Unit II :

Inheritance: Basic concepts - Types of inheritance - Member access rules - Usage of this and Super key word - Method Overloading - Method overriding - Abstract classes - Dynamic method dispatch - Usage of final keyword.

Packages: Definition - Access Protection - Importing Packages.

Interfaces: Definition – Implementation – Extending Interfaces.

Exception Handling: *try – catch - throw - throws – finally* – Built-in exceptions - Creating own Exception classes.

Java Inheritance: Basic Concepts and Types of Inheritance

Inheritance is one of the fundamental concepts of Object-Oriented Programming (OOP) in Java. It allows a class to inherit properties and behaviors (fields and methods) from another class. Inheritance helps achieve **reusability** and **method overriding**.

Basic Concepts of Inheritance in Java

- **Superclass (Parent Class):** The class whose properties and methods are inherited.
- **Subclass (Child Class):** The class that inherits the properties and methods of the superclass.
- **extends keyword:** In Java, a subclass uses the extends keyword to inherit from a superclass.

Key points of inheritance in Java:

- A **child class** can inherit methods and fields from the **parent class**, but it cannot inherit constructors.
- A subclass can **extend** only one superclass (Java does not support multiple inheritance via classes).
- A subclass can **override** methods from the parent class to provide specific implementations.

Example of Inheritance in Java:

```
class Animal {
    void sound() {
        System.out.println("Animals make sounds");
    }
}

class Dog extends Animal {
    @Override
    void sound() {
```

```
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.sound(); // Outputs: Dog barks
    }
}
```

In this example:

- Dog is the subclass that inherits the sound() method from the Animal superclass.
- The sound() method is **overridden** in the Dog class to provide a more specific implementation.

Types of Inheritance in Java

Java supports several types of inheritance. Here are the main ones:

1. **Single Inheritance:**
    - A class inherits from only one superclass.
    - This is the most basic type of inheritance.

    Example:

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

    class Dog extends Animal {
        void makeSound() {
            System.out.println("Bark");
        }
    }
    ```

2. **Multilevel Inheritance:**
    - In multilevel inheritance, a class inherits from another class, which itself is a subclass of another class.
    - It forms a chain of inheritance.

    Example:

    ```
    class Animal {
        void makeSound() {
    ```

```
        System.out.println("Animal sound");
    }
}

class Dog extends Animal {
    void makeSound() {
        System.out.println("Bark");
    }
}

class Puppy extends Dog {
    void makeSound() {
        System.out.println("Puppy yelps");
    }
}
```

3. **Hierarchical Inheritance:**
    - In this type, multiple classes inherit from a single parent class.

Example:

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

class Dog extends Animal {
    void makeSound() {
        System.out.println("Bark");
    }
}

class Cat extends Animal {
    void makeSound() {
        System.out.println("Meow");
    }
}
```

4. **Multiple Inheritance (Through Interfaces):**
    - Java does not support multiple inheritance through classes, but it does support it through **interfaces**. A class can implement multiple interfaces, inheriting behavior from multiple sources.

Example:

```
interface Animal {
    void makeSound();
}

interface Pet {
    void play();
}

class Dog implements Animal, Pet {
    public void makeSound() {
        System.out.println("Bark");
    }

    public void play() {
        System.out.println("Dog plays fetch");
    }
}
```

In this example, Dog implements both Animal and Pet interfaces.

5. **Hybrid Inheritance:**
    - This is a combination of more than one type of inheritance (e.g., hierarchical and multiple inheritance). Java doesn't allow hybrid inheritance through classes, but it can be done through interfaces.

Example:

```
interface Animal {
    void makeSound();
}

interface Pet {
    void play();
}

class Dog implements Animal, Pet {
    public void makeSound() {
        System.out.println("Bark");
    }

    public void play() {
        System.out.println("Dog plays fetch");
    }
}
```

```
class GoldenRetriever extends Dog {
    void showAffection() {
        System.out.println("Golden Retriever shows affection");
    }
}
```

In this example, GoldenRetriever is a subclass of Dog, and Dog implements two interfaces, Animal and Pet.

Key Points to Remember:

- **Single Inheritance:** One class inherits from another.
- **Multilevel Inheritance:** Inheritance occurs in a chain (class A → class B → class C).
- **Hierarchical Inheritance:** Multiple classes inherit from one parent class.
- **Multiple Inheritance:** Supported through interfaces (not through classes).
- **Hybrid Inheritance:** A mix of different types of inheritance (primarily through interfaces).

Java's inheritance system helps build modular, reusable, and organized code. It also allows for method overriding, where a subclass can provide its own implementation of methods defined in the superclass.

Member Access Rules in Java

In Java, access to class members (fields, methods, constructors) is governed by **access modifiers** and certain rules for the this and super keywords.

Access Modifiers in Java:

1. **public**: The member is accessible from any other class.
2. **protected**: The member is accessible within the same package or by subclasses.
3. **default (no modifier)**: The member is accessible only within the same package.
4. **private**: The member is accessible only within the class it is defined in.

These access modifiers control the visibility and accessibility of class members.

Usage of this Keyword

The this keyword in Java refers to the **current instance** of the class. It is commonly used for the following purposes:

1. **Referring to instance variables**:
    - When local variables (like method parameters) have the same name as instance variables, this is used to differentiate between the two.

    Example:

    ```
    class Person {
        String name;
    ```

```
    // Constructor that uses 'this' to distinguish between parameter and field
    Person(String name) {
        this.name = name; // 'this.name' refers to the instance variable, 'name' is the parameter
    }

    void display() {
        System.out.println("Name: " + this.name); // Refers to the instance variable
    }
}
```

2. **Invoking current class methods**:
    - this can be used to call other methods in the current class.

Example:

```
class Calculator {
    void add(int a, int b) {
        this.printResult(a + b);  // Calling another method in the same class
    }

    void printResult(int result) {
        System.out.println("Result: " + result);
    }
}
```

3. **Passing the current object as a parameter**:
    - this can be passed to another method or constructor, allowing the current object to be passed as a parameter.

Example:

```
class Car {
    String model;

    Car(String model) {
        this.model = model;
    }

    void display(Car car) {
        System.out.println("Car model: " + car.model);
    }

    void showCar() {
```

```
        this.display(this);  // Passing the current object
    }
}
```

4. **Calling the constructor of the current class**:
    - this() can be used to invoke another constructor of the same class.

Example:

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

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

    Person(String name, int age) {
        this(name);  // Calls the constructor Person(String name)
        this.age = age;
    }
}
```

Usage of super Keyword

The super keyword in Java refers to the **parent (super) class** and is used in the following contexts:

1. **Accessing superclass methods**:
    - The super keyword is used to call methods of the parent class. It is useful when the subclass has overridden the method and you still want to call the parent class's version of the method.

Example:

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

class Dog extends Animal {
    void sound() {
        super.sound();  // Calls the parent class method
        System.out.println("Dog barks");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.sound();
    }
}
```

Output:

```
css
Copy code
Animal makes a sound
Dog barks
```

2. **Accessing superclass constructors**:
    - super() is used to invoke the constructor of the parent class. If no constructor is explicitly called in the subclass, the default constructor of the superclass is invoked automatically (if it exists).
    - You can use super() with parameters to call a parameterized constructor of the superclass.

Example:

```
class Animal {
    Animal(String name) {
        System.out.println("Animal's name: " + name);
    }
}

class Dog extends Animal {
    Dog(String name) {
        super(name);  // Calls the constructor of the superclass
        System.out.println("Dog's name: " + name);
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog("Buddy");
    }
}
```

Output:

```
rust
Copy code
```

Animal's name: Buddy
Dog's name: Buddy

3. **Accessing superclass instance variables**:
    - If a subclass has a field with the same name as a field in the parent class, you can use super to refer to the field of the parent class.

Example:

java
Copy code

```
class Animal {
   String color = "White";
}

class Dog extends Animal {
   String color = "Black";

   void display() {
      System.out.println("Dog's color: " + color);  // Refers to Dog's color
            System.out.println("Animal's color: " + super.color);   // Refers to
Animal's color
   }
}

public class Main {
   public static void main(String[] args) {
      Dog dog = new Dog();
      dog.display();
   }
}
```

Output:

mathematica
Copy code

```
Dog's color: Black
Animal's color: White
```

Note:

- **this Keyword**:
    - Refers to the current instance of the class.
    - Used to differentiate between instance variables and local variables with the same name.
    - Can be used to call other methods in the current class or pass the current object to other methods.

- Used to invoke another constructor of the same class using this().

- **super Keyword**:
  - Refers to the parent class and is used to access parent class methods, constructors, and instance variables.
  - It allows a subclass to invoke a method or constructor from the superclass, even if it has been overridden in the subclass.

Both this and super are crucial for working with inheritance and for managing and differentiating the members of the parent and child classes.

Method Overloading in Java

Method Overloading is a feature in Java where a class can have more than one method with the same name, but the methods differ in the number or type of their parameters. Method overloading helps increase the readability of the program and allows performing similar tasks using the same method name.

Key Points about Method Overloading:

- The methods must have the same name.
- The methods must differ in their parameter list (number or type of parameters).
- The return type can be different, but it alone is not enough to distinguish overloaded methods.
- Method overloading is resolved at compile time (also known as compile-time polymorphism).

Example of Method Overloading:

```
class Calculator {
    // Overloaded add method with two integer parameters
    int add(int a, int b) {
        return a + b;
    }

    // Overloaded add method with three integer parameters
    int add(int a, int b, int c) {
        return a + b + c;
    }

    // Overloaded add method with two double parameters
    double add(double a, double b) {
        return a + b;
    }
}

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

```
        Calculator calc = new Calculator();

        // Calls add(int, int)
        System.out.println(calc.add(2, 3));

        // Calls add(int, int, int)
        System.out.println(calc.add(2, 3, 4));

        // Calls add(double, double)
        System.out.println(calc.add(2.5, 3.5));
    }
}
```

Output:

Copy code

```
5
9
6.0
```

In this example:

- The add() method is overloaded with different parameter signatures.
- The appropriate add() method is called based on the arguments passed.

Method Overriding in Java

Method Overriding occurs when a subclass provides a specific implementation of a method that is already defined in its superclass. It is an example of **runtime polymorphism** (dynamic method dispatch), where the method that gets executed is determined at runtime based on the object type (not the reference type).

Key Points about Method Overriding:

- The method in the subclass must have the same name, return type, and parameter list as the method in the superclass.
- The overriding method can provide a different implementation in the subclass.
- The method in the subclass can call the superclass method using super.
- Overriding is resolved at runtime.

Example of Method Overriding:

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

class Dog extends Animal {
    // Overriding the sound method
```

```
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}

class Cat extends Animal {
    // Overriding the sound method
    @Override
    void sound() {
        System.out.println("Cat meows");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal a1 = new Dog();  // Polymorphism
        a1.sound();  // Outputs: Dog barks

        Animal a2 = new Cat();  // Polymorphism
        a2.sound();  // Outputs: Cat meows
    }
}
```

Output:

Copy code

Dog barks

Cat meows

In this example:

- Both Dog and Cat override the sound() method of the Animal class.
- The method that gets executed is determined at runtime (polymorphism).

Key Rules of Method Overriding:

- The **method signature** (name, return type, and parameters) in the subclass must be identical to the method in the superclass.
- The **access level** of the overridden method must be the same or more permissive than the method in the superclass (e.g., a private method cannot be overridden).
- The **final, static, and abstract** methods cannot be overridden.
- If a method is marked as **abstract** in the superclass, the subclass must either implement it (unless the subclass is abstract).

Abstract Classes in Java

An **abstract class** is a class that cannot be instantiated on its own and must be subclassed. It may contain both abstract methods (methods without implementation) and concrete methods (methods with implementation). Abstract classes are used to provide a base class with some common functionality that subclasses can share and extend.

Key Points about Abstract Classes:

- An **abstract method** is a method declared in an abstract class but does not have a body (i.e., it's not implemented).
- A **concrete method** can be implemented in the abstract class.
- An abstract class cannot be instantiated directly.
- A **subclass** of an abstract class must provide implementations for all the abstract methods, unless the subclass is also abstract.

Example of an Abstract Class:

java
Copy code

```
abstract class Animal {
    abstract void sound();  // Abstract method (no implementation)

    void eat() {  // Concrete method
        System.out.println("This animal eats food");
    }
}

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

class Cat extends Animal {
    @Override
    void sound() {
        System.out.println("Cat meows");
    }
}

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

```
        // Animal a = new Animal(); // Error: Cannot instantiate an abstract class

        Animal dog = new Dog();
        dog.sound();  // Outputs: Dog barks
        dog.eat();    // Outputs: This animal eats food

        Animal cat = new Cat();
        cat.sound();  // Outputs: Cat meows
        cat.eat();    // Outputs: This animal eats food
    }
}
```

Output:

Copy code
Dog barks
This animal eats food
Cat meows
This animal eats food

In this example:

- The Animal class is abstract and contains both an abstract method sound() and a concrete method eat().
- The Dog and Cat classes override the sound() method and provide their own implementations.
- We cannot create an instance of the Animal class directly, but we can instantiate Dog or Cat, which are subclasses.

Definition:

1. **Method Overloading:**
    - Methods with the same name but different parameter lists in the same class.
    - The compiler resolves which method to call based on the method signature at compile time.
2. **Method Overriding:**
    - A subclass provides a specific implementation of a method already defined in its superclass.
    - This enables runtime polymorphism, where the actual method invoked is determined at runtime.
3. **Abstract Classes:**
    - A class that cannot be instantiated directly and may contain abstract methods that must be implemented by subclasses.

- Used as a blueprint for other classes to extend and provide concrete implementations.

These concepts are fundamental for writing modular, reusable, and maintainable object-oriented programs in Java.

Dynamic Method Dispatch in Java

Dynamic Method Dispatch (also known as **runtime polymorphism**) is a mechanism in Java where a call to an overridden method is resolved at runtime rather than compile-time. This is possible due to method overriding, where a subclass provides a specific implementation of a method that is already defined in its superclass.

In dynamic method dispatch, the **object type** (not the reference type) determines which method gets called, making it a key feature of **polymorphism**.

How Dynamic Method Dispatch Works:

1. A method is overridden in the subclass.
2. The reference variable of the superclass points to an object of the subclass.
3. The method that gets called is based on the **object type** (the actual object, not the reference type), and it is determined at runtime.

Example of Dynamic Method Dispatch:

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

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

class Cat extends Animal {
    @Override
    void sound() {
        System.out.println("Cat meows");
    }
}

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

```
        Animal animal1 = new Dog();  // Animal reference but Dog object
        Animal animal2 = new Cat();  // Animal reference but Cat object

        animal1.sound();  // Outputs: Dog barks (determined at runtime)
        animal2.sound();  // Outputs: Cat meows (determined at runtime)
    }
}
```

Output:

Copy code
Dog barks
Cat meows

Explanation:

- In this example, the reference type is Animal, but the object types are Dog and Cat.
- Even though both animal1 and animal2 are declared as Animal, the actual method invoked is determined by the **object type** at runtime.
- This is the essence of **dynamic method dispatch**—the method resolution happens at runtime based on the actual object, not the reference type.

Usage of the final Keyword in Java

The final keyword in Java is used to define constants, prevent method overriding, and prevent inheritance. It can be applied to variables, methods, and classes.

1. final with Variables:

- A final variable can only be assigned once.
- Once a final variable has been assigned a value, it cannot be changed.

Example:

```
class Circle {
    final double PI = 3.14159;  // Constant value of PI

    double area(double radius) {
        return PI * radius * radius;  // PI cannot be changed
    }
}

public class Main {
    public static void main(String[] args) {
        Circle circle = new Circle();
        System.out.println("Area: " + circle.area(5));  // Area calculation using PI
    }
}
```

Explanation:

- The PI variable is declared as final, meaning it cannot be reassigned once it has been given a value.
- It helps ensure that the value of PI remains constant throughout the program.

2. final with Methods:

- A final method cannot be overridden by subclasses.
- This is used when we want to ensure that the behavior defined in the superclass method is preserved in all subclasses.

Example:

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

class Dog extends Animal {
    // This will cause a compile-time error because sound() is final
    // void sound() {
    //     System.out.println("Dog barks");
    // }
}

public class Main {
    public static void main(String[] args) {
        Animal animal = new Dog();
        animal.sound();  // Outputs: Animal makes a sound
    }
}
```

Explanation:

- The sound() method in the Animal class is marked as final, so it cannot be overridden in the Dog class.
- Trying to override the sound() method in the Dog class would result in a compile-time error.

3. final with Classes:

- A final class cannot be subclassed or inherited.
- This is useful when you want to prevent further inheritance of a class, ensuring that the class's behavior cannot be altered.

Example:

```
final class Animal {
    void sound() {
```

```
        System.out.println("Animal makes a sound");
    }
}

// This will cause a compile-time error because Animal is final
/class Dog extends Animal {
//     void sound() {
//         System.out.println("Dog barks");
//     }
// }

public class Main {
    public static void main(String[] args) {
        Animal animal = new Animal();
        animal.sound();  // Outputs: Animal makes a sound
    }
}
```

Explanation:

- The Animal class is marked as final, so it cannot be subclassed.
- Trying to create a Dog class that extends Animal would result in a compile-time error.

Key Points to Remember About final:

- **Final Variables:** A final variable can only be assigned once. After initialization, it cannot be changed.
- **Final Methods:** A final method cannot be overridden in subclasses.
- **Final Classes:** A final class cannot be subclassed (inherited).

Packages in Java

In Java, a **package** is a **namespace** that organizes a set of related classes, interfaces, and sub-packages. Packages help in managing the complexity of large systems by grouping related classes together. They provide a mechanism for encapsulation, preventing naming conflicts, and controlling access to classes and their members.

1. Definition of Packages in Java

A **package** in Java is a collection of classes, interfaces, and sub-packages that are grouped together. Packages are used to avoid class name conflicts and to provide access control.

- **Built-in Packages:** Java provides many built-in packages like java.util, java.io, java.lang, etc., that contain commonly used classes, interfaces, and functions.
- **User-defined Packages:** Developers can also create their own packages to organize classes based on their functionalities.

Types of Packages in Java:

1. **Built-in Packages:**
    - These packages are part of the Java standard library (Java API). Examples include java.util, java.io, and java.lang.
2. **User-defined Packages:**
    - Developers can define their own packages to group related classes. To create a user-defined package, we use the package keyword.

Example of a Package Declaration:

java
Copy code

```
package com.myapp.utils;  // Package declaration at the start of the class file

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

Here, the MathUtils class is part of the com.myapp.utils package.

---

2. Access Protection in Packages

Access control in Java is provided using **access modifiers**. These modifiers control the visibility of classes, methods, and variables across packages and help in encapsulating data and ensuring that only intended users can access certain members.

Access Modifiers in Java:

1. **public:**
    - The member is accessible from any other class in any package.
    - Classes, methods, or variables marked public are accessible to all other classes and packages.
2. **protected:**
    - The member is accessible within the same package and by subclasses (even if they are in different packages).
    - **Note**: protected access only applies to methods and fields, not top-level classes.
3. **default (no modifier):**
    - The member is **package-private**, meaning it is accessible only within the same package. No keyword is used for this access level.
4. **private:**
    - The member is accessible only within the same class.
    - It cannot be accessed from outside the class, even by subclasses.

Example of Access Protection:

```
// In com/myapp/utils/MathUtils.java
```

```
package com.myapp.utils;

public class MathUtils {
    public int add(int a, int b) {
        return a + b;  // Accessible anywhere
    }

    protected int subtract(int a, int b) {
        return a - b;  // Accessible within the same package or by subclass
    }

    int multiply(int a, int b) {
        return a * b;  // Accessible within the same package
    }

    private int divide(int a, int b) {
        return a / b;  // Accessible only within the MathUtils class
    }
}
```

In this example:

- The add() method is public, so it can be accessed anywhere.
- The subtract() method is protected, so it can be accessed within the same package or by subclasses.
- The multiply() method is **package-private** (no modifier), so it can be accessed only within the same package.
- The divide() method is private, so it can only be accessed within the MathUtils class.

---

3. Importing Packages in Java

To use classes from other packages, you need to **import** them. Java provides two types of importing mechanisms:

1. **Single Class Import:**
    - You can import a single class from a package using the import statement.

    Example:

    ```
    import com.myapp.utils.MathUtils;  // Importing a specific class from the package

    public class Main {
        public static void main(String[] args) {
            MathUtils math = new MathUtils();
    ```

```
        System.out.println(math.add(5, 3));
    }
}
```

2. **Wildcard Import:**
    - You can import all the classes from a package using the wildcard (*) symbol.

    Example:

```
import com.myapp.utils.*;  // Importing all classes from the com.myapp.utils package

public class Main {
    public static void main(String[] args) {
        MathUtils math = new MathUtils();
        System.out.println(math.add(5, 3));
    }
}
```

    Note: Wildcard imports are less efficient because they import all classes from the package, even if you are not using all of them.

Interfaces in Java

An **interface** in Java is a reference type, similar to a class, that is used to specify a set of abstract methods (methods without body) that a class must implement. It can also contain constants and default methods (methods with implementations). Interfaces allow us to achieve **abstraction** and **multiple inheritance** in Java.

1. Definition of an Interface

An interface in Java defines a contract for what a class can do, without specifying how it does it. It only contains method signatures (abstract methods), and the classes that implement the interface must provide the actual method implementations.

Key Points about Interfaces:

- An interface can only declare abstract methods (methods without body).
- A class implements an interface by providing concrete implementations for all its methods.
- Interfaces cannot contain instance fields. They can only contain constants (public, static, final variables).
- An interface can extend other interfaces.
- Java allows a class to implement multiple interfaces, making it a way to achieve multiple inheritance.

Syntax for Declaring an Interface:

```
interface InterfaceName {
    // abstract methods (no body)
    void method1();
```

```
    void method2();

    // constants
    int CONSTANT = 100;
}
```

---

2. Implementation of Interfaces

To implement an interface, a class must:

- Use the implements keyword.
- Provide concrete implementations for all the abstract methods declared in the interface.

Example of Implementing an Interface:

```
// Define an interface
interface Animal {
    void sound();  // Abstract method
    void eat();    // Abstract method
}

// Implement the interface in a class
class Dog implements Animal {
    // Providing implementation of sound()
    @Override
    public void sound() {
        System.out.println("Dog barks");
    }

    // Providing implementation of eat()
    @Override
    public void eat() {
        System.out.println("Dog eats meat");
    }
}

class Main {
    public static void main(String[] args) {
        Animal animal = new Dog();  // Creating an object of the Dog class
        animal.sound();  // Outputs: Dog barks
        animal.eat();    // Outputs: Dog eats meat
    }
}
```

Explanation:

- The Animal interface defines two methods: sound() and eat().
- The Dog class implements the Animal interface and provides concrete implementations for both methods.
- In the Main class, we create a Dog object and use it via the Animal reference.

---

3. Extending Interfaces

An interface in Java can extend another interface, allowing a subclass to inherit multiple method declarations from more than one interface. This is a way of achieving **multiple inheritance** of method signatures, unlike classes, which can only inherit from one class.

Syntax for Extending an Interface:

```
interface InterfaceA {
   void methodA();
}

interface InterfaceB {
   void methodB();
}

// A new interface can extend multiple interfaces
interface CombinedInterface extends InterfaceA, InterfaceB {
   void methodC();
}
```

In this example:

- CombinedInterface extends both InterfaceA and InterfaceB, so it inherits all the methods from both interfaces and can declare additional methods like methodC().

Example of Implementing an Interface that Extends Other Interfaces:

```
// Define the first interface
interface Animal {
   void sound();
}

// Define the second interface
interface Mammal {
   void eat();
}

// Define a third interface that extends Animal and Mammal
```

```
interface Dog extends Animal, Mammal {
    void sleep();
}

// Implement the Dog interface
class Bulldog implements Dog {
    @Override
    public void sound() {
        System.out.println("Bulldog barks");
    }

    @Override
    public void eat() {
        System.out.println("Bulldog eats bones");
    }

    @Override
    public void sleep() {
        System.out.println("Bulldog sleeps in the kennel");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Bulldog();
        dog.sound();  // Outputs: Bulldog barks
        dog.eat();    // Outputs: Bulldog eats bones
        dog.sleep();  // Outputs: Bulldog sleeps in the kennel
    }
}
```

Explanation:

- The Dog interface extends both the Animal and Mammal interfaces, inheriting the methods sound() and eat().
- The Bulldog class implements the Dog interface and provides concrete implementations for all methods.
- The Bulldog class is now forced to implement the methods from both Animal and Mammal, as well as the additional method sleep() declared in the Dog interface.

Exception Handling in Java

Exception Handling in Java is a powerful mechanism that allows programs to handle runtime errors, maintaining the normal flow of application execution. It helps in managing unexpected conditions that may occur during program execution, such as input/output errors, network failures, or incorrect data processing.

In Java, exceptions are objects that represent an error or an exceptional condition during the execution of a program. Exception handling in Java is accomplished using the try, catch, throw, throws, and finally keywords.

1. Keywords in Exception Handling

try-catch Block:

- The try block contains code that might throw an exception.
- The catch block catches the exception and handles it. It specifies the type of exception it will handle.
- If an exception occurs in the try block, the control is transferred to the appropriate catch block.

Syntax:

```
try {
   // Code that may throw an exception
} catch (ExceptionType e) {
   // Handling the exception
}
```

Example of try-catch:

```
public class Main {
   public static void main(String[] args) {
      try {
         int[] arr = new int[5];
         arr[10] = 50;  // This will throw an ArrayIndexOutOfBoundsException
      } catch (ArrayIndexOutOfBoundsException e) {
         System.out.println("Exception caught: " + e);
      }
   }
}
```

Output:

Exception caught: java.lang.ArrayIndexOutOfBoundsException: Index 10 out of bounds for length 5

In this example:

- The code inside the try block throws an ArrayIndexOutOfBoundsException because we are trying to access an invalid index of the array.
- The catch block catches the exception and prints a message with the exception details.

throw:

- The throw keyword is used to explicitly throw an exception from a method or block of code.
- It is followed by an instance of an exception class.

Syntax:

java
Copy code

```
throw new ExceptionType("Error message");
```

Example of throw:

```
public class Main {
   public static void main(String[] args) {
      try {
         throw new ArithmeticException("Custom Arithmetic Exception");
      } catch (ArithmeticException e) {
         System.out.println("Exception caught: " + e.getMessage());
      }
   }
}
```

Output:

Exception caught: Custom Arithmetic Exception

In this example:

- We use throw to explicitly throw an ArithmeticException with a custom error message.
- The catch block handles this exception.

throws:

- The throws keyword is used in a method declaration to specify that the method might throw certain exceptions.
- It is used when a method calls other methods that may throw exceptions and does not handle them internally.

Syntax:

```
public void method() throws ExceptionType {
   // Code that might throw an exception
}
```

Example of throws:

```
public class Main {
   public static void main(String[] args) {
      try {
         testMethod();  // Calling method that throws an exception
      } catch (IOException e) {
         System.out.println("Exception caught: " + e.getMessage());
```

```
        }
    }

    // Method that throws an exception
    public static void testMethod() throws IOException {
        throw new IOException("Custom I/O Exception");
    }
}
```

Output:

mathematica

Copy code

```
Exception caught: Custom I/O Exception
```

In this example:

- The testMethod() declares that it throws an IOException.
- In the main method, we handle this exception using a try-catch block.

finally:

- The finally block is used to execute code that must always run, regardless of whether an exception was thrown or not.
- It is typically used to close resources like files, network connections, or database connections.

Syntax:

```
try {
    // Code that may throw an exception
} catch (ExceptionType e) {
    // Handling the exception
} finally {
    // Code that will always execute
}
```

Example of finally:

```
public class Main {
    public static void main(String[] args) {
        try {
            int result = 10 / 0;  // This will throw ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Exception caught: " + e.getMessage());
        } finally {
            System.out.println("This will always execute");
        }
    }
}
```

Output:
Exception caught: / by zero
This will always execute
In this example:

- The try block throws an ArithmeticException.
- The catch block handles the exception.
- The finally block always executes, regardless of whether an exception occurred or not.

---

2. Built-in Exceptions in Java
Java provides a rich set of built-in exceptions to handle various types of errors. These exceptions are subclasses of the Throwable class, which has two main categories:

- **Error:** Represents severe errors (e.g., OutOfMemoryError).
- **Exception:** Represents conditions that a program can catch and handle (e.g., IOException, SQLException, NullPointerException).

Some common **built-in exceptions** include:

1. **ArithmeticException:** Thrown when an exceptional arithmetic condition occurs, like division by zero.
2. **ArrayIndexOutOfBoundsException:** Thrown when trying to access an invalid index of an array.
3. **NullPointerException:** Thrown when trying to use a reference that points to null.
4. **IOException:** Thrown when there is an issue with input/output operations.
5. **FileNotFoundException:** Thrown when a file is not found during I/O operations.
6. **ClassNotFoundException:** Thrown when a class is not found during runtime.

Example of Built-in Exception:

```
public class Main {
    public static void main(String[] args) {
        try {
            int[] arr = new int[3];
            arr[5] = 10;  // This will throw ArrayIndexOutOfBoundsException
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index out of bounds: " + e.getMessage());
        }
    }
}
```

---

3. Creating Custom Exception Classes

- In the main method, we catch and handle the custom exception.

Note:

1. **try-catch Block:**
   - Used to catch and handle exceptions. The code that may throw an exception is placed inside the try block, and the exception is caught and handled inside the catch block.
2. **throw:**
   - Used to explicitly throw an exception.
3. **throws:**
   - Used in method declarations to specify that the method may throw exceptions, passing the responsibility of handling the exception to the calling method.
4. **finally:**
   - Code in the finally block always executes, regardless of whether an exception is thrown or not. It is typically used to clean up resources.
5. **Built-in Exceptions:**
   - Java provides many built-in exceptions (e.g., ArithmeticException, IOException, ArrayIndexOutOfBoundsException), which can be caught and handled.
6. **Creating Custom Exceptions:**
   - You can create your own exceptions by extending the Exception class or its subclasses. Custom exceptions are useful for handling specific error conditions in your application.

Exception handling is an essential part of robust and reliable Java applications, allowing developers to manage unexpected conditions gracefully and improve user experience.