException e) {
            System.err.println("Error reading text: " + e.getMessage());
        }
    }
}
```

Reading text is essential for tasks like loading configuration settings, reading data from user inputs, parsing log files, and processing textual information from various sources. It allows your program to work with and understand human-readable data.

Logging with a print stream: Logging with a print stream refers to the practice of recording information, typically messages or events, generated during the execution of a program. A print stream, such as “System.out” in Java, can be used to direct these log messages to a designated output, such as the console, a file, or a network stream, allowing developers to monitor and debug the application.

```
public class LoggingExample {
    public static void main(String[] args) {
        // Log informational messages
        System.out.println("Application started.");
        System.out.println("Processing data...");

        // Simulate an error
        int result = divide(10, 0);
        if (result == Integer.MIN_VALUE) {
            System.err.println("Error: Division by zero occurred.");
        } else {
            System.out.println("Result: " + result);
        }

        // Log a closing message
        System.out.println("Application finished.");
    }

    // A method that performs division and handles exceptions
    public static int divide(int numerator, int denominator) {
        try {
            return numerator / denominator;
        } catch (ArithmeticException e) {
            // Log the exception
            System.err.println("Exception: " + e.getMessage());
            return Integer.MIN_VALUE;
        }
    }
```

```
}
```

When you run this Java program, the log messages will be displayed in the console:

```
Application started.
Processing data...
Exception: / by zero
Application finished.
```

Random access files: Random access files refer to a type of file handling in which you can read from or write to any specific location within a file, rather than simply reading or writing sequentially from the beginning to the end of the file. This random access allows you to efficiently retrieve or modify data at a particular file position, making it useful for tasks like database storage, indexed data, or binary data storage.

```
import java.io.*;

public class RandomAccessFileExample {
    public static void main(String[] args) {
        // Define a student record structure (id: 4 bytes, name: 20 bytes, score: 4
bytes)
        final int RECORD_SIZE = 28;

        try {
            RandomAccessFile file = new RandomAccessFile("student_data.dat", "rw");

            // Writing student records
            file.seek(0); // Move the file pointer to the beginning
            file.writeInt(1); // Student ID
            file.writeUTF("Alice"); // Student name
            file.writeInt(95); // Student score

            file.seek(RECORD_SIZE); // Move to the next record position
            file.writeInt(2);
            file.writeUTF("Bob");
            file.writeInt(88);

            // Reading student records
            file.seek(0); // Move the file pointer to the beginning
            while (file.getFilePointer() < file.length()) {
                int id = file.readInt();
                String name = file.readUTF();
                int score = file.readInt();
                     System.out.println("Student ID: " + id + ", Name: " + name + ",
                      Score: " + score);
            }

            file.close();
        } catch (IOException e) {
            System.err.println("Error: " + e.getMessage());
        }
    }
}
```

Random access files are useful when you need to manage structured data in a file, and you want to quickly access and update specific records without reading the entire file sequentially. This approach is commonly used in database systems and file formats that require efficient data retrieval and modification.

Object serialization: Object serialization is the process of converting an object's state (its fields and their values) into a binary or textual format that can be easily stored in a file, sent over a network, or otherwise persisted. This allows you to save the current state of an object and later recreate it by deserializing the stored data. Serialization is particularly useful in Java for tasks like saving and restoring object states, caching, and remote communication.

```
import java.io.*;

class Person implements Serializable {
    private static final long serialVersionUID = 1L;
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}

public class SerializationExample {
    public static void main(String[] args) {
        // Create a Person object
        Person person = new Person("Alice", 30);

        // Serialize the object to a file
        try (ObjectOutputStream outputStream = new ObjectOutputStream(new
        FileOutputStream("person.ser"))) {
            outputStream.writeObject(person);
            System.out.println("Object serialized and saved.");
        } catch (IOException e) {
            System.err.println("Error during serialization: " + e.getMessage());
        }

        // Deserialize the object from the file
        try (ObjectInputStream inputStream = new ObjectInputStream(new
        FileInputStream("person.ser"))) {
            Person deserializedPerson = (Person) inputStream.readObject();
            System.out.println("Object deserialized: Name - " +
            deserializedPerson.getName() + ", Age - " + deserializedPerson.getAge());
        } catch (IOException | ClassNotFoundException e) {
```

```
            System.err.println("Error during deserialization: " + e.getMessage());
        }
    }
}
```

Serialization is particularly useful when you need to save the state of complex objects, transfer objects over a network, or implement persistence mechanisms in your Java applications. It allows you to preserve and restore the state of objects efficiently.

Collections: Collections in programming refer to data structures or classes that are used to group and manage multiple objects or elements as a single unit. These collections provide various operations and methods for adding, removing, and manipulating elements, making it easier to work with groups of data. Collections are essential in most programming languages for tasks like storing, organizing, and processing data efficiently.

```
import java.util.ArrayList;
import java.util.List;

public class CollectionsExample {
    public static void main(String[] args) {
        // Create an ArrayList to store a list of names
        List<String> names = new ArrayList<>();

        // Add elements to the ArrayList
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");

        // Access elements by index
        String firstPerson = names.get(0);
        System.out.println("First person: " + firstPerson);

        // Modify elements
        names.set(1, "Eve");
        System.out.println("Modified list: " + names);

        // Remove elements
        names.remove(2);
        System.out.println("Updated list: " + names);

        // Check if an element exists in the collection
        boolean containsAlice = names.contains("Alice");
        System.out.println("Contains Alice? " + containsAlice);
    }
}
```

Collections like “ArrayList” provide a convenient way to work with groups of data, and Java offers various other collection types such as sets, maps, and queues, each tailored for specific use cases. Other programming languages also offer similar collection classes or data structures to facilitate data manipulation and organization.

For-each loop: A for-each loop, also known as an enhanced for loop or a foreach loop, is a programming construct that simplifies the process of iterating over elements in an iterable collection, such as an array, list, or other data structures. It allows you to loop through all the elements of the collection without the need for explicit index management or loop control variables.

```
public class ForEachLoopExample {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3, 4, 5};

        // Traditional for loop
        System.out.println("Using a traditional for loop:");
        for (int i = 0; i < numbers.length; i++) {
            System.out.println(numbers[i]);
        }

        // For-each loop (enhanced for loop)
        System.out.println("\nUsing a for-each loop:");
        for (int number : numbers) {
            System.out.println(number);
        }
    }
}
```

The output for both loops will be the same, displaying the numbers 1 through 5. For-each loops are particularly useful when you want to iterate over all elements in a collection without the complexity of managing indices. They are available in many programming languages and are a concise and readable way to work with collections.

GUI concept: GUI concepts, or Graphical User Interface concepts, refer to the principles, elements, and techniques used to create user interfaces in software applications. GUIs provide a visual way for users to interact with software, making it more user-friendly and intuitive. These concepts include the design of interface components, layout, navigation, and user interaction.

Consider a simple calculator application with a graphical user interface:

1. **GUI Elements:** GUI concepts involve designing and using various elements such as buttons, text fields, labels, checkboxes, and menus to create a visually appealing and functional interface. In the calculator example, you would have buttons for numbers (0-9), arithmetic operations (+, -, *, /), and an equals (=) button.
2. **Layout:** GUI design often includes arranging elements in a visually pleasing and logically structured layout. For the calculator, you might arrange the buttons and display area in a grid layout to mimic the appearance of a physical calculator.
3. **Interactivity:** GUIs allow users to interact with the application. Users can click buttons, enter numbers, and perform calculations. In the calculator, when the user clicks on numbers and operators, they see the input and results displayed in the appropriate area.
4. **Feedback:** GUIs provide visual feedback to users, such as highlighting a button when it's clicked or displaying error messages. If a user tries to divide by zero in the calculator, an error message might appear.

5. **Navigation:** GUIs often include navigation elements like menus, tabs, and buttons that allow users to move between different parts of the application or access various features. In the calculator, you might have a menu with options to clear the input or exit the application.
6. **User Experience (UX):** GUI concepts aim to create a positive user experience by considering factors like simplicity, responsiveness, consistency in design, and accessibility for different devices and users.

GUI concepts are fundamental in modern software development, as they greatly impact how users interact with and perceive an application. Effective GUI design enhances user satisfaction and usability, ultimately contributing to the success of software products.

Components & Containers: Components and containers are fundamental concepts in graphical user interface (GUI) programming. They are used to build the structure and appearance of graphical user interfaces in software applications.

- **Components:** Components are individual user interface elements that can be placed on a GUI. These elements can be buttons, text fields, labels, checkboxes, radio buttons, etc. Each component serves a specific purpose and interacts with the user in some way.

  Example of Components: In a simple Java Swing application, you might use components like “Jbutton” for buttons, “JtextField” for text input, and “Jlabel” for displaying text or images.

  For instance, you could create a button with the code:

```
JButton submitButton = new JButton("Submit");
```

- **Containers:** Containers are GUI elements that can hold and manage other components. They provide a structure for arranging and organizing components on the user interface. Containers can be simple, like a panel, or complex, like a window or frame.

  Example of Containers: In Java Swing, a common container is the “Jframe”, which represents the application's main window. Inside the “Jframe”, you can add components such as buttons, text fields, and labels.

  For Example:

```
JFrame frame = new JFrame("My Application");
frame.setSize(400, 300);

JButton button = new JButton("Click Me");
frame.add(button);

frame.setVisible(true);
```

Containers provide structure and layout capabilities, allowing you to organize components effectively within a GUI. Components, on the other hand, are the interactive elements with which users can interact, and they are often placed inside containers to create functional user interfaces.

Abstract Window Toolkit (AWT) & Swing: Abstract Window Toolkit (AWT) and Swing are two Java libraries used for creating graphical user interfaces (GUIs) in Java applications.

- **AWT (Abstract Window Toolkit):** AWT is Java's original GUI library, and it is part of the core Java platform. AWT components are platform-dependent, meaning they rely on the underlying operating system's GUI elements. This

can lead to platform-specific differences in appearance and behavior. AWT provides a basic set of GUI components, including buttons, text fields, and windows. AWT is lightweight and suitable for simple GUIs, but it has limitations in terms of customization and look-and-feel consistency.

Example of AWT:

```
import java.awt.*;
import java.awt.event.*;

public class AWTExample {
    public static void main(String[] args) {
        Frame frame = new Frame("AWT Example");
        Button button = new Button("Click Me");

        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.out.println("Button clicked!");
            }
        });

        frame.add(button);
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
}
```

- **Swing:** Swing is a more modern and feature-rich GUI library built on top of AWT. Swing components are pure Java and not reliant on the underlying operating system's GUI components. This makes Swing applications more platform-independent and consistent in appearance. Swing provides a wide range of customizable and powerful GUI components, including advanced components like tables, trees, and dialogs. Swing supports a pluggable look-and-feel system, allowing you to change the appearance of your application easily.

Example of Swing:

```
import javax.swing.*;
import java.awt.event.*;

public class SwingExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Swing Example");
        JButton button = new JButton("Click Me");

        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                JOptionPane.showMessageDialog(frame, "Button clicked!");
            }
        });

        frame.add(button);
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
}
```

Windows and Frames: Windows and frames are terms commonly used in graphical user interface (GUI) programming to describe the main graphical containers that hold and manage the components of a graphical application.

- **Windows:** In GUI programming, a window generally refers to the top-level container that encapsulates an entire application or a significant part of it. Windows typically have a title bar with controls for minimizing, maximizing, and closing the window, and they can usually be moved and resized by the user. A window can contain one or more frames or other types of components.
- **Frames:** A frame is a specific type of window, often used as the main application window. Frames serve as the outermost containers for organizing and displaying components, such as buttons, labels, text fields, and more, that make up the application's graphical user interface. Frames are responsible for managing the layout and behavior of their components.

Creating a simple window (frame) with a button:

```
import javax.swing.*;

public class FrameExample {
    public static void main(String[] args) {
        // Create a frame (window)
        JFrame frame = new JFrame("My Window");

        // Create a button component
        JButton button = new JButton("Click Me");

        // Add the button to the frame
        frame.add(button);

        // Set frame size and make it visible
        frame.setSize(300, 200);
        frame.setVisible(true);

        // Handle window close event
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

Layout managers: Layout managers are an essential part of graphical user interface (GUI) programming. They are responsible for determining how components (such as buttons, labels, and text fields) are arranged and displayed within a container (such as a frame or panel) in a graphical application. Layout managers help ensure that your GUIs are properly structured, resizable, and adaptable to different screen sizes and resolutions.

Example of Layout Managers in Java Swing:

```
import javax.swing.*;
import java.awt.*;

public class LayoutManagerExample {
```

```
    public static void main(String[] args) {
        JFrame frame = new JFrame("Layout Manager Example");

        // Create a panel to hold components
        JPanel panel = new JPanel();

        // Create buttons
        JButton button1 = new JButton("Button 1");
        JButton button2 = new JButton("Button 2");
        JButton button3 = new JButton("Button 3");

        // Add buttons to the panel
        panel.add(button1);
        panel.add(button2);
        panel.add(button3);

        // Use a layout manager (FlowLayout in this case)
        panel.setLayout(new FlowLayout());

        // Add the panel to the frame
        frame.add(panel);

        // Set frame size and make it visible
        frame.setSize(300, 200);
        frame.setVisible(true);

        // Handle window close event
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

Different layout managers (e.g., GridLayout, BorderLayout, BoxLayout, GridBagLayout, etc.) have various rules for component arrangement, and you can choose the one that best suits your GUI design. Layout managers help create visually appealing and responsive GUIs by ensuring that components adapt to changes in window size and content.

Panels: Panels are containers or lightweight components used in graphical user interface (GUI) programming to group and organize other GUI components. Panels provide a way to structure and manage the layout of components within a larger container, such as a window (frame) or another panel.

Example of Panels in Java Swing:

```
import javax.swing.*;
import java.awt.*;

public class PanelExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Panel Example");

        // Create a main panel to hold other components
        JPanel mainPanel = new JPanel();

        // Create two subpanels to group components
```

```
        JPanel leftPanel = new JPanel();
        JPanel rightPanel = new JPanel();

        // Create buttons to add to the subpanels
        JButton button1 = new JButton("Button 1");
        JButton button2 = new JButton("Button 2");
        JButton button3 = new JButton("Button 3");

        // Add buttons to the left panel
        leftPanel.add(button1);
        leftPanel.add(button2);

        // Add a label to the right panel
        JLabel label = new JLabel("This is a label");
        rightPanel.add(label);
        rightPanel.add(button3);

        // Set layouts for the main panel and subpanels
        mainPanel.setLayout(new BorderLayout());
        leftPanel.setLayout(new FlowLayout());
        rightPanel.setLayout(new FlowLayout());

        // Add subpanels to the main panel
        mainPanel.add(leftPanel, BorderLayout.WEST);
        mainPanel.add(rightPanel, BorderLayout.EAST);

        // Add the main panel to the frame
        frame.add(mainPanel);

        // Set frame size and make it visible
        frame.setSize(400, 200);
        frame.setVisible(true);

        // Handle window close event
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

This results in a GUI with organized components, demonstrating how panels can help structure the layout of a graphical application. Panels are a powerful tool for creating complex and structured GUIs in which components are logically grouped and positioned.

Event-Driven Programming: Event-Driven Programming is a programming paradigm where the flow of a program's execution is determined by events. An event is a specific occurrence or change in the system's state, often generated by user interactions (e.g., mouse clicks, keyboard inputs) or changes in the environment (e.g., data updates, sensor readings). In event-driven programming, the program responds to these events by executing predefined event handlers or event-driven functions.

Example of Event-Driven Programming:

```
import javax.swing.*;
```

```
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class EventDrivenExample {
    public static void main(String[] args) {
        // Create a JFrame (window)
        JFrame frame = new JFrame("Event-Driven Example");

        // Create a JButton (a clickable button)
        JButton button = new JButton("Click Me");

        // Register an ActionListener to handle button clicks
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                // This code runs when the button is clicked
                JOptionPane.showMessageDialog(frame, "Button clicked!");
            }
        });

        // Add the button to the JFrame
        frame.add(button);

        // Set frame size and make it visible
        frame.setSize(300, 200);
        frame.setVisible(true);

        // Handle window close event
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

Delegation event model: The delegation event model is an event-handling mechanism used in many programming languages, including Java, to manage and respond to events generated by user interactions or other sources. In this model, components or objects delegate the responsibility of handling events to specialized event listener or handler objects. This design promotes separation of concerns, making code more modular and maintainable.

Example of the Delegation Event Model in Java:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class DelegationEventModelExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Delegation Event Model Example");

        JButton button = new JButton("Click Me");

        // Create an ActionListener (event listener) and register it with the button
        ActionListener buttonClickListener = new ActionListener() {
```

```
            public void actionPerformed(ActionEvent e) {
                // This code runs when the button is clicked
                JOptionPane.showMessageDialog(frame, "Button clicked!");
            }
        };

        // Register the ActionListener with the button
        button.addActionListener(buttonClickListener);

        frame.add(button);
        frame.setSize(300, 200);
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

Event classes: Event classes are Java classes that represent specific types of events that can occur in a program. These classes define the structure and information associated with an event, such as its type, source, and additional data. Event classes are used in event-driven programming to encapsulate and transmit information about events to event handlers or listeners.

Example:

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class ButtonClickListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        // Handle the button click event here
        if (e.getSource() == someButton) {
            // Perform specific actions for this button
        }
    }
}
```

Mouse Events: Mouse events are events generated by user interactions with a computer mouse. These events include actions such as clicking the mouse buttons, moving the cursor, dragging objects, and scrolling. Mouse events are commonly used in graphical user interfaces (GUIs) to capture and respond to user input.

Example:

```
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;

public class MyMouseListener extends MouseAdapter {
    public void mouseClicked(MouseEvent e) {
        // Handle mouse click event
        if (e.getSource() == myButton) {
            // Perform specific actions when the button is clicked
        }
```

```
    }
}
```

Keyboard events: Keyboard events are events generated by user interactions with a computer keyboard. These events include actions such as key presses, key releases, and key combinations (e.g., Ctrl+C). Keyboard events are commonly used in software applications to capture and respond to user input, such as text input and shortcut key combinations.

Example:

```
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;

public class MyKeyListener implements KeyListener {
    public void keyPressed(KeyEvent e) {
        // Handle key press event
        int keyCode = e.getKeyCode();
        if (keyCode == KeyEvent.VK_ENTER) {
            // Perform specific actions when Enter key is pressed
        }
    }

    // Implement other KeyListener methods: keyReleased and keyTyped
}
```

Using actions: Using actions is a design pattern in graphical user interface (GUI) programming that involves encapsulating user interface interactions (such as button clicks or menu selections) as discrete, reusable objects. Actions simplify event handling, enhance code modularity, and make it easier to associate user interface components with specific functionalities.

Example:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;

public class UsingActionsExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Using Actions Example");

        // Create an action
        Action action = new AbstractAction("Click Me") {
            public void actionPerformed(ActionEvent e) {
                // Handle the action (button click) here
                JOptionPane.showMessageDialog(frame, "Button clicked!");
            }
        };

        // Create a button and associate the action with it
        JButton button = new JButton(action);
```

```
        frame.add(button);
        frame.setSize(300, 200);
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

Components & Jcomponents: Components and JComponents are fundamental elements in graphical user interface (GUI) programming, particularly in Java Swing. They represent various user interface elements like buttons, labels, text fields, and more.

- **Components:** Components: In a broader sense, components refer to the basic building blocks of a GUI. They are the individual user interface elements that you can place on a GUI, such as buttons, labels, and checkboxes. Components provide specific functionality and appearance.
- **JComponents:** JComponents: JComponents are a specific type of component in Java Swing, an extension of the AWT (Abstract Window Toolkit) components. Swing components are lightweight and platform-independent, offering more features and customization options than their AWT counterparts.

Example:

```
import javax.swing.*;

public class ComponentExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Component Example");

        // Create a JButton (a JComponent)
        JButton button = new JButton("Click Me");

        // Create a Label (an AWT Component)
        Label label = new Label("Hello, World!");

        // Add components to the frame
        frame.add(button);
        frame.add(label);

        frame.setSize(300, 200);
        frame.setLayout(new FlowLayout());
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

JComponents like JButton offer enhanced functionality and look-and-feel compared to AWT components. However, you can use both types of components in Java GUI programming based on your specific needs.

Buttons: Buttons are user interface elements in graphical applications that users can interact with to trigger actions or perform specific tasks. Buttons typically have a visual representation on the screen, often as a labeled rectangle or icon, and can respond to user inputs like mouse clicks or keyboard presses.

```
import javax.swing.*;

public class ButtonExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Button Example");

        // Create a JButton (button component)
        JButton button = new JButton("Click Me");

        // Add the button to the frame
        frame.add(button);

        frame.setSize(300, 200);
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

Labels: Labels are user interface elements used to display text or images in graphical applications. They provide a way to present information or provide descriptions for other components. Labels are typically non-interactive and serve as a means to convey information to users.

```
import javax.swing.*;

public class LabelExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Label Example");

        // Create a JLabel (label component)
        JLabel label = new JLabel("Welcome to our application!");

        // Add the label to the frame
        frame.add(label);

        frame.setSize(300, 200);
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

Text fields: Text fields are user interface components in graphical applications that allow users to input and edit text. They provide an area where users can enter alphanumeric characters, numbers, or other types of textual information. Text fields are commonly used for data entry, search bars, and various forms in graphical user interfaces (GUIs).

```
import javax.swing.*;
```

```
public class TextFieldExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Text Field Example");

        // Create a JTextField (text field component)
        JTextField textField = new JTextField("Type something here");

        // Add the text field to the frame
        frame.add(textField);

        frame.setSize(300, 200);
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

Text areas: Text areas are user interface components in graphical applications that provide a larger and often multiline space for users to input, edit, or display text. Unlike text fields, which are typically single-line, text areas are designed for handling larger amounts of text, such as paragraphs or multiple lines of input.

```
import javax.swing.*;

public class TextAreaExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Text Area Example");

        // Create a JTextArea (text area component)
        JTextArea textArea = new JTextArea("Type something here\nand here...");

        // Add the text area to a JScrollPane for scrolling (if needed)
        JScrollPane scrollPane = new JScrollPane(textArea);
        frame.add(scrollPane);

        frame.setSize(300, 200);
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

Dialog boxes: Dialog boxes are user interface components in graphical applications that pop up as separate windows or overlays to interact with users and request specific information or decisions. They are often used to gather user input, display messages, or prompt users for confirmation or options. Dialog boxes are typically modal, meaning they block interaction with the rest of the application until they are closed.

```
import javax.swing.*;

public class DialogBoxExample {
    public static void main(String[] args) {
```

```
        JFrame frame = new JFrame("Dialog Box Example");

        // Create a button that shows a dialog box when clicked
        JButton button = new JButton("Show Dialog");
        button.addActionListener(e -> {
            // Display a simple message dialog
            JOptionPane.showMessageDialog(frame, "This is a message dialog.", "Dialog
            Example", JOptionPane.INFORMATION_MESSAGE);
        });

        frame.add(button);
        frame.setSize(300, 200);
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

Checkboxes & Radio buttons: Checkboxes and radio buttons are user interface elements in graphical applications used for making selections or choices. They are often used in forms or settings to allow users to indicate their preferences or select from multiple options.

- **Checkboxes:** Checkboxes are typically used when users can make multiple selections from a list of options. Each checkbox represents an independent choice, and users can check or uncheck them as needed. Checkboxes are suitable for scenarios where multiple options can be chosen simultaneously.
- **Radio buttons:** Radio buttons, also known as option buttons, are used when users need to select one option from a group of mutually exclusive choices. In a group of radio buttons, only one option can be selected at a time. Radio buttons are used for situations where users should make a single selection from a predefined set of options.

Example:

```
import javax.swing.*;

public class CheckBoxRadioButtonExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Checkbox and Radio Button Example");

        // Create checkboxes
        JCheckBox checkBox1 = new JCheckBox("Option 1");
        JCheckBox checkBox2 = new JCheckBox("Option 2");

        // Create a button group for radio buttons
        ButtonGroup buttonGroup = new ButtonGroup();
        JRadioButton radioButton1 = new JRadioButton("Choice 1");
        JRadioButton radioButton2 = new JRadioButton("Choice 2");

        // Add radio buttons to the button group
        buttonGroup.add(radioButton1);
        buttonGroup.add(radioButton2);

        frame.setLayout(new BoxLayout(frame.getContentPane(), BoxLayout.Y_AXIS));
        frame.add(checkBox1);
```

```
        frame.add(checkBox2);
        frame.add(radioButton1);
        frame.add(radioButton2);

        frame.setSize(300, 200);
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

Checkboxes and radio buttons are essential for user input scenarios where you want users to specify their choices or preferences within a GUI application.

Menus: Menus are user interface components in graphical applications that provide a structured and organized way to present a list of options or actions to users. Menus are commonly used to group related functions, commands, or settings in an easily accessible manner. They are often found in the form of drop-down menus, context menus, or menu bars within the application's graphical interface.

```
import javax.swing.*;

public class MenuExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Menu Example");

        // Create a menu bar
        JMenuBar menuBar = new JMenuBar();

        // Create menus
        JMenu fileMenu = new JMenu("File");
        JMenu editMenu = new JMenu("Edit");

        // Create menu items
        JMenuItem openMenuItem = new JMenuItem("Open");
        JMenuItem saveMenuItem = new JMenuItem("Save");
        JMenuItem cutMenuItem = new JMenuItem("Cut");
        JMenuItem copyMenuItem = new JMenuItem("Copy");
        JMenuItem pasteMenuItem = new JMenuItem("Paste");

        // Add menu items to menus
        fileMenu.add(openMenuItem);
        fileMenu.add(saveMenuItem);
        editMenu.add(cutMenuItem);
        editMenu.add(copyMenuItem);
        editMenu.add(pasteMenuItem);

        // Add menus to the menu bar
        menuBar.add(fileMenu);
        menuBar.add(editMenu);

        frame.setJMenuBar(menuBar);
        frame.setSize(300, 200);
        frame.setVisible(true);
```

```
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

Menus are a fundamental part of many graphical applications, providing a convenient way for users to access various features and functionalities.

J-Slider: A J-Slider is a user interface component in Java Swing used to allow users to select a value from a continuous range by dragging a slider thumb along a track. It's commonly used to set preferences, adjust settings, or choose values within a specified range, such as volume control or selecting a numeric value.

```
import javax.swing.*;

public class JSliderExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JSlider Example");

        // Create a JSlider
        JSlider slider = new JSlider(JSlider.HORIZONTAL, 0, 100, 50); // Minimum: 0,
        Maximum: 100, Initial: 50

        // Add the slider to the frame
        frame.add(slider);

        frame.setSize(300, 100);
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

J-Tabbed pane: A JTabbedPane is a user interface component in Java Swing that allows you to organize and present multiple sets of components or content in a tabbed format. Each tab represents a distinct view or content area, and users can switch between tabs to access different information or functionalities within the same application window.

```
import javax.swing.*;

public class JTabbedPaneExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JTabbedPane Example");

        // Create a JTabbedPane
        JTabbedPane tabbedPane = new JTabbedPane();

        // Create panels for the tabs
        JPanel panel1 = new JPanel();
        JPanel panel2 = new JPanel();
        JPanel panel3 = new JPanel();

        // Add components to the panels (content for each tab)
```

```
        panel1.add(new JLabel("This is Tab 1 content."));
        panel2.add(new JLabel("Content for Tab 2 goes here."));
        panel3.add(new JLabel("Tab 3 has its own content."));

        // Add tabs with labels to the JTabbedPane
        tabbedPane.addTab("Tab 1", panel1);
        tabbedPane.addTab("Tab 2", panel2);
        tabbedPane.addTab("Tab 3", panel3);

        // Add the JTabbedPane to the frame
        frame.add(tabbedPane);

        frame.setSize(400, 200);
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

JTabbedPane is commonly used to create tabbed interfaces for organizing and presenting information or features in a user-friendly and space-efficient manner.

(Past Papers Q's)

Short Q's (2020)

Q1: Write purpose of copy constructor?

The purpose of a copy constructor is to create a new object that is a copy of an existing object, ensuring that the new object has the same state or attributes as the original.

Example:

```
class Person {
    private String name;

    public Person(String name) {
        this.name = name;
    }

    // Copy constructor
    public Person(Person other) {
        this.name = other.name;
    }
}
```

Q2: How do you differentiate a class from an object?

Class: A class is a blueprint or template that defines the structure and behavior of objects. It's like a recipe for creating objects.

Example: Think of a "Car" class that defines what attributes (e.g., color, model) and methods (e.g., start, stop) a car should have.

Object: An object is an actual instance created from a class. It represents a specific, tangible entity with its own data and functionality.

Example: An actual car, such as a "Red Honda Civic," is an object created from the "Car" class, with its own unique color, model, and the ability to start and stop.

Q3: What is Polymorphism?

Polymorphism is a concept in object-oriented programming that allows objects of different classes to be treated as objects of a common superclass. It enables you to perform actions on objects without knowing their specific types, making code more flexible and extensible.

Example: Imagine a "Shape" superclass with subclasses "Circle" and "Rectangle." Polymorphism allows you to have a list of shapes and call a common method like “calculateArea()” on each shape, regardless of whether it's a circle or rectangle. The specific implementation of “calculateArea()” will be invoked based on the actual object type, demonstrating polymorphism.

Q4: Give example of exception handling?

```
public class ExceptionHandlingExample {
    public static void main(String[] args) {
        try {
            int result = divide(10, 0); // Attempting to divide by zero
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Error: Division by zero is not allowed.");
        }
    }

    public static int divide(int numerator, int denominator) {
        return numerator / denominator;
    }
}
```

Q5: Give example of any overloaded operator of string class?

In Java, the String class does not support operator overloading. Instead, you use methods for string operations. Here's a short example using the ‘+’ operator for string concatenation:

```
String str1 = "Hello, ";
String str2 = "world!";
String result = str1 + str2; // Concatenating strings with the + operator
System.out.println(result); // Output: Hello, world!
```

While the + operator is commonly used for string concatenation, it's not operator overloading in the traditional sense, as seen in languages like C++. Java treats string concatenation as a special case and translates it into calls to the concat method behind the scenes.

Q6: Can sorting help in searching array's elements?

Sorting can help in searching for elements in an array more efficiently, especially when using binary search. Binary search relies on the array being sorted in order to quickly locate a specific element.

```
import java.util.Arrays;

public class SearchInSortedArray {
    public static void main(String[] args) {
        int[] sortedArray = {2, 4, 6, 8, 10, 12, 14};

        int target = 8;
        int index = Arrays.binarySearch(sortedArray, target);

        if (index >= 0) {
            System.out.println(target + " found at index " + index);
        } else {
            System.out.println(target + " not found in the array.");
        }
    }
}
```

Q7: What happens in case a user does not provide a constructor?

If a user does not provide a constructor in a Java class, a default constructor is automatically provided by the Java compiler. This default constructor takes no arguments and initializes the object's fields with default values (e.g., 0 for numeric types, null for objects).

```
public class MyClass {
    // No constructor provided by the user

    public void display() {
        System.out.println("Hello from MyClass!");
    }

    public static void main(String[] args) {
        MyClass obj = new MyClass();
        obj.display();
    }
}
```

Q8: Discuss the relative merits of using protected access vs. using private access in base classes?

Protected Access: Allows derived classes to access and modify members. Provides flexibility for customization. Use when you want to allow derived classes to extend the base class's behavior.

```
class Base {
    protected int value;

    protected void setValue(int newValue) {
        value = newValue;
    }
}

class Derived extends Base {
    public void modifyValue() {
        setValue(42); // Access and modify protected member
    }
}
```

Private Access: Restricts access to members within the base class only. Enhances encapsulation by hiding implementation details. Use when you want to hide the member from derived classes.

```
class Base {
    private int secretValue;

    public void setSecretValue(int newValue) {
        secretValue = newValue;
    }
}
```

Q9: What is the purpose of information hiding?

The purpose of information hiding is to hide the internal details of a class or module, providing a clean and controlled interface. This enhances security and maintainability.

Example: In a banking application, account balance should be hidden within a class and accessed only through methods like “getBalance()”, preventing direct manipulation of the balance from outside the class.

Q10: What do you mean by default member-wise assignment?

In Java, there's no concept of "default member-wise assignment" as in some other languages like C++. In Java, when you create a new object, it doesn't copy the state of another object member-wise by default. Instead, you generally use constructors, methods, or clone methods to create new objects with desired state.

```
class MyClass {
    private int value;

    public MyClass(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}

public class Main {
    public static void main(String[] args) {
        MyClass obj1 = new MyClass(42); // Creating obj1 with a specific value
        MyClass obj2 = new MyClass(obj1.getValue()); // Creating obj2 by explicitly
        setting its value

        System.out.println(obj2.getValue()); // Output: 42
    }
}
```

Q11: Differentiate between sequence container and associative container?

Sequence Containers: These containers store elements in a specific order, and elements can be accessed by their position or index within the container. Examples include lists and arrays.

```
List<String> myList = new ArrayList<>();
```

```
myList.add("Apple");
myList.add("Banana");
myList.add("Cherry");

String fruit = myList.get(1); // Accessing "Banana" by index
```

Associative Containers: These containers store elements in key-value pairs and provide efficient access to elements based on their keys. Examples include maps and dictionaries.

```
Map<String, Integer> myMap = new HashMap<>();
myMap.put("One", 1);
myMap.put("Two", 2);
myMap.put("Three", 3);

int value = myMap.get("Two"); // Accessing value 2 by key "Two"
```

Q12: When do we need to allocate memory dynamically?

Dynamic memory allocation is needed when you want to allocate memory for objects or data structures at runtime, and you don't know the size or number of elements in advance. This is common in situations where you need flexibility in managing memory.

```
List<Integer> dynamicList = new ArrayList<>(); // Memory is allocated dynamically
dynamicList.add(1);
dynamicList.add(2);
dynamicList.add(3);

// The list can grow or shrink as needed without manual memory management
```

Long Q's (2020)

Q1: Compare in detail the following access modifiers;

1. **Public**
2. **Private**
3. **Protected**

Let's compare the access modifiers public, private, and protected in detail with examples:

1. **"Public" Access Modifier:** Members with public access are accessible from anywhere, including outside the class, in other classes, and even in different packages. It provides the highest level of visibility and is often used for methods and fields that should be accessible to all.

   Example:

```
public class PublicExample {
    public int publicVar = 42;

    public void publicMethod() {
        System.out.println("This is a public method.");
    }
}
```

2. **"Private" Access Modifier:** Members with private access are only accessible within the class where they are declared. It provides the highest level of encapsulation, hiding the implementation details from external classes.

   Example:

```
public class PrivateExample {
    private int privateVar = 42;

    private void privateMethod() {
        System.out.println("This is a private method.");
    }
}
```

3. **"Protected" Access Modifier:** Members with protected access are accessible within the class, by derived classes (subclasses), and within the same package. It allows customization and extension of base classes by derived classes.

   Example:

```
public class ProtectedExample {
    protected int protectedVar = 42;

    protected void protectedMethod() {
        System.out.println("This is a protected method.");
    }
}

class Derived extends ProtectedExample {
    void modifyBase() {
        protectedVar = 99; // Access and modify protected member
    }
}
```

Q2: Write a simple program that overloads the Unary prefix and postfix ++ and – operators?

```
class MyNumber {
    private int value;

    public MyNumber(int value) {
        this.value = value;
    }

    // Overload the prefix increment operator (++)
    public MyNumber operatorIncrement() {
        return new MyNumber(++value);
    }

    // Overload the prefix decrement operator (--)
    public MyNumber operatorDecrement() {
        return new MyNumber(--value);
    }

    public int getValue() {
```

```
        return value;
    }
}

public class OperatorOverloadingExample {
    public static void main(String[] args) {
        MyNumber num = new MyNumber(5);

        // Prefix increment
        MyNumber incremented = ++num;
        System.out.println("Prefix Increment: " + incremented.getValue());
        // Output: 6

        // Prefix decrement
        MyNumber decremented = --num;
        System.out.println("Prefix Decrement: " + decremented.getValue());
        // Output: 5
    }
}
```

Q3: Write detailed note on Inheritance? Also, clearly mention the relationship between base class and derived class.

Inheritance is one of the fundamental concepts in object-oriented programming (OOP) that allows you to create a new class based on an existing class. The existing class is referred to as the base class or superclass, and the new class is called the derived class or subclass. Inheritance establishes a relationship where the derived class inherits attributes and behaviors (fields and methods) from the base class.

Here are some key points about inheritance:

1. **Code Reusability:** Inheritance promotes code reuse by allowing you to create a new class that inherits the properties and methods of an existing class. This avoids redundancy and makes your code more efficient and maintainable.
2. **Is-A Relationship:** Inheritance represents the "is-a" relationship. It means that a derived class is a specialized version of the base class. For example, if you have a base class “Vehicle”, a derived class “Car” is a specialized type of “Vehicle”.
3. **Access to Base Class Members:** Inheritance allows the derived class to access the public and protected members of the base class. Private members of the base class are not directly accessible in the derived class.

Example of Inheritance:

```
// Base class
class Vehicle {
    protected String brand;

    public Vehicle(String brand) {
        this.brand = brand;
    }

    public void startEngine() {
        System.out.println("Engine started for " + brand);
```

```
    }
}

// Derived class
class Car extends Vehicle {
    private int year;

    public Car(String brand, int year) {
        super(brand); // Call the base class constructor
        this.year = year;
    }

    public void drive() {
        System.out.println("Driving the " + year + " " + brand + " car.");
    }
}

public class InheritanceExample {
    public static void main(String[] args) {
        Car myCar = new Car("Toyota", 2022);

        myCar.startEngine(); // Inherited method from Vehicle
        myCar.drive();       // Method specific to Car
    }
}
```

Q4: Differentiate between virtual functions and pure virtual functions?

Virtual Functions in Java: In Java, all non-static methods are virtual by default, meaning they can be overridden by subclasses. When a method in a base class is overridden in a derived class, the derived class's implementation is called when the method is invoked on an object of the derived class.

Example:

```
class Base {
    public void display() {
        System.out.println("Display from Base");
    }
}

class Derived extends Base {
    @Override
    public void display() {
        System.out.println("Display from Derived");
    }
}

public class InheritanceExample {
    public static void main(String[] args) {
        Base obj = new Derived();
        obj.display(); // Calls the display() of Derived
    }
}
```

Pure Virtual Functions in Java: In Java, pure virtual functions are equivalent to abstract methods. Abstract methods are declared in an abstract class and do not have a default implementation in the abstract class. Classes containing abstract methods are abstract classes, and you cannot create objects of abstract classes.

Example:

```
abstract class Shape {
    public abstract void area(); // Abstract method (pure virtual)
}

class Circle extends Shape {
    @Override
    public void area() {
        System.out.println("Area of Circle");
    }
}

class Rectangle extends Shape {
    @Override
    public void area() {
        System.out.println("Area of Rectangle");
    }
}

public class AbstractExample {
    public static void main(String[] args) {
        Shape shape1 = new Circle();
        Shape shape2 = new Rectangle();

        shape1.area(); // Calls the area() of Circle
        shape2.area(); // Calls the area() of Rectangle
    }
}
```

Q5: Create a Distance class with instance variable feet and inches. Write suitable parameterized constructor, getter and setter functions. Also write another function to subtract two objects of class distance in such a way that if resultant inches are less than 1, feet should be decremented by 1 and inches incremented by 12 by using the statement dist3.sub(dist1, dist2); where dist1, dist2 and dist3 are objects of class Distance and sub is a user-defined function.

```
public class Distance {
    private int feet;
    private int inches;

    // Parameterized constructor
    public Distance(int feet, int inches) {
        this.feet = feet;
        this.inches = inches;
    }

    // Getter for feet
```

```
    public int getFeet() {
        return feet;
    }

    // Setter for feet
    public void setFeet(int feet) {
        this.feet = feet;
    }

    // Getter for inches
    public int getInches() {
        return inches;
    }

    // Setter for inches
    public void setInches(int inches) {
        this.inches = inches;
    }

    // Subtract two Distance objects
    public void sub(Distance dist1, Distance dist2) {
        int totalInches1 = dist1.feet * 12 + dist1.inches;
        int totalInches2 = dist2.feet * 12 + dist2.inches;
        int difference = totalInches1 - totalInches2;

        if (difference < 0) {
            this.feet = 0;
            this.inches = 0;
        } else {
            this.feet = difference / 12;
            this.inches = difference % 12;
        }
    }
}
```

Here's how you can use the "Distance" class:

```
public class DistanceTest {
    public static void main(String[] args) {
        Distance dist1 = new Distance(5, 8);
        Distance dist2 = new Distance(2, 6);
        Distance dist3 = new Distance(0, 0);

        // Subtract dist2 from dist1 and store the result in dist3
        dist3.sub(dist1, dist2);

        System.out.println("Result: " + dist3.getFeet() + " feet " + dist3.getInches()
        + " inches");
    }
}
```

Q6: Give example of overloading a binary operator.

In Java, you cannot directly overload binary operators like you can in languages like C++. However, you can achieve similar functionality by defining methods that mimic the behavior of binary operators. Here's an example of overloading the addition '+' operator for a custom class:

```
class ComplexNumber {
    private double real;
    private double imaginary;

    public ComplexNumber(double real, double imaginary) {
        this.real = real;
        this.imaginary = imaginary;
    }

    // Overload the + operator
    public ComplexNumber add(ComplexNumber other) {
        double newReal = this.real + other.real;
        double newImaginary = this.imaginary + other.imaginary;
        return new ComplexNumber(newReal, newImaginary);
    }

    public void display() {
        System.out.println(real + " + " + imaginary + "i");
    }
}

public class OperatorOverloadingExample {
    public static void main(String[] args) {
        ComplexNumber num1 = new ComplexNumber(3.0, 4.0);
        ComplexNumber num2 = new ComplexNumber(1.5, 2.5);

        ComplexNumber sum = num1.add(num2); // Overloaded addition
        System.out.print("Sum: ");
        sum.display(); // Output: 4.5 + 6.5i
    }
}
```

Q7: Create an Address class that used House#, Street#, and name of a City as data members of the Address class. Create another class Person that defines Name as its data member and uses the above Address class's data members as its data members. Use constructors (both with and without parameters) to initialize the data members of Person class and display functions to display the object of the Person class.

```
class Address {
    private String houseNumber;
    private String streetNumber;
    private String cityName;

    public Address(String houseNumber, String streetNumber, String cityName) {
        this.houseNumber = houseNumber;
        this.streetNumber = streetNumber;
        this.cityName = cityName;
```

```
    }

    public String toString() {
        return houseNumber + ", " + streetNumber + ", " + cityName;
    }
}

class Person {
    private String name;
    private Address address;

    // Constructor with parameters to initialize name and address
    public Person(String name, String houseNumber, String streetNumber, String
    cityName) {
        this.name = name;
        this.address = new Address(houseNumber, streetNumber, cityName);
    }

    // Constructor without parameters
    public Person() {
        this.name = "Unknown";
        this.address = new Address("N/A", "N/A", "N/A");
    }

    // Display function to print Person's details
    public void display() {
        System.out.println("Name: " + name);
        System.out.println("Address: " + address);
    }
}

public class AddressPersonExample {
    public static void main(String[] args) {
        // Create a Person object with parameters
        Person person1 = new Person("John Doe", "123", "Main Street", "New York");

        // Create a Person object with default values
        Person person2 = new Person();

        // Display details of Person objects
        System.out.println("Person 1 Details:");
        person1.display();

        System.out.println("\nPerson 2 Details:");
        person2.display();
    }
}
```

In the main method, we create two Person objects, one with parameters and one with default values, and display their details using the display method.

Q8: Define a class for a bank account that includes the following data members.

Name of the depositor, account number, type of account, and the balance amount in the account

The class also contains the following members functions:

1. **A constructor to assign initial value.**
2. **A constructor to assign values from the user.**
3. **Deposit function to deposit some accounts. It should display an error message if the deposited amount is less than or equal to 0, and not add that amount to the balance amount.**
4. **Withdraw function to withdraw amount from an account. It should display an error message if the withdrawn amount is greater than the balance amount in the account or if the withdrawn amount is negative.**

```
import java.util.Scanner;

class BankAccount {
    private String depositorName;
    private int accountNumber;
    private String accountType;
    private double balanceAmount;

    // Constructor to assign initial values
    public BankAccount() {
        depositorName = "Unknown";
        accountNumber = 0;
        accountType = "Savings";
        balanceAmount = 0.0;
    }

    // Constructor to assign values from the user
    public BankAccount(String name, int accountNumber, String type, double
    initialBalance) {
        depositorName = name;
        this.accountNumber = accountNumber;
        accountType = type;
        balanceAmount = initialBalance;
    }

    // Deposit function
    public void deposit(double amount) {
        if (amount <= 0) {
            System.out.println("Error: Invalid deposit amount. Please deposit a
            positive amount.");
        } else {
            balanceAmount += amount;
            System.out.println("Deposit successful. New balance: " + balanceAmount);
        }
    }

    // Withdraw function
    public void withdraw(double amount) {
        if (amount <= 0) {
            System.out.println("Error: Invalid withdrawal amount. Please withdraw a
            positive amount.");
        } else if (amount > balanceAmount) {
```

```
            System.out.println("Error: Insufficient balance. Withdrawal not
            allowed.");
        } else {
            balanceAmount -= amount;
            System.out.println("Withdrawal successful. New balance: " +
            balanceAmount);
        }
    }

    // Display account details
    public void displayAccountDetails() {
        System.out.println("Account Details:");
        System.out.println("Depositor Name: " + depositorName);
        System.out.println("Account Number: " + accountNumber);
        System.out.println("Account Type: " + accountType);
        System.out.println("Balance Amount: " + balanceAmount);
    }
}

public class BankAccountExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // Create a BankAccount object with user input
        System.out.print("Enter Depositor Name: ");
        String name = scanner.nextLine();
        System.out.print("Enter Account Number: ");
        int accountNumber = scanner.nextInt();
        scanner.nextLine(); // Consume the newline character
        System.out.print("Enter Account Type: ");
        String type = scanner.nextLine();
        System.out.print("Enter Initial Balance: ");
        double initialBalance = scanner.nextDouble();

        BankAccount account = new BankAccount(name, accountNumber, type,
        initialBalance);

        // Deposit and withdraw operations
        account.deposit(500.0);
        account.withdraw(200.0);
        account.withdraw(-50.0); // Invalid withdrawal
        account.deposit(-100.0); // Invalid deposit

        // Display account details
        account.displayAccountDetails();

        scanner.close();
    }
}
```

Some Additional Q's

Introduction of Java: Java is a versatile and widely-used programming language known for its platform independence. It was developed by Sun Microsystems (now owned by Oracle) and is characterized by its simplicity, portability, and robustness. Java is commonly used for developing a wide range of applications, from web and mobile apps to desktop software and embedded systems. It's known for its "Write Once, Run Anywhere" capability, which means Java programs can run on different platforms without modification. Java's key features include object-oriented programming, a rich standard library, and strong community support.

Introduction of OOP in Java: Object-Oriented Programming (OOP) in Java is a programming paradigm that revolves around the concept of objects. It is characterized by the following key principles:

1. **Objects:** Everything in OOP is treated as an object, which is an instance of a class. Objects have both data (attributes) and behaviors (methods).
2. **Classes:** Classes are blueprint templates for creating objects. They define the structure (attributes and methods) that objects of that class will have.
3. **Encapsulation:** Encapsulation is the concept of bundling data (attributes) and methods that operate on that data into a single unit (class). It enforces access control and data hiding.
4. **Inheritance:** Inheritance allows a class (subclass or derived class) to inherit attributes and methods from another class (superclass or base class). It promotes code reuse and the "is-a" relationship.
5. **Polymorphism:** Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables method overriding and dynamic method binding.
6. **Abstraction:** Abstraction is the process of simplifying complex reality by modeling classes based on the essential properties and behaviors relevant to a problem.
7. **Modularity:** OOP promotes modularity by breaking down complex systems into smaller, more manageable classes and objects.

Method Overloading: Method overloading involves multiple methods with the same name in the same class, differing in parameters.

```
class Calculator {
    public int add(int a, int b) {
        return a + b;
    }

    public double add(double a, double b) {
        return a + b;
    }
}
```

Method Overriding: Method overriding involves providing a new implementation of a method in a subclass, which replaces the superclass's implementation when called on objects of the subclass.

```
class Animal {
    public void makeSound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Dog barks");
    }
}
```

ASCII: ASCII (American Standard Code for Information Interchange) was developed in the early 1960s. It became an official standard in 1967 and has since played a significant role in computer systems and data communication. ASCII is a character encoding standard that represents text and control characters in computers. It uses 7 or 8 bits to encode characters and was widely adopted in early computer systems. It contains 128 characters, which can be represented using 7 bits (0 to 127).

Unicode: Unicode was first conceived in the late 1980s and was formally established in 1991 when the Unicode Consortium was founded. Unicode is a standardized character encoding system that assigns unique numeric codes to represent text characters and symbols from virtually all writing systems in the world. Unicode currently includes over 143,000 characters, covering a wide range of languages, symbols, and special characters.

Why ASCII Is Still Used Though Unicode Is Available:

1. **Legacy Systems**: Many older systems and software still rely on ASCII encoding, and migrating them to Unicode can be complex and costly.
2. **Efficiency:** For English text and simple data, ASCII encoding is more space-efficient than Unicode, which uses 16 bits per character.
3. **Compatibility:** ASCII is a subset of Unicode, ensuring compatibility and ease of conversion when needed.
4. **Simplicity:** ASCII is simpler and faster to process, making it suitable for certain applications.
