* Create a class with members of all access modifiers. Try accessing each

2. 3. 4. 5. 6. 7. 8. 9. from:

a. Same class

b. Same package

c. Different package

d. Subclass in a different package

package com.example.modifiers;

public class MyClass {

*// Private member: Accessible only within MyClass*

private int privateVar = 10;

*// Default (package-private) member: Accessible within com.example.modifiers package*

int defaultVar = 20;

*// Protected member: Accessible within com.example.modifiers package and by subclasses*

protected int protectedVar = 30;

*// Public member: Accessible from anywhere*

public int publicVar = 40;

*// Private method*

private void privateMethod() {

System.out.println("Private method called.");

}

*// Default method*

void defaultMethod() {

System.out.println("Default method called.");

}

*// Protected method*

protected void protectedMethod() {

System.out.println("Protected method called.");

}

*// Public method*

public void publicMethod() {

System.out.println("Public method called.");

}

*// Method to demonstrate access from the same class*

public void accessFromSameClass() {

System.out.println("Accessing from Same Class:");

System.out.println("Private Var: " + privateVar);

System.out.println("Default Var: " + defaultVar);

System.out.println("Protected Var: " + protectedVar);

System.out.println("Public Var: " + publicVar);

privateMethod();

defaultMethod();

protectedMethod();

publicMethod();

}

* Can private methods be inherited? Show this through a method in the

superclass and a method with the same name in the subclass.

Private methods in a superclass are not directly accessible or inheritable by subclasses in most object-oriented programming languages like Java. While the superclass's private methods exist within the superclass's memory space, they are encapsulated and hidden from external access, including from its subclasses.

When a subclass defines a method with the same name as a private method in its superclass, it is not considered an override. Instead, it is a new, independent method specific to the subclass. The subclass's method does not interact with or inherit from the superclass's private method.

Here's an illustration using Java:

Java

class Superclass {

private void privateMethod() {

System.out.println("Private method in Superclass.");

}

public void callPrivateMethod() {

privateMethod(); *// Accessible within the Superclass*

}

}

class Subclass extends Superclass {

*// This is a new, independent private method in Subclass, not an override.*

private void privateMethod() {

System.out.println("Private method in Subclass.");

}

public void callSubclassPrivateMethod() {

privateMethod(); *// Calls the privateMethod of Subclass*

}

}

public class Main {

public static void main(String[] args) {

Subclass obj = new Subclass();

*// obj.privateMethod(); // This would result in a compilation error*

*// because privateMethod is not accessible from outside its class.*

obj.callSubclassPrivateMethod(); *// Calls the privateMethod in Subclass*

*// If you want to call the superclass's private method,*

*// you must do so through a public/protected method within the superclass itself.*

obj.callPrivateMethod(); *// Calls the privateMethod in Superclass via callPrivateMethod()*

}

}

In this example:

Superclass has a privateMethod().

Subclass also defines a privateMethod(). This is a new method in Subclass, not an override of the one in Superclass.

Attempting to call privateMethod() directly on a Subclass object from Main would lead to a compilation error, demonstrating its private scope.

The callPrivateMethod() in Superclasscan access its own privateMethod().

The callSubclassPrivateMethod() in Subclass can access its own privateMethod().

When obj.callPrivateMethod() is invoked, it executes the privateMethod() defined within the Superclass context, even though obj is an instance of Subclass. This is because callPrivateMethod() is a public method of Superclass that internally invokes its own private

* Build an abstract class with protected abstract methods and test how

subclasses implement and access them.

Java

*// Abstract class with protected abstract methods*

public abstract class AbstractWorker {

*// Protected abstract method 1*

protected abstract void performTaskA();

*// Protected abstract method 2*

protected abstract String getWorkerType();

*// Concrete method that uses the abstract methods*

public void startWork() {

System.out.println("Starting work for " + getWorkerType());

performTaskA();

System.out.println(getWorkerType() + " work completed.");

}

}

*// Subclass implementing the abstract methods*

class Engineer extends AbstractWorker {

@Override

protected void performTaskA() {

System.out.println("Engineer is designing a blueprint.");

}

@Override

protected String getWorkerType() {

return "Engineer";

}

*// Engineer-specific method accessing inherited protected methods*

public void reviewDesign() {

System.out.println("Engineer reviewing design, type: " + getWorkerType());

*// Can call performTaskA() here if needed, as it's protected*

}

}

*// Another subclass implementing the abstract methods*

class Developer extends AbstractWorker {

@Override

protected void performTaskA() {

System.out.println("Developer is writing code.");

}

@Override

protected String getWorkerType() {

return "Developer";

}

}

*// Test class*

public class AbstractClassTest {

public static void main(String[] args) {

*// Create instances of subclasses*

Engineer engineer = new Engineer();

Developer developer = new Developer();

*// Call the concrete method inherited from the abstract class*

engineer.startWork();

System.out.println("---");

developer.startWork();

System.out.println("---");

*// Accessing subclass-specific method that might use inherited protected methods*

engineer.reviewDesign();

}

}

* What happens when a superclass constructor is private? Try instantiating a subclass.

When a superclass constructor is declared as private, it means that the constructor can only be accessed and invoked from within the superclass itself. This has significant implications for subclassing.

Attempting to Instantiate a Subclass:

When a subclass is instantiated, its constructor implicitly or explicitly calls a constructor of its superclass using super(). If the only constructor (or the default no-argument constructor) in the superclass is private, then the subclass cannot access it. This results in a compile-time error because the subclass cannot fulfill the requirement of calling a superclass constructor.

Example (Java):

Java

class SuperClass {

private SuperClass() {

System.out.println("SuperClass private constructor called.");

}

}

class SubClass extends SuperClass {

public SubClass() {

*// This line implicitly calls super(), which attempts to access the private constructor of SuperClass.*

*// This will result in a compile-time error.*

System.out.println("SubClass constructor called.");

}

}

public class Main {

public static void main(String[] args) {

*// new SubClass(); // This line would cause a compile-time error*

}

}

* Implement multilevel inheritance and show how constructors in each

level behave.

Multilevel inheritance in Java involves a chain of inheritance where a class extends another class, which in turn extends another class, and so on.

Constructor Behavior in Multilevel Inheritance:

When an object of the lowest-level subclass in a multilevel inheritance hierarchy is created, the constructors are executed in a specific order:

**Topmost Superclass Constructor:**  
The constructor of the highest-level superclass in the hierarchy is executed first.

**Intermediate Superclass Constructors:**  
Subsequently, the constructors of intermediate superclasses are executed in descending order, from the highest to the lowest.

**Subclass Constructor:**  
Finally, the constructor of the class whose object is being created is executed.

This order is enforced because each subclass constructor implicitly or explicitly calls its immediate superclass's constructor using the super() keyword. If super() is not explicitly called, the compiler automatically inserts a call to the default (no-argument) superclass constructor. If the superclass does not have a default constructor and no explicit super() call is made, a compilation error occurs.

Multilevel inheritance involves a chain of inheritance where a class inherits from another class, which in turn inherits from another class. Constructors in such a hierarchy are called in a specific order: from the topmost base class down to the most derived class.

*/ Topmost Superclass*

class Grandparent {

public Grandparent() {

System.out.println("Grandparent constructor executed.");

}

}

*// Intermediate Superclass*

class Parent extends Grandparent {

public Parent() {

*// super() is implicitly called here if not explicitly provided*

System.out.println("Parent constructor executed.");

}

}

*// Subclass*

class Child extends Parent {

public Child() {

*// super() is implicitly called here if not explicitly provided*

System.out.println("Child constructor executed.");

}

}

public class MultilevelInheritanceExample {

public static void main(String[] args) {

System.out.println("Creating a Child object:");

Child myChild = new Child();

}

}

* What issues might arise if a superclass has overloaded constructors but

no default constructor?

If a superclass has overloaded constructors but no explicitly defined default (no-argument) constructor, and a subclass attempts to instantiate itself without explicitly calling a superclass constructor, a compile-time error will occur.

This issue arises because:

**Implicit Superclass Constructor Call:**  
When a subclass constructor does not explicitly call a superclass constructor using super(), the compiler automatically inserts an implicit call to the superclass's no-argument constructor.

**Missing Default Constructor:**  
If the superclass has overloaded constructors but no no-argument constructor is explicitly defined, and no implicit one is generated (which happens when any parameterized constructor is defined), the compiler cannot find a suitable constructor to call implicitly.

**Compiler Error:**  
Consequently, the compiler reports an error, indicating that the superclass lacks a no-argument constructor.

// Superclass with only a parameterized constructor

class SuperClass {

public SuperClass(String name) {

System.out.println("SuperClass constructor with name: " + name);

}

}

// Subclass attempting to instantiate without explicit super() call

class SubClass extends SuperClass {

// This will cause a compile-time error

public SubClass() {

// Implicit call to super() is inserted here by the compiler

// but SuperClass has no no-argument constructor

System.out.println("SubClass constructor");

}

}

* Create a superclass Animal and subclasses Dog, Cat, Cow. Override the

makeSound() method. Loop through a list of Animal and invoke

makeSound() using polymorphic references.

// Superclass

class Animal {

public void makeSound() {

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

}

}

// Subclass Dog

class Dog extends Animal {

@Override

public void makeSound() {

System.out.println("Dog barks: Woof! Woof!");

}

}

// Subclass Cat

class Cat extends Animal {

@Override

public void makeSound() {

System.out.println("Cat meows: Meow!");

}

}

// Subclass Cow

class Cow extends Animal {

@Override

public void makeSound() {

System.out.println("Cow moos: Moo!");

}

}

public class PolymorphismExample {

public static void main(String[] args) {

// Create a list of Animal objects using polymorphic references

Animal[] animals = new Animal[4];

animals[0] = new Dog();

animals[1] = new Cat();

animals[2] = new Cow();

animals[3] = new Animal(); // Demonstrate the superclass method

// Loop through the list and invoke makeSound()

for (Animal animal : animals) {

animal.makeSound(); // Polymorphic method call

}

}

}

* Create a parent class Account and subclasses like SavingsAccount,

FixedDepositAccount. Implement interest calculation differently in each

subclass. Instantiate and print account details.

// BankAccount.java

// Parent class BankAccount

// Declare the BankAccount class

public class BankAccount {

// Private field to store the account number

private String accountNumber;

// Private field to store the balance

private double balance;

// Constructor to initialize account number and balance

public BankAccount(String accountNumber, double balance) {

this.accountNumber = accountNumber;

this.balance = balance;

}

// Method to deposit an amount into the account

public void deposit(double amount) {

// Increase the balance by the deposit amount

balance += amount;

}

// Method to withdraw an amount from the account

public void withdraw(double amount) {

// Check if the balance is sufficient for the withdrawal

if (balance >= amount) {

// Decrease the balance by the withdrawal amount

balance -= amount;

} else {

// Print a message if the balance is insufficient

System.out.println("Insufficient balance");

}

}

// Method to get the current balance

public double getBalance() {

// Return the current balance

return balance;

}

}

* Create a superclass with a protected method, extend it in another

package, and verify if you can invoke that method directly

// package1/Superclass.java

package package1;

public class Superclass {

protected void protectedMethod() {

System.out.println("Protected method invoked from Superclass.");

}

}

// package2/Subclass.java

package package2;

import package1.Superclass; // Import the superclass

public class Subclass extends Superclass {

public void invokeProtectedFromSubclass() {

// This is a valid invocation because Subclass extends Superclass

protectedMethod();

System.out.println("Protected method invoked from Subclass instance.");

}

}

// package2/Main.java

package package2;

import package1.Superclass; // Import the superclass

public class Main {

public static void main(String[] args) {

Subclass subclass = new Subclass();

subclass.invokeProtectedFromSubclass(); // Valid: Invokes the protected method through a subclass method

// Attempting to directly invoke the protected method on a Superclass object

// from a different package and a non-subclass context will result in a compilation error.

// Superclass superObject = new Superclass();

// superObject.protectedMethod(); // This line would cause a compile-time error

}

}