INTERNPE INTERNSHIP/TRAINING PROGRAM
JAVA 2 (NOTES CONTENT)

6. Inheritance in Object-Oriented Programming (OOP) with Java:

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows one class to inherit the properties and behaviors (fields and methods) of another class. Inheritance promotes code reuse, modular design, and the creation of a hierarchical class structure.

6.1 Basic Concepts:

- Inheritance creates a parent-child relationship between classes, often referred to as the superclass (parent) and subclass (child) relationship.
- The subclass inherits the members (fields and methods) of the superclass.
- The subclass can extend the functionality of the superclass by adding new members or overriding existing ones.

6.2 Syntax:

```
class Superclass {
    // Fields and methods
}

class Subclass extends Superclass {
    // Additional fields and methods
}
```

6.3 Access Modifiers:

- Access modifiers control the visibility of class members within the inheritance hierarchy.
- Public members of the superclass are inherited and accessible in the subclass.
- Private members of the superclass are not directly accessible in the subclass.

6.4 Overriding Methods:

- Subclasses can provide their own implementation of methods inherited from the superclass. This is called method overriding.
- Use the `@Override` annotation to indicate that a method is intended to override a superclass method.
- Overriding allows customization of behavior while maintaining a consistent interface.

6.5 `super` Keyword:

- The `super` keyword is used in the subclass to refer to the superclass.
- It can be used to call the superclass constructor or invoke overridden methods from the superclass.

6.6 Constructors in Inheritance:

- Subclasses can have constructors that call the superclass constructor using the `super()` statement.
- The superclass constructor is usually called before the subclass constructor body.

6.7 Types of Inheritance:

- Single Inheritance: A class can inherit from only one superclass.
- Multiple Inheritance: A class can inherit from multiple superclasses (Java doesn't support this directly to avoid complications).
- Multilevel Inheritance: A class can be a subclass of another subclass, creating a chain of inheritance.

6.8 Abstract Classes:

- Abstract classes cannot be instantiated; they are meant to be subclassed.
- They can define abstract methods (methods without a body) that must be implemented by subclasses.
- Abstract classes provide a common interface and shared behavior for subclasses.

6.9 `final` Keyword:

- The `final` keyword can be applied to classes, methods, and fields.
- A `final` class cannot be subclassed.
- A `final` method cannot be overridden by subclasses.
- A `final` field is a constant and cannot be modified once initialized.

6.10 Object Class:

- In Java, all classes are implicitly subclasses of the `Object` class.
- The `Object` class provides basic methods like `toString()`, `equals()`, and `hashCode()` that can be overridden in subclasses.

6.11 Benefits of Inheritance:

- Code Reusability: Inherited members can be reused in multiple subclasses.
- Hierarchical Organization: Classes can be grouped into a hierarchy based on shared characteristics.
- Polymorphism: Subclasses can be treated as instances of their superclass, allowing flexible code design**.**

6.12 Considerations:

- Overuse of inheritance can lead to a complex and tightly coupled codebase.
- Prefer composition (combining existing classes) over inheritance when appropriate.
- Choose inheritance when there's a clear "is-a" relationship between classes.

Inheritance is a powerful tool in Java's object-oriented paradigm that facilitates code organization, reusability, and flexibility. By carefully designing class hierarchies and utilizing inheritance wisely, developers can create maintainable and extensible software systems.

7. Polymorphism in Object-Oriented Programming (OOP) with Java

1. Introduction to 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 super class. This enables a single interface to be used for a general class of actions, making code more flexible and extensible.

2. Types of Polymorphism: There are two main types of polymorphism in Java:

a. Compile-Time (Static) Polymorphism: Also known as method overloading, compile-time polymorphism occurs when multiple methods in the same class have the same name but different parameter lists. The appropriate method to be called is determined at compile-time based on the method's parameters.

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

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

b. Run-Time (Dynamic) Polymorphism: Also known as method overriding, run-time polymorphism involves creating a subclass that provides a specific implementation of a method defined in its superclass. The method to be executed is determined at run-time based on the actual object type.

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

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

```
class Square extends Shape {
    @Override
    void draw() {
        System.out.println("Drawing a square");
    }
```

3. Method Overriding: Method overriding is a key feature of run-time polymorphism. In Java, a subclass can provide its own implementation of a method that is already defined in its superclass. The `@Override` annotation is used to indicate that a method is intended to override a superclass method.

4. Dynamic Method Dispatch: Dynamic method dispatch is the mechanism that enables the invocation of overridden methods based on the actual type of the object, not just its reference type. This is a crucial aspect of achieving run-time polymorphism.

5. Polymorphism in Interfaces: Interfaces define a contract that classes implementing them must adhere to. Interfaces allow multiple classes to provide their own implementations of the methods defined in the interface, allowing for polymorphic behavior.

```
interface Eater {
    void eat();
}

class Human implements Eater {
    @Override
    public void eat() {
        System.out.println("Eating with a fork and knife");
    }
}

class Dog implements Eater {
    @Override
    public void eat() {
        System.out.println("Eating with enthusiasm");
    }
}
```

6. Advantages of Polymorphism:

- Code reusability: Polymorphism promotes reuse of common behaviors across different classes.
- Flexibility: New classes can be easily added without affecting existing code, as long as they adhere to the common interface.
- Maintainability: Changes in the behavior of a common interface affect all implementing classes uniformly.

7. Real-World Examples: Polymorphism is widely used in Java programming. For instance, the Java Collections Framework relies heavily on polymorphism to provide a consistent interface for various collection types (List, Set, Map). Also, graphical user interface (GUI) libraries often utilize polymorphism to handle different types of components in a unified way.

8. Summary: Polymorphism is a key concept in OOP that enables flexible and extensible code by allowing objects of different classes to be treated as objects of a common super class. Compile-time polymorphism involves method overloading, while run-time polymorphism involves method overriding and dynamic method dispatch. Interfaces also play a significant role in achieving polymorphism by defining common contracts for implementing classes.

8. Encapsulation in Object-Oriented Programming (OOP) with Java:

Encapsulation is one of the four fundamental principles of object-oriented programming (OOP), along with inheritance, polymorphism, and abstraction. It refers to the practice of bundling data (attributes or fields) and methods (functions) that operate on the data into a single unit called a class. The key idea behind encapsulation is to control the access to the internal state of an object, ensuring that

it can only be modified through well-defined methods while hiding the implementation details from the outside.

In Java, encapsulation is achieved through the use of access modifiers and getter and setter methods.

Access Modifiers: Java provides four main access modifiers that determine the visibility and accessibility of class members (fields and methods):

private: Members declared as private are accessible only within the same class. They are not visible outside the class, including in subclasses.
default (no modifier): Members without any access modifier are accessible within the same package. They are not visible outside the package.
protected: Members declared as protected are accessible within the same package and by subclasses, even if they are in a different package.
public: Members declared as public are accessible from anywhere in the program.

Getter and Setter Methods: Getter and setter methods are used to access and modify the private fields of a class, providing controlled access to the object's internal state.

- Getter Methods: These methods are used to retrieve the values of private fields.
- java
- Copy code

```
public ReturnType getFieldName() {

    return fieldName;
```

}

- Setter Methods: These methods are used to modify the values of private fields.
- java
- Copy code

```
public void setFieldName(ParameterType newValue) {

    this.fieldName = newValue;

}
```

-

Benefits of Encapsulation:

Data Hiding: Encapsulation hides the implementation details of a class, allowing the internal state to be protected and only accessed through well-defined methods. This prevents unauthorized or unintended modifications to the data.

Controlled Access: By providing getter and setter methods, you can control how the data is accessed and modified. This enables you to enforce business rules and validations before allowing changes to the data.

Flexibility: Encapsulation allows you to change the internal implementation of a class without affecting the code that uses it. As long as the public methods remain consistent, other code remains unaffected.

Code Maintenance: Encapsulation makes it easier to maintain and debug code since the impact of changes is localized to the class itself.

Example:

java

Copy code

```
public class Student {

    private String name;

    private int age;



    public String getName() {

        return name;

    }



    public void setName(String name) {

        if (name != null && !name.isEmpty()) {

            this.name = name;

        }

    }

    public int getAge() {

        return age;

    }
```

```
    public void setAge(int age) {
        if (age >= 0) {
            this.age = age;
        }
    }
}
```

In this example, the `Student` class encapsulates the `name` and `age` fields with getter and setter methods. The setter methods include validation logic to ensure that valid values are set for the fields.

Encapsulation is a crucial concept in Java and object-oriented programming as it promotes data integrity, code organization, and maintainability. It helps in building reliable and robust software systems by controlling the access to an object's internal state and providing a clear and controlled interface for interacting with that state.

9. Abstraction in Object-Oriented Programming (OOP) with Java:

Abstraction is a fundamental concept in object-oriented programming (OOP) that focuses on simplifying complex reality by modeling classes based on their essential characteristics. It involves creating abstract classes and interfaces to define a blueprint for other classes to follow. Abstraction allows you to hide the implementation details of a class and provide a clear and simplified interface for interacting with objects.

Key Concepts:

1. Abstract Classes: An abstract class is a class that cannot be instantiated on its own and is meant to be subclassed. It serves as a blueprint for other classes and can contain both abstract (unimplemented) methods and concrete (implemented) methods. Abstract methods have no body and are meant to be overridden by subclasses. Abstract classes can have fields, constructors, and methods just like regular classes.

Example:

```
abstract class Shape {

    double area; // Field


    abstract void calculateArea(); // Abstract method


    void displayArea() {

        System.out.println("Area: " + area);

    }
```

2. Interfaces: An interface is a collection of abstract methods that define a contract for classes to follow. Unlike classes, a class can implement multiple interfaces. Interfaces are used to achieve multiple inheritance and ensure that implementing classes provide specific behaviors. All methods in an interface are implicitly public and abstract.

Example:

```
interface Drawable {
    void draw(); // Abstract method
}

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

3. Benefits of Abstraction:

- Modularity: Abstraction promotes modularity by separating the interface from the implementation. This makes it easier to update or modify the implementation without affecting the interface.

- Code Reusability: Abstract classes and interfaces encourage code reuse. Subclasses can extend abstract classes and implement interfaces to inherit common behavior.
- Flexibility: Abstraction allows you to change the implementation of a class without affecting the code that uses the class's interface.
- Encapsulation: Abstraction complements encapsulation by controlling access to class members through access modifiers. This helps in maintaining data integrity and security.
- Polymorphism: Abstraction is a key factor in achieving polymorphism, as it enables different classes to be treated as instances of a common super class or interface.

4. Choosing Between Abstract Classes and Interfaces:

- Use Abstract Classes when you want to provide a common base class with some default implementations for subclasses to extend. You can have fields and constructors in abstract classes.
- Use Interfaces when you want to define a contract that multiple unrelated classes can implement. Interfaces are particularly useful when you want to enable classes to inherit behavior from multiple sources.

5. Java 8 and Default Methods:

Java 8 introduced default methods in interfaces, allowing you to provide a default implementation for methods. This was done to ensure backward compatibility when adding new methods to existing interfaces.

Example:

```
interface Greeting {

    default void sayHello() {

        System.out.println("Hello from Greeting!");

    }
```

```
}
```

6. Java 9 and Private Methods in Interfaces:

Java 9 introduced the concept of private methods in interfaces. These methods can be used to encapsulate common functionality within an interface without exposing them to implementing classes.

Example:

```
interface Calculation {

    default double calculate() {

        return calculateIntermediate() * 2;

    }



    private double calculateIntermediate() {

        // ... perform calculation

    }
}
```

In Summary:

Abstraction in Java helps you design more modular, maintainable, and flexible code by providing a clear separation between the interface and implementation of classes. It encourages code reuse, enhances polymorphism, and supports multiple inheritance through interfaces. Abstract

classes and interfaces are essential tools for building robust and extensible object-oriented systems.

10. Exception Handling:

Exception handling in Java is a mechanism to gracefully manage and handle runtime errors and exceptional situations that can occur during program execution. Instead of crashing the program, exceptions allow developers to handle errors in a controlled manner.

Java's exception handling is based on the concept of try-catch blocks:

- try: This block contains the code that might throw an exception.
- catch: This block catches and handles exceptions. It specifies the type of exception it can catch and provides a code block to handle the exception.
- finally: This optional block is used to specify code that should be executed regardless of whether an exception occurred or not. For example, it's commonly used to release resources.

Here's an example of exception handling in Java:

```
try {
```

```
    int result = 10 / 0; // This will throw an ArithmeticException
    System.out.println("Result: " + result);
} catch (ArithmeticException e) {
    System.out.println("An error occurred: " + e.getMessage());
} finally {
    System.out.println("Finally block executed.");
}
```

In this example:

- The `try` block attempts to perform a division operation that will cause an exception (`ArithmeticException`) due to division by zero.
- The `catch` block catches the exception and prints an error message.
- The `finally` block always executes, regardless of whether an exception occurred. It's used for cleanup tasks.

Java provides a hierarchy of exception classes, with `java.lang.Exception` being the root class. Common exceptions include `NullPointerException`, `ArrayIndexOutOfBoundsException`, `FileNotFoundException`, and more. You can catch specific exceptions or use more general catch blocks to handle a broader range of exceptions.

Object-Oriented Programming (OOP) in Java:

Java is a powerful object-oriented programming language, which means it revolves around the concept of objects. OOP is a programming paradigm that organizes data and behaviors into reusable structures called classes and objects. Key concepts of OOP include:

Classes and Objects:

- A class is a blueprint that defines the properties (attributes/fields) and behaviors (methods) that objects of that class will have.
- An object is an instance of a class, with its own set of values for attributes.

Inheritance:

- Inheritance allows a new class (subclass/derived class) to inherit attributes and behaviors from an existing class (superclass/base class).
- It promotes code reuse and hierarchical organization.

Polymorphism:

- Polymorphism allows objects of different classes to be treated as objects of a common superclass through inheritance.
- Method overriding enables subclasses to provide their own implementations of methods defined in the superclass.

Encapsulation:

- Encapsulation restricts direct access to certain attributes of a class by using access modifiers (private, protected, public).
- Public methods can be used to access or modify private fields, ensuring controlled data manipulation.

Abstraction:

- Abstraction hides the complex implementation details of a class and exposes only the essential features through methods and interfaces.
- Abstract classes and interfaces are used to define common properties and methods that subclasses must implement.

Association and Composition:

- Classes can be associated with each other through composition (one class is part of another) or association (one class uses another).

Constructor and Destructor:

- Constructors are special methods that initialize objects when they are created.

- Java automatically manages memory through garbage collection, so there are no explicit destructors.

Java's support for OOP makes it well-suited for building modular, extensible, and maintainable software applications. It encourages better organization of code, promotes reusability, and enhances collaboration among developers.

Remember that becoming proficient in both exception handling and OOP requires practice and practical implementation.