THE COMPLETE CORE JAVA COURSE

PRESENTED BY,

KGM TECHNICAL TEAM

SLIDES FOR THEORY LECTURES

KGiSL MicrCollege

Genius as a Service (GaaS)

THE COMPLETE CORE JAVA COURSE

PRESENTED BY,

KGM TECHNICAL TEAM

DAY-04 & 05

Welcome Section

LECTURE

Watch before you Start

Inheritance

Definition:

- Inheritance is a mechanism in Java where one class (child/subclass) acquires the properties (fields) and behaviors (methods) of another class (parent/superclass).
- It promotes code reusability and a hierarchical class structure.

Key Features:

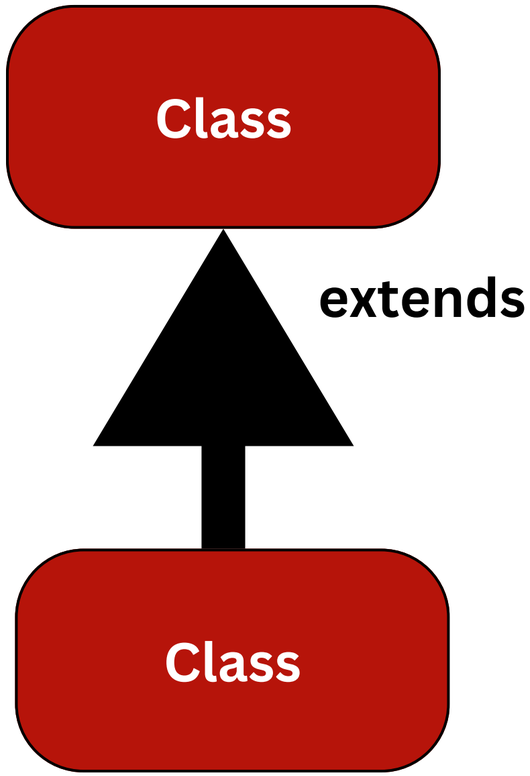
- **Super Class (Parent):** The class whose properties are inherited.
- **Sub Class (Child):** The class that inherits properties and methods.
- **extends Keyword:** Used to establish inheritance.
- **Reusability:** Common functionality is written once in the superclass and reused in subclasses.

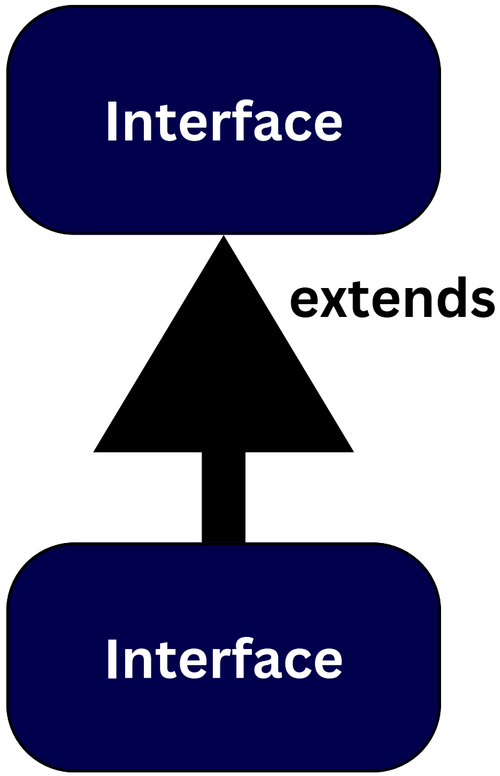
Inheritance

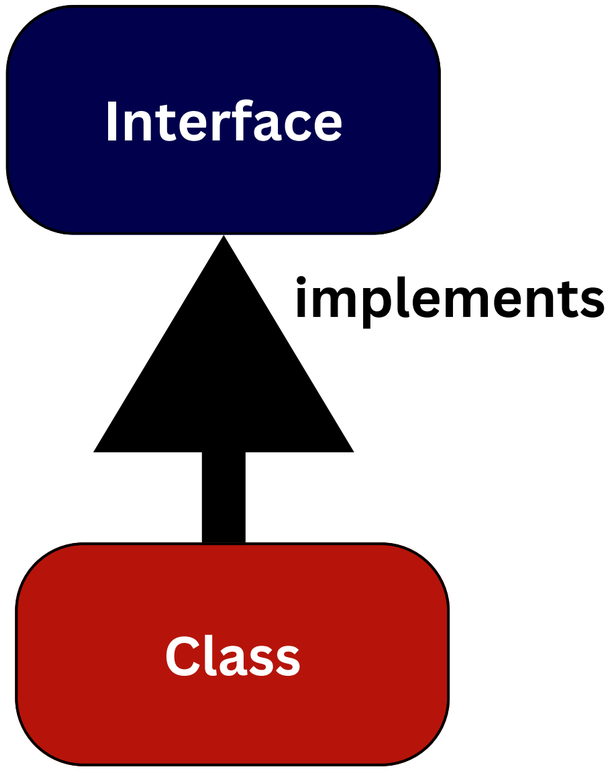
Types of Inheritance:

- Single Inheritance
- Multi-level Inheritance
- Hierarchical Inheritance
- Multiple Inheritance (achieved through interface)
- Hybrid Inheritance (achieved through interface)

KEYWORDS TO BE USED TO CALL CLASS AND INTERFACE







Single Inheritance

Definition:

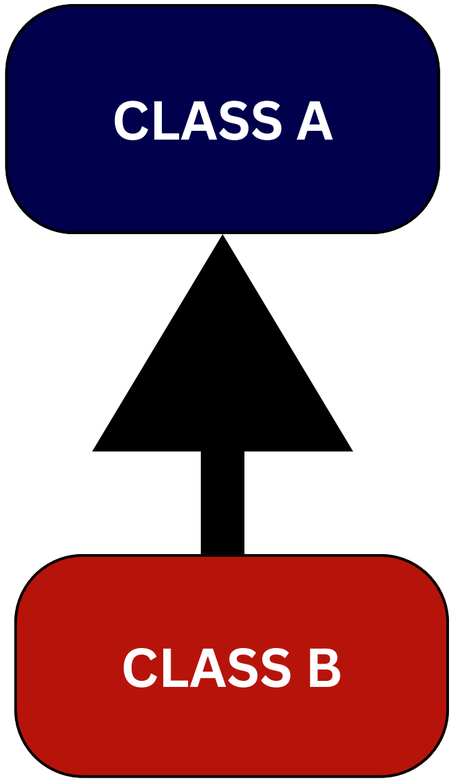
- A subclass inherits from a single parent class.
- The child class acquires the properties and methods of the parent class.

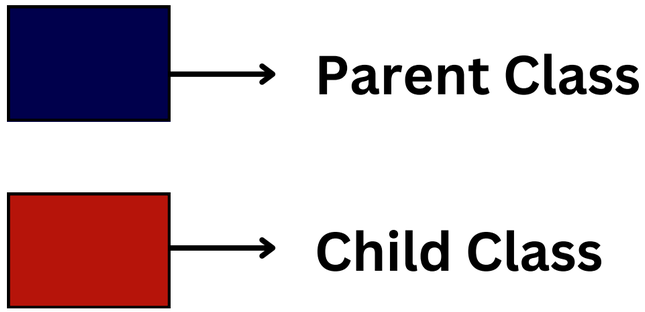
Syntax:

```
class Parent {
  // Parent class fields and methods
}

class Child extends Parent {
  // Child class fields and methods
}
```

Single Inheritance





Example:

```
// Parent class
class Animal {
  void eat() {
    System.out.println("This animal eats food.");
  }
}

// Child class
class Dog extends Animal {
  void bark() {
    System.out.println("The dog barks.");
  }
}
```

Example:

```
public class Main {
  public static void main(String[] args) {
    Dog dog = new Dog();
    dog.eat(); // Inherited method
    dog.bark(); // Child class method
  }
}
```

Output:

This animal eats food.

The dog barks.

Multi-Level Inheritance

Definition:

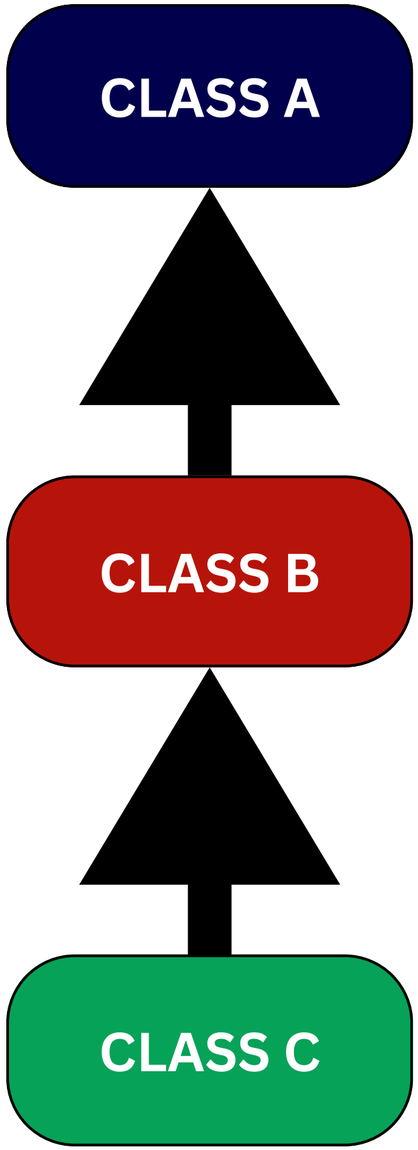
- A subclass inherits from another subclass, forming a multi-level hierarchy.
- This means a class can inherit from a parent class, and then another class can inherit from that subclass.

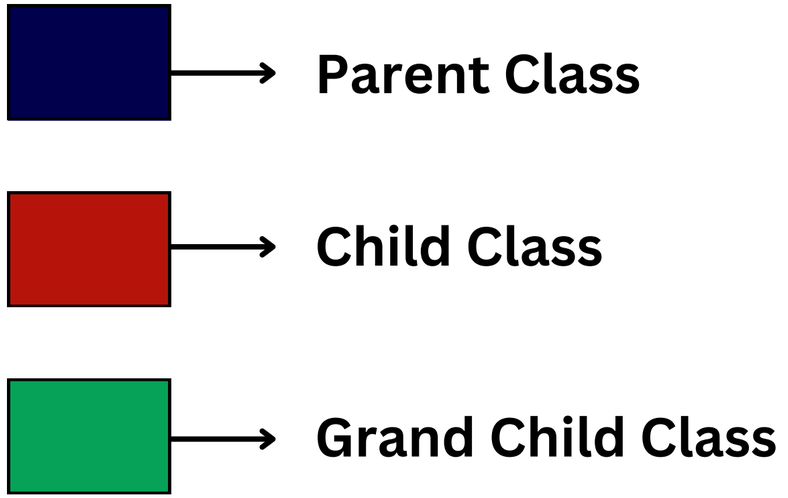
Syntax:

```
class Parent {
  // Parent class fields and methods
}

class Child extends Parent {
  // Child class fields and methods
}
```

Multi-Level Inheritance





Example:

```
// Parent class
class Vehicle {
  void move() {
    System.out.println("This vehicle moves.");
  }
}

// Child class
class Car extends Vehicle {
  void drive() {
    System.out.println("This car drives.");
  }
}
```

Example:

```
// Grandchild class
class ElectricCar extends Car {
  void charge() {
    System.out.println("This car charges its battery.");
  }
}

public class Main {
  public static void main(String[] args) {
    ElectricCar tesla = new ElectricCar();
    tesla.move();  // Method from Vehicle class
    tesla.drive(); // Method from Car class
    tesla.charge(); // Method from ElectricCar class
  }
}
```

Example:

Output:

This vehicle moves.

This car drives.

This car charges its battery.

Key Benefits of Inheritance:

- **Code Reusability:** Avoids duplicating code across related classes.
- **Ease of Maintenance:** Changes in the superclass automatically reflect in subclasses.
- **Extensibility:** Allows adding specific functionality in subclasses without affecting the superclass.

Method Overriding

Definition:

- Method Overriding is a feature in Java where a subclass provides a specific implementation of a method that is already defined in its superclass.
- The overridden method must have the same name, return type, and parameters.

Key Features:

- **Runtime Polymorphism:** Determined at runtime based on the object's type.
- **Annotation:** Use **@Override** to explicitly indicate overriding.
- **Dynamic Behavior:** Customizes or enhances the behavior of inherited methods.

Method Overriding

Rules of Overriding:

- The method must have the same name, return type, and parameters as the superclass method.
- The method in the subclass cannot have stricter access than the method in the superclass.
- Only inherited methods can be overridden.

Example: Method Overriding

1. Basic Overriding Example:

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

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

Example: Method Overriding

```
public class Main {
  public static void main(String[] args) {
    Animal myAnimal = new Dog(); // Polymorphism
    myAnimal.sound(); // Calls overridden method in Dog class
  }
}
```

Output:

Dogs bark.

Example: Method Overriding

The **super** keyword is used to call the superclass's version of the overridden method.

2. Overriding with super Keyword:

```
class Parent {
  void show() {
    System.out.println("This is the parent class method.");
  }
}

class Child extends Parent {
  @Override
  void show() {
    super.show(); // Call parent class method
    System.out.println("This is the child class method.");
  }
}
```

Example: Method Overriding

```
public class Main {
  public static void main(String[] args) {
    Child child = new Child();
    child.show();
  }
}
```

Output:

```
This is the parent class method.
This is the child class method.
```

Hierarchical Inheritance

Definition:

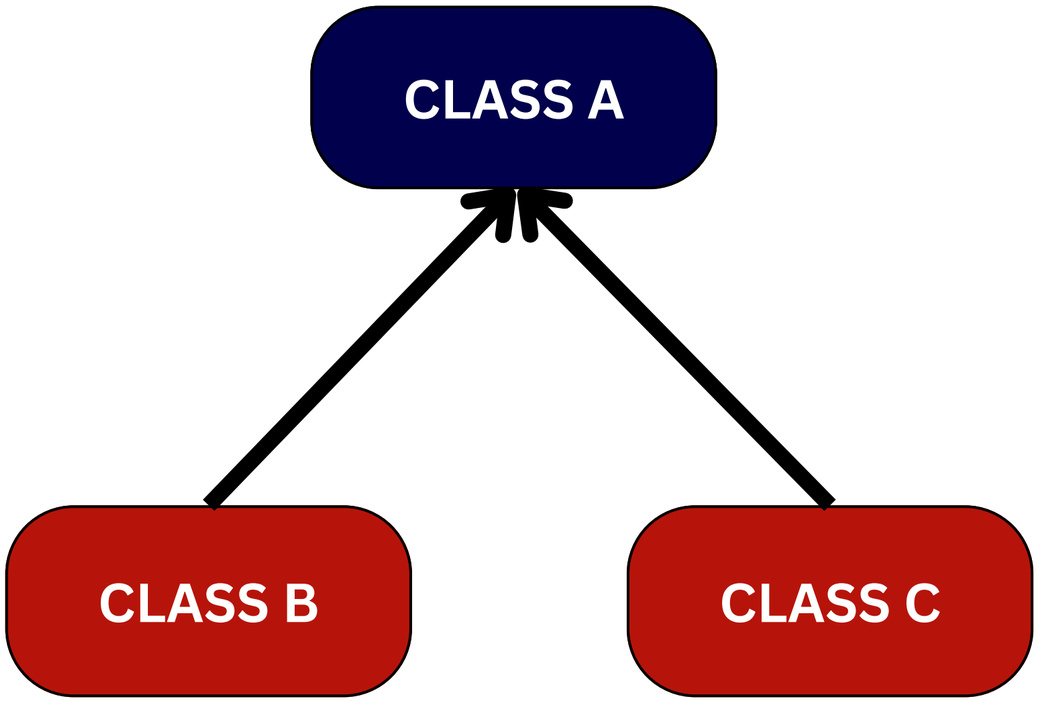
- Multiple classes inherit from the same parent class.
- The child classes are distinct, but they share common behavior or properties from the parent.

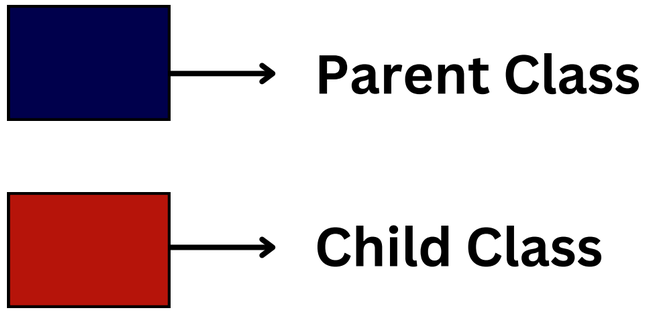
Syntax:

```
class Parent {
  // Parent class fields and methods
}

class Child extends Parent {
  // Child class fields and methods
}
```

Hierarchical Inheritance





Advanced Example: Overriding in Hierarchical Inheritance

Example:

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

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

Advanced Example: Overriding in Hierarchical Inheritance

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

public class Main {
  public static void main(String[] args) {
    Animal animal1 = new Dog();
    Animal animal2 = new Cat();

    animal1.sound(); // Calls Dog's version
    animal2.sound(); // Calls Cat's version
  }
}
```

Output:

Output:

Dogs bark.

Cats meow.

Combined Example: Inheritance with Method Overriding

Example:

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

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

Combined Example: Inheritance with Method Overriding

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

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

    shape1.draw(); // Calls Circle's version
    shape2.draw(); // Calls Rectangle's version
  }
}
```

Output:

Output:

Drawing a circle.

Drawing a rectangle.

Polymorphism

Definition:

- Polymorphism in Java refers to the ability of a single action to behave differently based on the context.
- It enables objects of different classes to be treated as objects of a common superclass, primarily achieved through method overriding (runtime polymorphism) and method overloading (compile-time polymorphism).

Types of Polymorphism:

1. **Compile-time Polymorphism (Static):**
   - Achieved through method overloading.
   - Resolved during compilation.
2. **Runtime Polymorphism (Dynamic):**
   - Achieved through method overriding.
   - Resolved at runtime based on the actual object type.

Overriding vs. Overloading:

Aspect	Overriding	Overloading
Definition	Redefining a method in a subclass.	Defining methods with the same name but different parameters.
Inheritance Required	Yes	No
Polymorphism	Runtime Polymorphism	Compile-time Polymorphism
Parameters	Must be the same as the superclass method.	Must differ in number, type, or order.

Compile-time Polymorphism (Method Overloading)

Definition:

- This type of polymorphism occurs when multiple methods in the same class have the same name but different parameter lists (method signatures).
- The method to be invoked is determined at compile time.

Key Features:

- Methods have the same name but differ in the number or type of parameters.
- Return type may or may not be the same.
- Achieved through **method overloading.**

---

2 ways to overload method in java:

- By changing number of arguments
- By changing the datatypes

Example: By changing number of arguments

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

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

Example: By changing number of arguments

```
public class Main {
  public static void main(String[] args) {
    Calculator calc = new Calculator();
    System.out.println("Sum of 2 numbers: " + calc.add(5, 10));
    System.out.println("Sum of 3 numbers: " + calc.add(5, 10, 15));
  }
}
```

Output:

```
Sum of 2 numbers: 15
Sum of 3 numbers: 30
```

Example: By changing the datatypes

```
import java.util.Scanner;

public class OverLoadingExample {
   int c; //Global variable

 public void add(int a, int b){
      c = a + b;
      System.out.println("The value of integer c is: "+c);
  }

  public void add(float a, int b){
      float c = a + b;
      System.out.println("The value of float c is: "+c);
  }
```

Example: By changing the datatypes

```
public void add(float a, double b){
   double c = a + b;
   System.out.println("The value of double c is: "+c);
}
public static void main(String[] args){
   OverLoadingExample ole = new OverLoadingExample();
   Scanner sc = new Scanner(System.in);

   int a, b;
   System.out.println("Enter the value for a and b:");
   a = sc.nextInt();
   b = sc.nextInt();
   ole.add(a,b);
```

Example: By changing the datatypes

```
        float c; int d;
        System.out.println("Enter the value for c and d:");
        c = sc.nextFloat();
        d = sc.nextInt();
        ole.add(c,d);

        float e; double f;
        System.out.println("Enter the value for e and f:");
        e = sc.nextFloat();
        f = sc.nextDouble();
        ole.add(e,f);

    }
}
```

Run-time Polymorphism (Method Overriding)

Definition:

- This type of polymorphism occurs when a subclass provides a specific implementation of a method that is already defined in its parent class.
- The method that gets invoked is determined at runtime based on the object type.

Key Features:

- Requires inheritance (one class extends another).
- The method in the child class must have the same name, return type, and parameter list as the method in the parent class.
- Achieved through method overriding.
- It is a form of dynamic dispatch.

Example:

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

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

Example:

```
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(); // Polymorphic behavior
    Animal animal2 = new Cat();

    animal1.sound(); // Calls Dog's implementation
    animal2.sound(); // Calls Cat's implementation
  }
}
```

Example:

Output:

Dog barks.

Cat meows.

Interfaces

Definition:

- An interface in Java is a blueprint of a class that contains abstract methods (methods without implementation) and static constants.
- Classes implement interfaces to provide specific functionality, enabling multiple inheritance.

Key Features of Interfaces:

- Defined using the interface keyword.
- All methods in an interface are public and abstract by default.
- Fields in an interface are public, static, and final.
- A class can implement multiple interfaces.
- Default and static methods can also be included (Java 8 onwards).

Example: Using interfaces

Example:

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

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

Example: Using interfaces

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

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

Example: Using interfaces

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

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

Example: Using interfaces

```
    @Override
    public void eat() {
        System.out.println("Cat eats fish.");
    }
}
```

Example: Using interfaces

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

 dog.sound();
 dog.eat();

 cat.sound();
 cat.eat();
 }
}
```

Output for Example:

Output:

Dog barks.

Dog eats bones.

Cat meows.

Cat eats fish.

Advanced Example: Multiple Interfaces

Example:

```
interface Printable {
  void print();
}

interface Showable {
  void show();
}
```

Advanced Example: Multiple Interfaces

```
class Document implements Printable, Showable {
  @Override
  public void print() {
    System.out.println("Printing document...");
  }

  @Override
  public void show() {
    System.out.println("Showing document.");
  }
}
```

Advanced Example: Multiple Interfaces

```
public class Main {
  public static void main(String[] args) {
    Document doc = new Document();
    doc.print();
    doc.show();
  }
}
```

Output:

```
Printing document...
Showing document.
```

Abstraction

Definition:

- Abstraction is the process of hiding implementation details and showing only the essential features of an object.

Key Features of Abstraction:

- Implemented using **abstract classes** and **interfaces** in Java.
- Helps focus on "what an object does" rather than "how it does it."

Abstract Class Example:

Example:

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

  // Regular method
  void sleep() {
System.out.println("Sleeping...");
  }
}
```

Abstract Class Example:

```
// Subclass providing implementation
class Dog extends Animal {
  void sound() {
    System.out.println("Dog barks!");
  }
}

public class Main {
  public static void main(String[] args) {
    Animal dog = new Dog();
    dog.sound(); // Calls overridden method
    dog.sleep(); // Calls inherited method
  }
}
```

Abstract Class Output:

Output:

Dog barks!

Sleeping...

Interface Example:

Example:

```
// Define an interface
interface Vehicle {
  void start(); // Abstract method
  void stop(); // Abstract method
}

// Implement the interface
class Car implements Vehicle {
  public void start() {
     System.out.println("Car is starting...");
  }
```

Interface Example:

```
  public void stop() {
    System.out.println("Car is stopping...");
  }
}

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

Interface Output:

Output:

Car is starting...

Car is stopping...

Encapsulation vs. Abstraction:

Aspect	Encapsulation	Abstraction
Focus	Protecting data by restricting direct access.	Hiding implementation details.
Implementation	Achieved using private fields and public methods.	Achieved using abstract classes or interfaces.
UseCase	Ensures controlled access to sensitive data.	Provides a clear separation between behavior and implementation.

Abstract Classes

Definition:

- An abstract class is a class that cannot be instantiated and is declared using the abstract keyword.
- It can have both abstract methods (without implementation) and concrete methods (with implementation).
- Subclasses must provide implementations for abstract methods.

Key Features of Abstract Classes:

- Used when there is a need for shared behavior (via concrete methods) and enforced behavior (via abstract methods).
- Can have instance variables and constructors.
- 3. Subclasses use the extends keyword to inherit.

Example: Abstract Class and Abstract Method

Example:

```
abstract class Animal {
  abstract void sound(); // Abstract method

  void sleep() {
    System.out.println("This animal sleeps.");
  }
}

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

Example: Abstract Class and Abstract Method

Example:

```
abstract class Animal {
  abstract void sound(); // Abstract method

  void sleep() {
    System.out.println("This animal sleeps.");
  }
}

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

Example: Abstract Class and Abstract Method

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

Example: Abstract Class and Abstract Method

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

 dog.sound();
 dog.sleep();

 cat.sound();
 cat.sleep();
 }
}
```

Example: Abstract Class and Abstract Method

Output:

```
This animal sleeps.
Cat meows.
This animal sleeps.
```

Comparison: Interface vs Abstract Class

Feature	Interface	Abstract Class
Keyword	interface	abstract
Methods	All methods are abstract (except default or static methods).	Can have both abstract and concrete methods.
Variables	public static final by default.	Can have instance variables.
Inheritance	A class can implement multiple interfaces.	A class can extend only one abstract class.
Constructor	Not allowed	Allowed

Advanced Example: Abstract Class with Concrete Methods

Example:

```
abstract class Shape {
  abstract void draw(); // Abstract method

  void color(String color) { // Concrete method
    System.out.println("Coloring the shape: " + color);
  }
}

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

Advanced Example: Abstract Class with Concrete Methods

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

Advanced Example: Abstract Class with Concrete Methods

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

    circle.draw();
    circle.color("Red");

    rectangle.draw();
    rectangle.color("Blue");
  }
}
```

Advanced Example: Abstract Class with Concrete Methods

Output:

```
Drawing a circle.
Coloring the shape: Red
Drawing a rectangle.
Coloring the shape: Blue
```

Multiple Inheritance

Definition:

- In Java, multiple inheritance occurs when a class or object inherits features from more than one parent class.
- However, Java does not support multiple inheritance with classes directly to avoid ambiguity (the diamond problem).
- Instead, it supports multiple inheritance using interfaces.

Key Points:

- **Using Interfaces:** A class can implement multiple interfaces, achieving multiple inheritance.
- **Avoiding Ambiguity:** Methods with the same name in multiple interfaces must be explicitly handled by the implementing class.

Multiple Inheritance

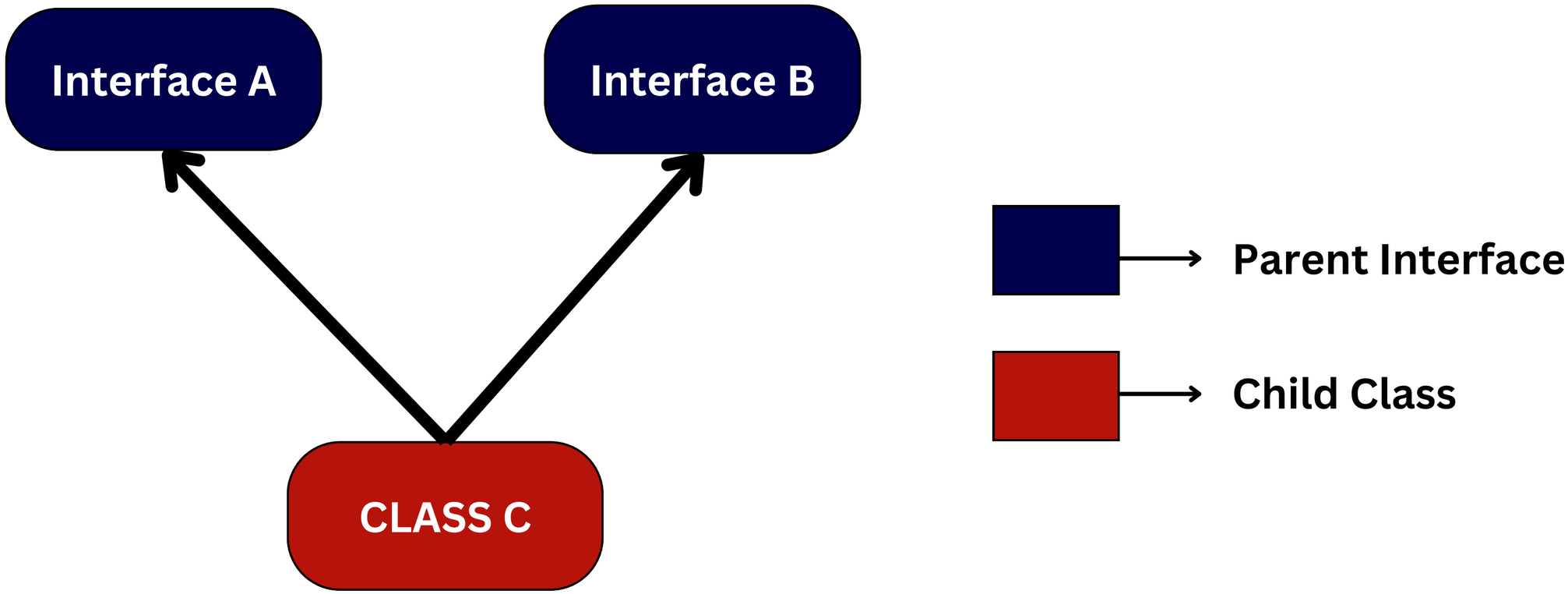
Syntax:

```
interface Parent1 {
  // Parent1 interface methods
}

interface Parent2 {
 // Parent1 interface methods
}

class Child implements Parent1, Parent2 {
  // Child class fields and methods
}
```

Multiple Inheritance



Example:

Example:

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

interface B {
  void methodB();
}
```

Example:

```
// Class implementing both interfaces
class C implements A, B {
  public void methodA() {
    System.out.println("Method A from Interface A");
  }

  public void methodB() {
    System.out.println("Method B from Interface B");
  }
}
```

Example:

```
public class Main {
  public static void main(String[] args) {
    C obj = new C();
    obj.methodA(); // Calls methodA
    obj.methodB(); // Calls methodB
  }
}
```

Output:

```
Method A from Interface A
Method B from Interface B
```

Diamond Problem with Interfaces Example:

Example:

```
interface A {
  default void display() {
    System.out.println("Display from Interface A");
  }
}

interface B {
  default void display() {
    System.out.println("Display from Interface B");
  }
}
```

Diamond Problem with Interfaces Example:

```
// Class resolving ambiguity
class C implements A, B {
  public void display() {
     A.super.display(); // Explicitly calling Interface A's display
     B.super.display(); // Explicitly calling Interface B's display
  }
}
```

Diamond Problem with Interfaces Example:

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

Output:

Display from Interface A

Display from Interface B

Hybrid Inheritance

Definition:

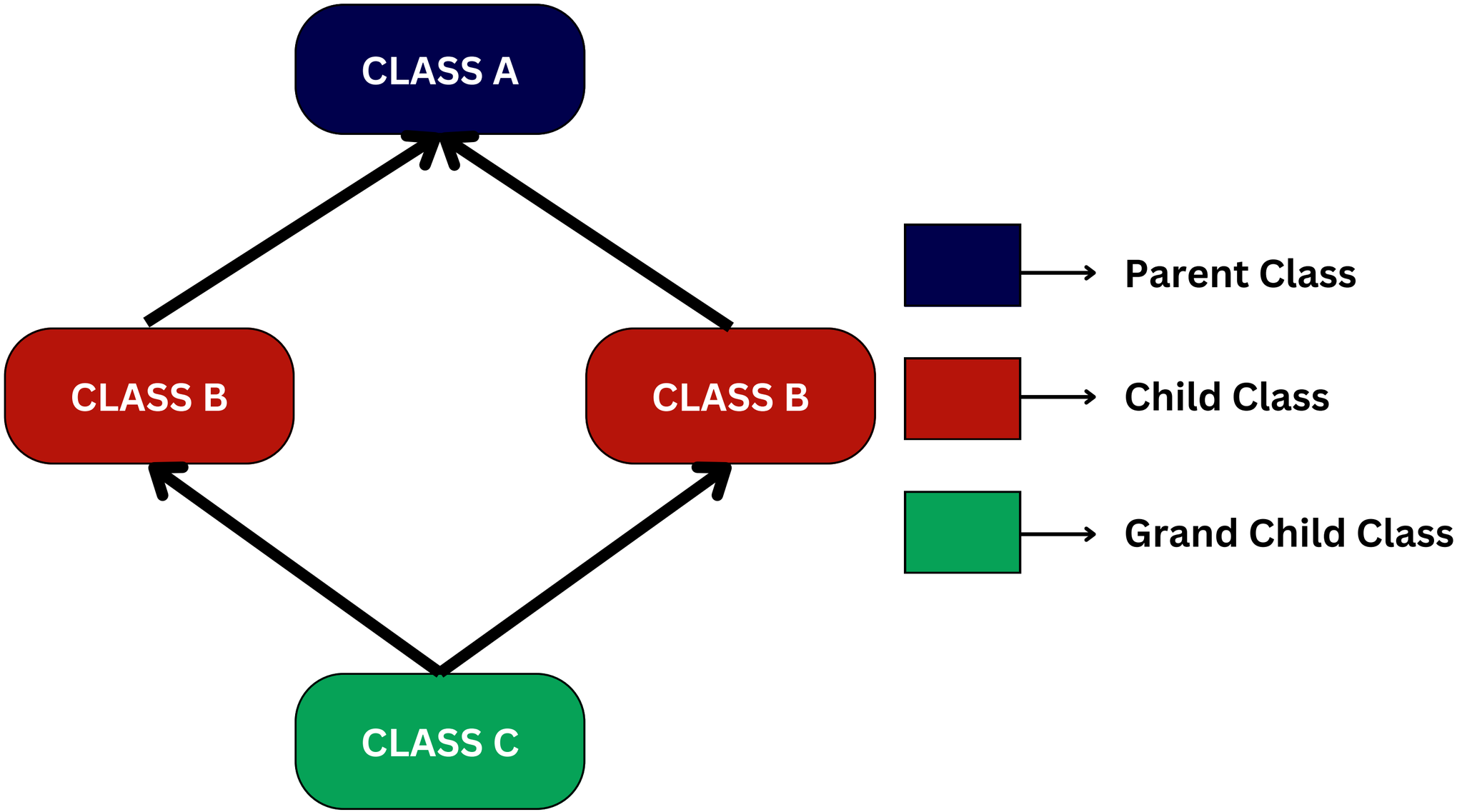
- Hybrid Inheritance is a combination of two or more types of inheritance, such as **single, hierarchical, multilevel, and multiple inheritance**.
- Java does not support **multiple inheritance with classes** to avoid ambiguity (diamond problem).
- However, hybrid inheritance can be achieved in Java using a combination of **classes and interfaces.**

Hybrid Inheritance

Key Points of Hybrid Inheritance in Java:

- Java does not support multiple inheritance with classes but allows it using interfaces.
- Hybrid inheritance is a combination of different types of inheritance.
- It avoids the diamond problem by using interfaces.
- The extends keyword is used for class inheritance, and implements is used for interface inheritance.
- Hybrid inheritance is commonly used in real-world applications where different types of relationships exist between classes and interfaces.

Hybrid Inheritance



Example:

Example:

```
// Parent class
class Parent {
  void parentMethod() {
    System.out.println("Parent class method");
  }
}

// First child class inherits from Parent
class Child1 extends Parent {
  void child1Method() {
    System.out.println("Child1 class method");
  }
}
```

Example:

```
// Second child class inherits from Parent
class Child2 extends Parent {
  void child2Method() {
     System.out.println("Child2 class method");
  }
}

// Interface (to support multiple inheritance)
interface ExtraFeature {
  void extraMethod();
}
```

Example:

```
// Hybrid inheritance: This class inherits from Child2 and implements ExtraFeature

class GrandChild extends Child2 implements ExtraFeature {
  public void extraMethod() {
     System.out.println("Interface method implemented in GrandChild");
  }

  void grandChildMethod() {
     System.out.println("GrandChild class method");
  }
}
```

Example:

```
// Main class
public class HybridInheritanceExample {
  public static void main(String[] args) {
    GrandChild obj = new GrandChild();

    obj.parentMethod();   // From Parent class
    obj.child2Method();   // From Child2 class
    obj.extraMethod();    // From Interface
    obj.grandChildMethod();// From GrandChild class
  }
}
```

Output:

Output:

Parent class method
Child2 class method
Interface method implemented in GrandChild
GrandChild class method

Composition and Aggregation in Java

Definition:

- Both Composition and Aggregation are forms of Association in Java, representing relationships between classes.
    - Composition: A strong relationship where one class cannot exist without the other (tight coupling).
    - Aggregation: A weaker relationship where one class can exist independently of the other (loose coupling).

Composition and Aggregation in Java

1. Composition (Strong Association)

- **"Has-a" relationship** with **strong dependency**.
- If the container (parent) is destroyed, the contained object (child) is also destroyed.
- Used when the lifecycle of the contained object **depends** on the lifecycle of the container.

Example:

Example:

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

class Car {
   private Engine engine; // Composition (Car cannot exist without Engine)
```

Example:

```
    public Car() {
        engine = new Engine(); // Engine is created inside Car
    }

    void drive() {
        engine.start();
        System.out.println("Car is driving...");
    }
}
```

Example:

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

Output:

```
Output:
Engine started...
Car is driving...
```

Key Point:

- Car **owns** Engine, and Engine cannot exist without Car.

Composition and Aggregation in Java

2. Aggregation (Weak Association)

- **"Has-a" relationship** but with **weak dependency**.
- The contained object **can exist independently** of the container.
- Used when objects should be **loosely coupled**.

Key Point:

- Person **has an** Address, but Address **can exist independently**.

Example:

Example:

```
class Address {
   String city, state;

   Address(String city, String state) {
      this.city = city;
      this.state = state;
   }

   void showAddress() {
      System.out.println(city + ", " + state);
   }
}
```

Example:

```
class Person {
    String name;
    Address address; // Aggregation (Person has an Address)

    Person(String name, Address address) {
        this.name = name;
        this.address = address; // Address is passed as an argument (not created inside)
    }
void display() {
        System.out.println("Name: " + name);
        address.showAddress();
    }
}
```

Example & Output:

```
public class AggregationExample {
    public static void main(String[] args) {
        Address addr = new Address("New York", "USA");
        Person p1 = new Person("John", addr);

        p1.display();
    }
}
```

Output:

```
Name: John
New York, USA
```

Comparison: Composition vs Aggregation

Feature	Composition	Aggregation
Dependency	Strong (Tight Coupling)	Weak (Loose Coupling)
Lifecycle	Child object cannot exist without the parent	Child object can exist independently
Relationship	"Part-of" relationship	"Has-a" relationship
Object Creation	Contained object is created inside the container class	Contained object is passed as a reference

Composition and Aggregation in Java

Summary

- **Use Composition** when objects **cannot exist separately** (e.g., Engine in Car).
- **Use Aggregation** when objects **can exist independently** (e.g., Address in Person).

Inner Classes and Anonymous Classes in Java

- Java allows defining **classes within classes**, known as **inner classes**, which help in logically grouping related functionality.
- Among them, **anonymous classes** are a special type of inner class that **does not have a name** and is used for short-lived operations.

1. Inner Classes in Java

- An **inner class** is a class that is **declared inside another class.** It provides **better encapsulation and logical grouping** of code.

Inner Classes and Anonymous Classes in Java

Types of Inner Classes

- Member Inner Class (Non-static inner class)
- Static Nested Class (Static inner class)
- Local Inner Class (Inside a method)
- Anonymous Inner Class (Inner class without a name)

1.1 Member Inner Class (Non-Static Inner Class)

- Declared inside another class and not static.
- It can access all members of the outer class.

Example:

Example:

```
class Outer {
   private String message = "Hello from Outer class";

   class Inner {
      void show() {
         System.out.println(message); // Accessing private member of Outer
      }
   }
}
```

Example:

```
public class MemberInnerClassExample {
    public static void main(String[] args) {
        Outer outer = new Outer();
        Outer.Inner inner = outer.new Inner(); // Creating an instance of Inner class
        inner.show();
    }
}
```

Output:

Hello from Outer class

Static Nested Class(Static Inner Class)

1.2 Static Nested Class (Static Inner Class)

- Declared **inside another class** but **marked as static**.
- Can **only access static members** of the outer class.

Example:

```
class Outer {
   private String message = "Hello from Outer class";

   class Inner {
      void show() {
         System.out.println(message); // Accessing private member of Outer
      }
   }
}
```

Example:

```
public class StaticInnerClassExample {
    public static void main(String[] args) {
        Outer.StaticInner inner = new Outer.StaticInner(); // No need for Outer instance
        inner.show();
    }
}
```

Output:

Hello from Static Inner Class

Local Inner Class (Inside a Method)

1.3 Local Inner Class (Inside a Method)

- Defined **inside a method** and **accessible only within that method**.
- Cannot have public, private, or protected modifiers.

Example:

```
class Outer {
   void display() {
      class LocalInner {
         void show() {
            System.out.println("Inside Local Inner Class");
         }
      }
      LocalInner inner = new LocalInner();
      inner.show();
   }
}
```

Example:

```
public class LocalInnerClassExample {
    public static void main(String[] args) {
        Outer outer = new Outer();
        outer.display();
    }
}
```

Output:

Inside Local Inner Class

Anonymous Inner Class

2. Anonymous Inner Class

- An **anonymous inner class** is a **class without a name** that is created for **one-time use**.
- It is mainly used to implement **interfaces or abstract classes on the spot**.

Features of Anonymous Inner Class:

✅Does not have a name
✅ Used for **short-lived objects**
✅ Declared and instantiated in a single expression
✅ Cannot have constructors
✅ Useful for **event handling and callbacks**

Example 1: Anonymous Inner Class Implementing an Interface

Example:

```
interface Greeting {
    void sayHello();
}
```

Example 1: Anonymous Inner Class Implementing an Interface

```
public class AnonymousInnerClassExample {
    public static void main(String[] args) {
        // Anonymous class implementing the Greeting interface
        Greeting greet = new Greeting() {
            public void sayHello() {
                System.out.println("Hello from Anonymous Inner Class");
            }
        };
        greet.sayHello();
    }
}
```

Output:

Hello from Anonymous Inner Class

Example 2: Anonymous Inner Class Extending a Class

Example:

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

Example 2: Anonymous Inner Class Extending a Class

```
public class AnonymousClassExample {
    public static void main(String[] args) {
        Animal dog = new Animal() {  // Creating an anonymous subclass of Animal
            void makeSound() {
                System.out.println("Bark! Bark!");
            }
        };
        dog.makeSound();
    }
}
```

Output:

```
Bark! Bark!
```

Comparison: Inner Class vs Anonymous Inner Class

Feature	Inner Class	Anonymous Inner Class
Named?	Yes	No
Use case	Used when multiple instances are needed	Used for one-time use
Instantiation	Explicitly instantiated	Created inline
Can have constructors?	Yes	No

Conclusion

- **Inner classes** are useful for logically grouping related functionality within a class.
- **Anonymous inner classes** are best suited for **short-lived implementations** like event handling and interface implementations.

KGiSL MicroCollege

Genius as a Service (GaaS)

THE COMPLETE CORE JAVA COURSE

PRESENTED BY,

KGM TECHNICAL TEAM

DAY-04 & 05 COMPLETED 🎉