Concept note on DTO (Data Transfer Object) and ModelMapper,

---

1. What is a DTO?

DTO stands for **Data Transfer Object**. It is a simple Java class used to transfer data between different layers of an application — especially between the **controller** and **service** layers or between **API clients** and **servers**.

Key Features:

- Contains **only data** — no business logic.
- Uses **getters and setters**.
- Can include **only the fields you want to expose**, not everything from the entity.
- Often used to avoid exposing internal database structure.

Example:

```
public class CustomerDTO {
    private String customerName;
    private long customerContactNumber;
    private List<LoanDTO> loans;
}
```

This DTO might represent a subset of the actual Customer entity.

---

2. Why Use DTOs?

Reason	Explanation
Security	Hides sensitive entity fields from external clients.
Performance	Sends only needed data — avoids large or nested objects.
Flexibility	You can customize DTOs for different API responses.
Clean Architecture	Keeps controller/service layer decoupled from the entity/model layer.

---

3. What is ModelMapper?

ModelMapper is a Java library that automatically maps one object to another — like **converting an Entity to a DTO** or vice versa.

Benefits:

- Reduces **manual mapping code**.
- Automatically matches fields with **same names**.
- Supports **nested mappings**, **custom converters**, and **field skipping**.

Basic Usage:

Example: Mapping a DTO to Entity

```
ModelMapper modelMapper = new ModelMapper();

Customer customer = modelMapper.map(customerDTO, Customer.class);
```

Example: Mapping Entity to DTO

```
CustomerDTO customerDTO = modelMapper.map(customer, CustomerDTO.class);
```

---

4. How It Fits in a Spring Boot Project

Controller Layer:

Accepts or returns DTOs.

```
@PostMapping("/customer")
public ResponseEntity<String> addCustomer(@RequestBody CustomerDTO
customerDTO) {
   bankService.addCustomer(customerDTO);
   return ResponseEntity.ok("Customer added");
}
```

Service Layer:

Uses ModelMapper to convert between DTO and Entity.

```
public void addCustomer(CustomerDTO customerDTO) {
   Customer customer = modelMapper.map(customerDTO, Customer.class);
   customerRepository.save(customer);
}
```

Summary Table

Concept	DTO	ModelMapper
What is it?	A plain object for data transfer	A library to convert objects
Purpose	Avoid exposing entity structure	Automate object-to-object mapping
Benefits	Security, clarity, separation	Less boilerplate, cleaner code
Used in	API input/output	Service layer (converting DTO ↔ Entity)
