how to design two  classes with same attribute, such that client application can consume relating its data

Designing Two Classes with the Same Attributes for Client Consumption

If two classes have the same attributes, but the client application needs to relate their data, you can approach the design in several ways:

🔷 Approach 1: Inheritance (Extending a Common Parent Class)

📌 Use Case:

- When both classes share the same structure, but one may need additional behavior.

```
// Common Parent Class
class BaseEntity {
    protected int id;
    protected String name;

    public BaseEntity(int id, String name) {
        this.id = id;
        this.name = name;
    }

    public int getId() { return id; }
    public String getName() { return name; }
}

// Two Child Classes
class Employee extends BaseEntity {
    public Employee(int id, String name) {
        super(id, name);
    }
}

class Customer extends BaseEntity {
    public Customer(int id, String name) {
        super(id, name);
    }
}
```

✅ Client Application Can Use:

```
BaseEntity emp = new Employee(1, "Alice");
BaseEntity cust = new Customer(2, "Bob");

System.out.println(emp.getName());  // Alice
System.out.println(cust.getName()); // Bob
```

🔷 Pros:

✔ Code reuse through inheritance
✔ Polymorphic behavior

🔷 Cons:

❌ Not flexible if both classes require different behaviors

🔷 Approach 2: Implementing a Common Interface

📌 Use Case:

- When both classes should be treated similarly but implement their own behavior.

```
// Define a Common Interface
interface Identifiable {
    int getId();
    String getName();
}

// Implementing Classes
class Employee implements Identifiable {
    private int id;
    private String name;

    public Employee(int id, String name) {
        this.id = id;
        this.name = name;
    }

    public int getId() { return id; }
    public String getName() { return name; }
}

class Customer implements Identifiable {
    private int id;
    private String name;

    public Customer(int id, String name) {
        this.id = id;
        this.name = name;
    }

    public int getId() { return id; }
    public String getName() { return name; }
}
```

✅ Client Application Can Use:

```
List<Identifiable> entities = Arrays.asList(
    new Employee(1, "Alice"),
    new Customer(2, "Bob")
);

for (Identifiable entity : entities) {
    System.out.println(entity.getName());
}
```

🔷 Pros:

✔ More flexible than inheritance
✔ Supports multiple implementations

🔷 Cons:

❌ No default behavior, each class must implement the methods

🔷 Approach 3: Using a Generic Wrapper (Composition)

📌 Use Case:

- When you need aggregation of both types under a common wrapper.

```
class Entity<T> {
    private T data;

    public Entity(T data) {
        this.data = data;
    }

    public T getData() { return data; }
}

class Employee {
    int id;
    String name;

    public Employee(int id, String name) {
        this.id = id;
        this.name = name;
    }
}

class Customer {
    int id;
    String name;

    public Customer(int id, String name) {
        this.id = id;
        this.name = name;
    }
}
```

✅ Client Application Can Use:

```
Entity<Employee> empEntity = new Entity<>(new Employee(1, "Alice"));
Entity<Customer> custEntity = new Entity<>(new Customer(2, "Bob"));

System.out.println(empEntity.getData().name);  // Alice
System.out.println(custEntity.getData().name); // Bob
```

🔷 Pros:

✔ Works well for encapsulation
✔ Avoids inheritance issues

🔷 Cons:

❌ Slightly more complex

🔷 Approach 4: DTO (Data Transfer Object) with Mapping

📌 Use Case:

- When you map both classes to a common DTO for external communication.

```
class PersonDTO {
    private int id;
    private String name;

    public PersonDTO(int id, String name) {
        this.id = id;
        this.name = name;
    }

    public static PersonDTO fromEmployee(Employee emp) {
        return new PersonDTO(emp.id, emp.name);
    }

    public static PersonDTO fromCustomer(Customer cust) {
        return new PersonDTO(cust.id, cust.name);
    }

    public int getId() { return id; }
    public String getName() { return name; }
}
```

✅ Client Application Can Use:

```
Employee emp = new Employee(1, "Alice");
Customer cust = new Customer(2, "Bob");

PersonDTO empDTO = PersonDTO.fromEmployee(emp);
PersonDTO custDTO = PersonDTO.fromCustomer(cust);

System.out.println(empDTO.getName());  // Alice
System.out.println(custDTO.getName()); // Bob
```

🔷 Pros:

✔ Ideal for microservices & API responses
✔ Ensures separation of concerns

🔷 Cons:

❌ Additional conversion logic required

🔷 Which Approach Should You Use?

| approach | Best When | Flexibility | Performance |
|---|---|---|---|
| Inheritance | Classes share common behavior | ❌ Low | ✅ High |
| Interface | Different behaviors needed | ✅ High | ✅ High |
| Composition | Encapsulation is needed | ✅ High | ✅ High |
| DTO Mapping | Used in API communication | ✅ High | ❌ Extra processing |

🔷 Conclusion

✅ If the classes only share attributes, use a DTO or a common interface.
✅ If they share logic, use inheritance.
✅ If encapsulation is important, use composition.

Would you like a real-world example of these approaches in a Spring Boot microservice? 🚀