Part A: Object-Oriented Programming Questions

1. Object-oriented programming uses classes and objects. What are classes and what are objects? What is the relationship between classes and objects?

(3 Marks)

In object-oriented programming (OOP), **classes** and **objects** are foundational concepts that support code organization, reusability, and modularity.

A **class** is a blueprint or template that defines the structure and behavior of entities. It encapsulates data (fields or attributes) and actions (methods or functions) that the entities can perform. A class does not occupy memory until an object is created from it.

An **object** is a concrete instance of a class. It represents a real-world entity with specific values for its attributes and can invoke the methods defined in the class. Objects are created at runtime and occupy memory space.

The relationship between classes and objects is that of a **template to instance**. A class defines what objects will look like and how they behave, while objects are the actual implementations based on that definition.

Example:

```
class Car {
    String color;
    String model;

    void start() {
        System.out.println("Car started");
    }
}

// Creating objects
Car car1 = new Car();
Car car2 = new Car();

car1.color = "Red";
car1.model = "Sedan";

car2.color = "Blue";
car2.model = "SUV";
```

In this example, `Car` is the class, and `car1`, `car2` are objects (instances) of that class. Each object holds its own data but shares the same structure and behavior defined by the class.

2. Why can't we use the "super" and "this" keywords in a static method in Java?

(3 Marks)

The `this` keyword refers to the current instance of the class, while `super` refers to the immediate parent class instance. Both are used to access instance-specific members — such as variables and methods — of the current or superclass object.

However, **static methods** belong to the class rather than any instance. They are loaded at class loading time and can be called without creating an object of the class.

Since static methods do not operate in the context of an object, there is no instance to refer to. Therefore:

- `this` cannot be used because there is no “current object” in a static context.

- `super` cannot be used because it refers to a parent object, which also requires an instance to exist.

Attempting to use `this` or `super` in a static method results in a **compile-time error**.

Example:

```
class Parent {
    int x = 10;

    void display() {
        System.out.println(x);
    }
}

class Child extends Parent {
    int y = 20;

    static void staticMethod() {
        // this.y = 30;        // ❌ Error: Cannot use 'this' in static context
        // super.display();    // ❌ Error: Cannot use 'super' in static context
    }
}
```

Hence, `this` and `super` are invalid in static methods due to their dependency on object instances, which are absent in static execution contexts.

3. When do we declare a method or class as final? Explain with an example.

(3 Marks)

In Java, the `final` keyword is used to impose restrictions on inheritance and modification.

A **method is declared final** when we want to prevent it from being overridden in subclasses. This ensures that the method's implementation remains consistent across all classes.

A **class is declared final** when we do not want it to be extended by other classes. This prevents inheritance entirely.

Common use cases include:

- Securing critical functionality (e.g., security-related methods).
- Improving performance (JVM can optimize final methods).
- Designing immutable or utility classes (e.g., `String` is a final class).

Example:

```
final class Bank {
    final void withdraw() {
        System.out.println("Withdrawal process initiated");
    }
}

// class SavingsAccount extends Bank {
//     // ❌ Compilation error: Cannot inherit from final 'Bank'
// }
```

In this example, the `Bank` class is final, so no class can inherit from it. The `withdraw()` method is also final, meaning it cannot be overridden — even if the class weren't final.

Another example:

```
class Animal {
    final void eat() {
        System.out.println("Animal eats food");
    }
}

class Dog extends Animal {
    // void eat() { }
    // ❌ Error: Cannot override final method 'eat()'
}
```

Thus, the `final` keyword helps enforce design constraints, prevent unintended behavior changes, and maintain integrity in class hierarchies.