Attempt the following

1. Explain the keywords this and super.

In java, **super** keyword is used to access methods of the **parent class** while **this** is used to access methods of the **current class**.

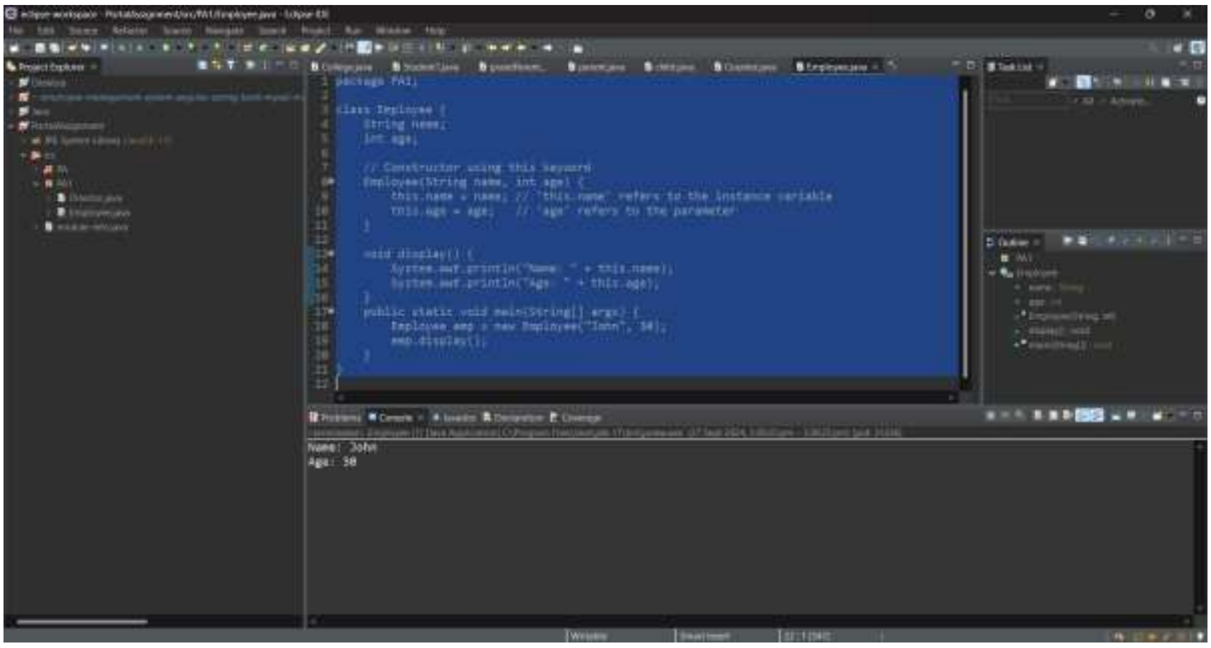
this keyword is a reserved keyword in java i.e, we can't use it as an identifier. It is used to refer current class's instance as well as static members. It can be used in various contexts as given below:

- to refer instance variable of current class
- to invoke or initiate current class constructor
- can be passed as an argument in the method call
- can be passed as argument in the constructor call
- can be used to return the current class instance

Example

```
package PA1;

class Employee {
    String name;
    int age;

    // Constructor using this keyword
    Employee(String name, int age) {
        this.name = name; // 'this.name' refers to the instance
variable
        this.age = age;   // 'age' refers to the parameter
    }

    void display() {
        System.out.println("Name: " + this.name);
        System.out.println("Age: " + this.age);
    }
    public static void main(String[] args) {
        Employee emp = new Employee("John", 30);
        emp.display();
    }
}
```

Output:-



```
package FA1;

class Employee {
    String name;
    int age;

    // Constructor using this keyword
    Employee(String name, int age) {
        this.name = name; // 'this.name' refers to the instance variable
        this.age = age;    // 'age' refers to the parameter
    }

    void display() {
        System.out.println("Name: " + this.name);
        System.out.println("Age: " + this.age);
    }
    public static void main(String[] args) {
        Employee emp = new Employee("John", 30);
        emp.display();
    }
}
```

```
Name: John
Age: 30
```

super keyword

1. super is a reserved keyword in java i.e, we can't use it as an identifier.
2. super is used to refer super-class's instance as well as static members.
3. super is also used to invoke super-class's method or constructor.
4. super keyword in java programming language refers to the superclass of the class where the super keyword is currently being used.

5. The most common use of super keyword is that it eliminates the confusion between the superclasses and subclasses that have methods with same name.

super can be used in various contexts as given below:

- it can be used to refer immediate parent class instance variable
- it can be used to refer immediate parent class method
- it can be used to refer immediate parent class constructor.

Example

Animal.java

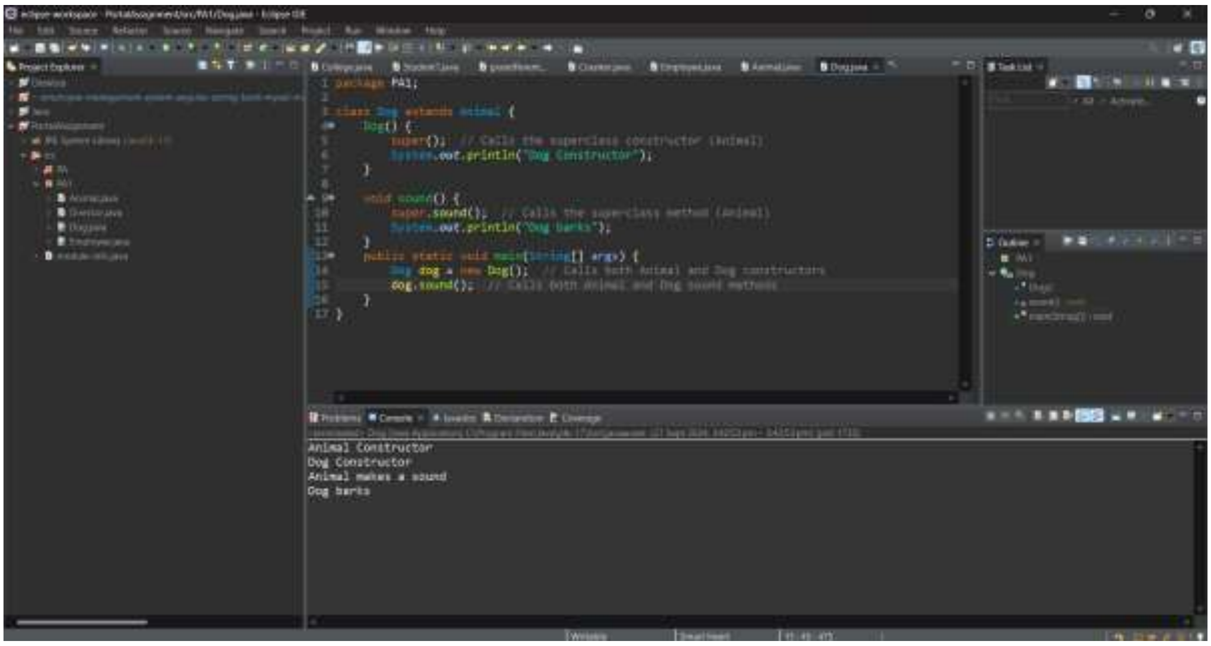
```
package PA1;

public class Animal {
        Animal() {
            System.out.println("Animal Constructor");
        }

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

    public static void main(String[] args) {
         // TODO Auto-generated method stub

    }

}
```

Dog.java

```
package PA1;

```

```java
class Dog extends Animal {
    Dog() {
        super();  // Calls the superclass constructor (Animal)
        System.out.println("Dog Constructor");
    }

    void sound() {
        super.sound();  // Calls the superclass method (Animal)
        System.out.println("Dog barks");
    }
    public static void main(String[] args) {
        Dog dog = new Dog();  // Calls both Animal and Dog
constructors
        dog.sound();  // Calls both Animal and Dog sound methods
    }
}
```

<u>Output:-</u>



```java
package PA1;

class Dog extends Animal {
    Dog() {
        super();  // Calls the superclass constructor (Animal)
        System.out.println("Dog Constructor");
    }

    void sound() {
        super.sound();  // Calls the superclass method (Animal)
        System.out.println("Dog barks");
    }
    public static void main(String[] args) {
        Dog dog = new Dog();  // Calls both Animal and Dog constructors
        dog.sound();  // Calls both Animal and Dog sound methods
    }
}
```

```
Animal Constructor
Dog Constructor
Animal makes a sound
Dog barks
```

2. Explain the concepts of garbage collection and finalize().

1. Garbage Collection in Java:

Garbage collection is the process by which Java automatically manages memory. It reclaims memory that is no longer in use, which helps prevent memory leaks and optimizes memory usage. In Java, objects are created in memory (on the heap), and when they are no longer reachable by the program (i.e., when there are no more references to them), the garbage collector identifies these objects as candidates for deletion.

Key Points:

- **Automatic Memory Management**: Java manages memory through the garbage collector, so developers don’t need to manually free up memory as in languages like C or C++ (which use functions like free() or delete).
- **Heap Space**: All objects are allocated memory in the heap, and when no live references point to them, they become eligible for garbage collection.
- **Garbage Collector**: The garbage collector works in the background, identifying unused objects and reclaiming memory.

How Garbage Collection Works:

The garbage collector follows certain algorithms to identify which objects are no longer in use, such as:

- **Reference Counting**: The number of references to an object is counted, and when the count reaches zero, the object is eligible for garbage collection.
- **Mark and Sweep**: The garbage collector traverses the object graph, starting from root objects (like static variables and local variables), and marks all

reachable objects. Then it sweeps through and collects the unmarked (unreachable) objects.

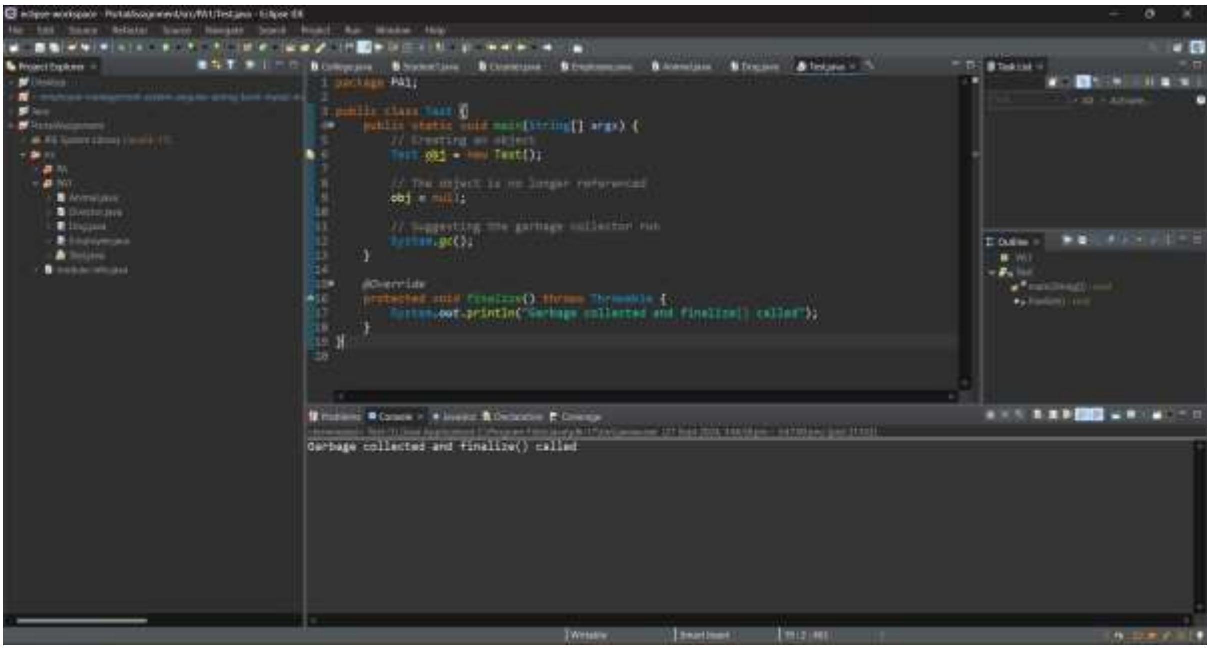
When does garbage collection happen?

Garbage collection occurs automatically during program execution when the Java Virtual Machine (JVM) detects that the heap is becoming full. However, developers can suggest garbage collection by calling System.gc(), but there is no guarantee it will run immediately or at all, as garbage collection is managed by the JVM.

Example:

```java
package PA1;

public class Test {
    public static void main(String[] args) {
        // Creating an object
        Test obj = new Test();

        // The object is no longer referenced
        obj = null;

        // Suggesting the garbage collector run
        System.gc();
    }

    @Override
    protected void finalize() throws Throwable {
        System.out.println("Garbage collected and finalize()
called");
    }
}
```

Output:-



```java
package PA1;

public class Test {
    public static void main(String[] args) {
        // Creating an object
        Test obj = new Test();

        // The object is no longer referenced
        obj = null;

        // Suggesting the garbage collector run
        System.gc();
    }

    @Override
    protected void finalize() throws Throwable {
        System.out.println("Garbage collected and finalize() called");
    }
}
```

```
Garbage collected and finalize() called
```

2. finalize() Method:

The finalize() method is a special method in Java that is called by the garbage collector before an object is destroyed. The purpose of the finalize() method is to give the object an opportunity to perform cleanup tasks before it is reclaimed by the garbage collector.

Key Points:

- Used for Cleanup: finalize() can be used to close resources like file streams or database connections, or to perform any other necessary cleanup.
- Deprecated: As of Java 9, the finalize() method is deprecated because its use is generally unreliable. Developers should use other mechanisms like try-with-resources or explicit resource management (e.g., using finally blocks).
- Not Guaranteed: There is no guarantee that finalize() will be called for an object before it is

garbage collected. If the garbage collector does not run or the program exits before it runs, finalize() may not execute.

Example:

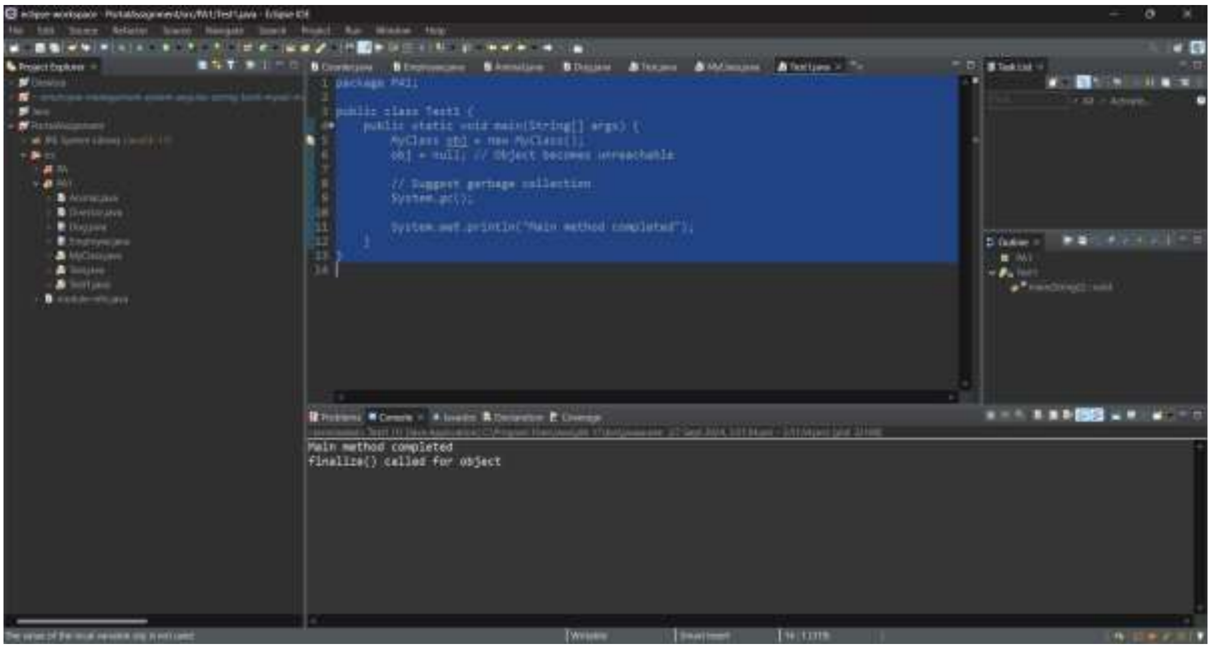
MyClass.java

```
package PA1;

public class MyClass {
         @Override
         protected void finalize() throws Throwable {
             System.out.println("finalize() called for object");
             // Cleanup code
             super.~~finalize~~();
         }

     public static void main(String[] args) {
           // TODO Auto-generated method stub

     }

}
```

Test1.java

```
package PA1;

public class Test1 {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        obj = null; // Object becomes unreachable

        // Suggest garbage collection
        System.gc();

        System.out.println("Main method completed");
    }
}
```

Output:-



```java
package PA1;

public class Test1 {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        obj = null; // Object becomes unreachable

        // Suggest garbage collection
        System.gc();

        System.out.println("Main method completed");
    }
}
```

```
Main method completed
finalize() called for object
```

3. Explain the purpose of package and how to import package.

A **package** in Java is a mechanism used to group related classes, interfaces, and sub-packages into a namespace. This helps organize code and prevent naming conflicts, especially in large projects.

Purpose of Packages:

1. **Namespace Management**: Packages prevent class name conflicts by providing namespaces. For example, two classes can have the same name if they are in different packages.
2. **Code Organization**: Packages help organize related classes and interfaces logically, making the project structure easier to manage.
3. **Access Control**: Packages offer a level of access control. Classes in the same package can access

each other's package-private members (i.e., members without an access modifier).

4. **Reusability**: You can group related classes into a package, which can be reused across multiple projects.

How to Create and Import Packages:

1. **Creating a Package**: To declare a package, you use the package keyword at the top of a Java file. For example:

```
package PA1;

public class utilities {

   public static void printMessage() {

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

   }

}
```

The above code creates a package named com.example.utilities, and the Utility class belongs to this package.

2. **Importing a Package**: To use a class from another package, you need to import it using the import keyword. For example:

```
import PA1.utilities.Utility;
```

```
public class Main {
    public static void main(String[] args) {
        Utility.printMessage();  // Calling method from
Utility class
    }
}
```

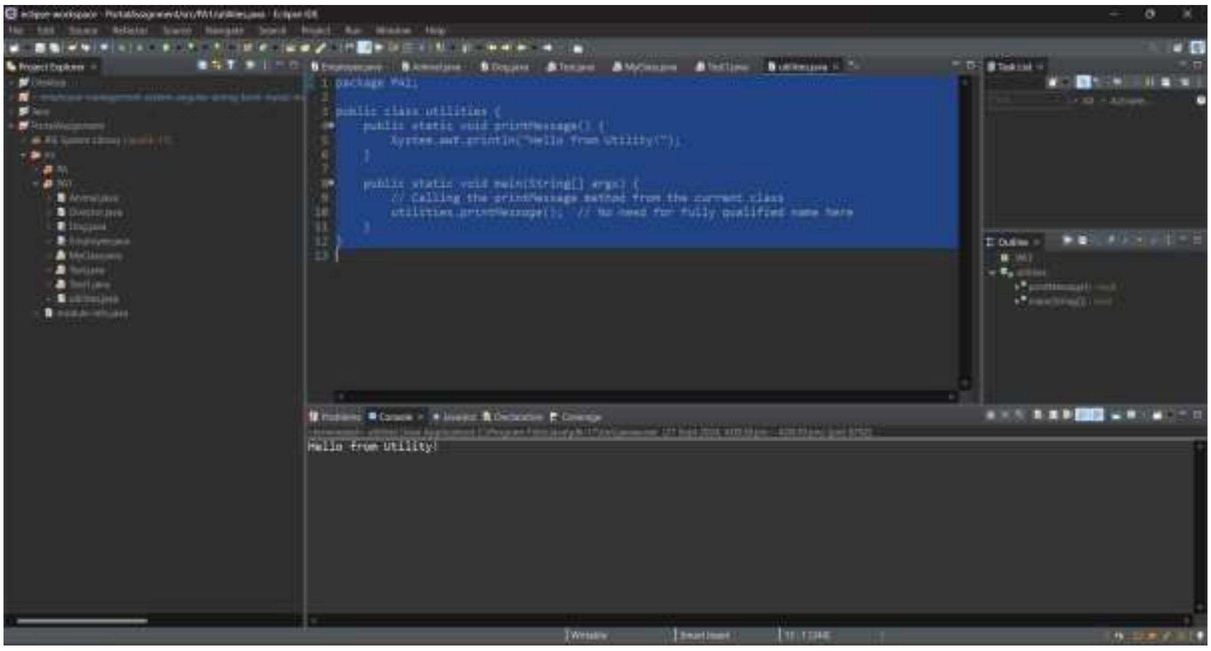
Alternatively, you can use wildcard (*) to import all classes from a package:

```
import PA1.utilities.*;
```

3. **Accessing Without Importing**: You can also use a fully qualified name to access a class without importing it:

```
package PA1;

public class utilities {
    public static void printMessage() {
        System.out.println("Hello from Utility!");
    }

    public static void main(String[] args) {
        // Calling the printMessage method from the current class
        utilities.printMessage();  // No need for fully qualified
name here
    }
}
```

Output:-



```java
package PA1;

public class utilities {
    public static void printMessage() {
        System.out.println("Hello from Utility!");
    }

    public static void main(String[] args) {
        // Calling the printMessage method from the current class
        utilities.printMessage();  // No need for fully qualified name here
    }
}
```

```
Hello from Utility!
```