**·     Write program for Runtime Polymorphism and Compile Polymorphism.**

**Ans –**

Program for compile time polymorphism is as follows –

package org.example;  
  
public class Polymorphism {  
 public static int add(int a, int b){  
 return a+b;  
 }  
 public static float add(float a, float b){  
 return a+b;  
 }  
 public static int add(int a, int b, int c) {  
 return a+b+c;  
 }  
 public static void main(String[] args) {  
 *add*(1,2); // 3  
 float x = *add*(2.3f, 4f); // 6.3  
 *add*(1,2,3); // 6  
 }  
}

Program for runtime polymorphism is as follows –

package org.example;  
  
class Employee {  
 int bonus() {  
 return 500;  
 }  
}  
class Sde extends Employee{  
 @Override  
 int bonus() {  
 return 1000;  
 }  
}  
class Manager extends Employee {  
 @Override  
 int bonus(){  
 return 2000;  
 }  
}  
  
public class Polymorphism {  
 public static void main(String[] args) {  
 Sde s = new Sde();  
 Manager m = new Manager();  
 System.*out*.println(s.bonus()); // 1000  
 System.*out*.println(m.bonus()); // 2000  
 }  
}

We achieved compile time polymorphism by method overloading and runtime polymorphism by method overriding.

**·     Explain why composition preferred over inheritance?**

**Ans –**

Composition is preferred over inheritance for several reasons:

**Flexibility and Reusability:** With inheritance, a subclass is tightly coupled to its superclass, and it inherits all the properties and methods of the superclass. With composition, classes can be composed of smaller, more modular components that can be easily reused and modified.

**Avoiding the Fragile Base Class Problem:** The Fragile Base Class Problem is a common issue with inheritance, where changes to the superclass can have unintended effects on the subclasses. With composition, the component can be modified independently of the rest of the codebase**.**

**Separation of Concerns:** Inheritance can lead to code that is tightly coupled and difficult to separate into separate concerns. Composition, on the other hand, allows for more modular and reusable components that can be combined in different ways to achieve different goals.

Overall, composition can lead to more flexible, maintainable, and reusable code than inheritance.

# ·     Can class extend enum? Can Enum implement interface? What is ordinal in Enum?

# Ans –

# Unlike classes enumerations neither inherits other classes nor can get extended. Yes, enumerations can implement interfaces. In enum ordinal is used to get index of respective value.

**·     Explain forLoop, ForEach which one is faster?**

**Ans –** In general, for-each loop is typically faster and more concise when iterating over arrays and other collections, such as ArrayList, because it automatically handles the index checking and retrieval of each element in the collection. On the other hand, for loops is more suitable when the iteration logic is more complex and requires explicit control over the index variable. However, the performance difference between for and for-each is negligible in most cases.

**·     Write a program for an Enum which has private constructor  which accepts two args?**

**Ans –**

package org.example;  
  
enum Colour {  
 *RED*(10 ,20),  
 *GREEN*(30, 40),  
 *BLACK*(50,60),  
 *BLUE*(70, 80);  
 int colorSum;  
 private Colour(int a, int b){  
 colorSum = a + b;  
 }  
 int getSum(){  
 return colorSum;  
 }  
}  
  
public class Enum {  
 public static void main(String[] args) {  
 for (Colour c : Colour.*values*()) {  
 System.*out*.println("COLOUR NAME - " + c + ", Color Sum = "+c.getSum());  
 }  
 }  
}

**·     Explain the types of association with example in java.**

**Ans –** Association is a relationship between two separate classes which is established through there objects. Association can be one-to-one, one-to-many, many-to-one, many-to-many. Composition and aggregation are two types of association.

Aggregation is a weak association. An association is said to be aggregation if both objects can exist independently. For example –

package org.example;  
  
import java.util.ArrayList;  
import java.util.List;  
  
// Company and employee are independent  
public class Company {  
 private List employee; // Employee can be 0 or more5  
 Company(){  
 employee = new ArrayList();  
 }  
}

Composition is a strong association. An association is said to be in composition if an object owns another object and another object cannot exist without the owner object. For example –

package org.example;  
  
// Engine is mandatory part of Car  
public class Car {  
 private Engine engine;  
 Car(){  
 engine = new Engine();  
 }  
}  
  
class Engine{}

**·     Write a program to explain the behaviour of  each access modifiers  in  inheritance**

**Ans –** There are four types of access modifiers in java, they are –

1 – private

2 – default

3 – protected

4 – public

package org.example;  
  
public class Parent {  
 private void displayPrivate() {  
 System.*out*.println("Private Display Method");  
 }  
   
 public void show() {  
 displayPrivate(); // Private members & methods are only accessible inside class  
 }  
  
 protected void displayProtected() {  
 System.*out*.println("Private Display Method");  
 }  
  
 public void displayPublic() {  
 System.*out*.println("Public Display Method");  
 }  
}

package org.example;  
  
class Child extends Parent {  
  
}  
public class Modifier {  
 public static void main(String[] args) {  
 Parent parent = new Parent();  
 parent.show();  
 Child obj = new Child();  
 obj.displayPublic(); // Works Fine because of public access  
 //obj.displayPrivate(); // Not accessible due to private access  
 }  
}

package org.newPackage;  
import org.example.\*; // Imported class from other package to explain  
 // default and protected  
  
// Parent is extended in another package due to public access  
// When parent had default access it was not accessible in another package  
public class Test extends Parent{  
 public static void main(String[] args) {  
 Test obj = new Test();  
 // protected modifier allows access in other packages also.  
 obj.displayProtected();  
 }  
}

# ·     What is Autoboxing and unboxing in java, write a program.

# Ans – Autoboxing refers to the conversion of a primitive value into an object of the corresponding wrapper class. Unboxing on the other hand refers to the conversion of an object of a wrapper type to its corresponding primitive value. For example –

# int a = 20;

# Integer b = a; // Autoboxing

# int c = b; // Unboxing