Evaluate Java DTO SOLID

I have a Java class in my project that I want to follow Java Solid Principles. Infer the class's responsibility and look for ways to improve it, ONLY if applicable.

```
package com.toolstore.demo.dto;

import com.fasterxml.jackson.annotation.JsonFormat;
import jakarta.validation.constraints.NotNull;
import lombok.Getter;
import lombok.Setter;

import java.time.LocalDate;

@Setter
@Getter
public class CheckoutRequest {

    @NotNull(message = "Tool code is required")
    private String toolCode;

    @NotNull(message = "Rental day count is required")
    private Integer rentalDayCount;

    @NotNull(message = "Discount percent is required")
    private Integer discountPercent;

    @NotNull(message = "Checkout date is required")
    @JsonFormat(pattern = "MM/dd/yy")
    private LocalDate checkoutDate;

    public CheckoutRequest(String toolCode, Integer rentalDayCount, Integer discountPercent, LocalDate checkoutDate) {
        this.toolCode = toolCode;
        this.rentalDayCount = rentalDayCount;
        this.discountPercent = discountPercent;
        this.checkoutDate = checkoutDate;
    }

    @Override
    public String toString() {
        return "CheckoutRequest{" +
```

````
            "toolCode='" + toolCode + '\'' +
            ", rentalDayCount=" + rentalDayCount +
            ", discountPercent=" + discountPercent +
            ", checkoutDate=" + checkoutDate +
            '}';
    }
}```
````

RouteLLM

✅ Inferred Responsibility

CheckoutRequest is clearly a **DTO (Data Transfer Object)** used to:

- Represent incoming checkout request data (likely from a REST API).
- Carry data between layers (controller → service).
- Provide validation metadata (@NotNull).
- Control JSON deserialization format for checkoutDate.

Its responsibility is **transporting and validating input data** — nothing more.

That's good. ✅ It is already close to SRP-compliant.

✅ SOLID Review

Let's evaluate this strictly under SOLID principles and suggest improvements **only where applicable**.

1️⃣ Single Responsibility Principle (SRP)

Current Responsibility:

- Holds checkout request data
- Defines validation rules
- Defines JSON date format

These are all part of *input contract definition*, so SRP is not violated.

✅ **No major SRP issue**

However, one small improvement:

🔹 Issue: DTO is mutable (@Setter)

Since this represents a request payload, mutability is usually unnecessary and potentially unsafe.

DTOs should ideally be immutable.

Why?

- Prevent accidental mutation in service layer
- Makes object state predictable
- Safer in concurrent environments

✅ Suggested Improvement (Make it Immutable)

```
@Getter
public class CheckoutRequest {

    @NotNull(message = "Tool code is required")
    private final String toolCode;

    @NotNull(message = "Rental day count is required")
    private final Integer rentalDayCount;

    @NotNull(message = "Discount percent is required")
    private final Integer discountPercent;

    @NotNull(message = "Checkout date is required")
    @JsonFormat(pattern = "MM/dd/yy")
    private final LocalDate checkoutDate;

    public CheckoutRequest(String toolCode,
                           Integer rentalDayCount,
                           Integer discountPercent,
                           LocalDate checkoutDate) {
        this.toolCode = toolCode;
        this.rentalDayCount = rentalDayCount;
        this.discountPercent = discountPercent;
        this.checkoutDate = checkoutDate;
    }
}
```

Or even better (if using Lombok):

```
import lombok.Value;

@Value
public class CheckoutRequest {

    @NotNull(message = "Tool code is required")
    String toolCode;

    @NotNull(message = "Rental day count is required")
    Integer rentalDayCount;

    @NotNull(message = "Discount percent is required")
    Integer discountPercent;

    @NotNull(message = "Checkout date is required")
    @JsonFormat(pattern = "MM/dd/yy")
    LocalDate checkoutDate;
}
```

✅ Removes setters
✅ Makes object immutable
✅ Cleaner and safer

2️⃣ Open/Closed Principle (OCP)

DTOs generally don’t violate OCP.

However:

If business validation rules are added here (like checking discount range), that would violate SRP and OCP.

Currently:
✅ No issue.

3️⃣ Liskov Substitution Principle (LSP)

Not applicable — this class does not participate in inheritance.

✅ No issue.

4️⃣ Interface Segregation Principle (ISP)

Not applicable — this class doesn’t implement interfaces.

✅ No issue.

5️⃣ Dependency Inversion Principle (DIP)

DTO depends on:

- Jackson annotations
- Jakarta validation annotations
- Lombok

That’s acceptable for a DTO layer.

If you wanted to be very strict about clean architecture, you might:

- Move this class to a web or adapter layer
- Keep domain model free of Jackson/Jakarta annotations

But that’s architectural — not necessarily required here.

✅ No real DIP violation.

🎯 Minor Design Improvements (If Applicable)

✅ 1. Add Range Validation (If Business Rules Exist)

If:

- discountPercent must be 0–100
- rentalDayCount must be > 0

Then you should use validation annotations:

```
@Min(1)
private Integer rentalDayCount;

@Min(0)
@Max(100)
private Integer discountPercent;
```

That keeps validation declarative and out of service logic.

✅ 2. Remove toString() (Optional)

Since you're using Lombok, you can replace manual toString() with:

```
@Value
@ToString
```

Or Lombok’s default behavior.

🏆 Final Verdict

Your class is already clean and SOLID-compliant.

✅ It has a single responsibility

✅ It does not violate OCP/LSP/ISP/DIP

✅ It properly separates transport data from business logic

✅ Recommended Improvements (Only If You Want Production-Level Clean Code)

1. Make it **immutable**
2. Add **range validation**
3. Remove manual toString() and use Lombok
4. Keep it strictly as a DTO (no business logic added later)

If you'd like, I can also:

- ✅ Review your Service layer for SOLID compliance
- ✅ Review your Domain model
- ✅ Show what a clean layered Toolstore architecture would look like

Just share the next class.

What exception is thrown by @min & @max ? I need to add it to my globalexception handler for clean error messages.

RouteLLM

When you use Jakarta Bean Validation annotations like @Min, @Max, or @NotNull on a request body in a Spring Boot controller, the exception thrown depends on **how** the validation is triggered.

1. The Primary Exception: MethodArgumentNotValidException

This is the most common exception. It is thrown when a @RequestBody parameter fails validation (i.e., you have @Valid or @Validated in your controller method signature).

How to handle it in your Global Exception Handler:

You should override or handle MethodArgumentNotValidException. This exception contains a BindingResult which holds the specific error messages you defined in your DTO (e.g., "Discount percent is required").

```
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<Map<String, String>>
handleValidationExceptions(MethodArgumentNotValidException ex) {
        Map<String, String> errors = new HashMap<>();

        ex.getBindingResult().getFieldErrors().forEach(error ->
            errors.put(error.getField(), error.getDefaultMessage())
        );

        return new ResponseEntity<>(errors, HttpStatus.BAD_REQUEST);
    }
}
```

---

2. The Secondary Exception: ConstraintViolationException

This is thrown in two specific scenarios:

1. When you validate **Request Parameters** or **Path Variables** (e.g., /checkout?discount=110) instead of a Request Body.
2. When you manually trigger validation on a service-layer method using @Validated at the class level.

How to handle it:

```
@ExceptionHandler(ConstraintViolationException.class)
public ResponseEntity<Map<String, String>>
handleConstraintViolationException(ConstraintViolationException ex) {
   Map<String, String> errors = new HashMap<>();

   ex.getConstraintViolations().forEach(violation -> {
      String propertyPath = violation.getPropertyPath().toString();
      errors.put(propertyPath.substring(propertyPath.lastIndexOf('.') + 1),
violation.getMessage());
   });

   return new ResponseEntity<>(errors, HttpStatus.BAD_REQUEST);
}
```

3. Summary Table for your Handler

Scenario	Exception Thrown
JSON Body (@RequestBody @Valid)	MethodArgumentNotValidException
URL Params (@RequestParam)	ConstraintViolationException
Path Variables (@PathVariable)	ConstraintViolationException
Invalid JSON Format (e.g., string instead of int)	HttpMessageNotReadableException

💡 Pro-Tip for Clean Messages

If the user sends a string where an Integer is expected (like discountPercent: "abc"), the validation annotations won't even be reached. Instead, HttpMessageNotReadableException is thrown. You might want to add a handler for that as well to avoid showing a messy stack trace to the user.

This is my global expectation handler ```@RestControllerAdvice

~~~
public class GlobalExceptionHandler {

    @ExceptionHandler(IllegalArgumentException.class)
    public ResponseEntity handleValidationException(IllegalArgumentException ex) {
        System.err.println("Checkout Validation Failed: " + ex.getMessage());
        return ResponseEntity.badRequest().body(new ErrorResponse(ex.getMessage()));
    }

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity handleMethodArgumentNotValid(MethodArgumentNotValidException ex) {
        String message = ex.getBindingResult()
                .getFieldErrors()
                .stream()
                .map(FieldError::getDefaultMessage)
                .collect(Collectors.joining(", "));

        System.err.println("Request Body Invalid: " + message);
        return ResponseEntity.badRequest().body(new ErrorResponse(message));
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity handleGeneralException(Exception ex) {
        System.err.println("An unexpected error occurred: " + ex.getMessage());
        ex.printStackTrace(); // Useful for a console-based demo
        return ResponseEntity.internalServerError().body(new ErrorResponse("An internal error occurred."));
    }
}```
~~~

This is a snippet from my checkout service

~~~
```private void validateRentalDayCount(int rentalDayCount) {
        if (rentalDayCount < 1) {
            throw new IllegalArgumentException("Rental day count must be 1 or greater");
        }
    }

    private void validateDiscountPercent(int discountPercent) {
        if (discountPercent < 0 || discountPercent > 100) {
~~~

```
            throw new IllegalArgumentException("Discount percent must be between 0 and 100");
        }
    }```
```

I want to format clean user friendly error messages along with a 400.

RouteLLM

Looking at your code, you actually **don't need to add anything new**. Here's why:

✅ You're Already Covered

Your service throws IllegalArgumentException, and your handler already catches it and returns a 400 Bad Request with the message directly:

```
@ExceptionHandler(IllegalArgumentException.class)
public ResponseEntity<ErrorResponse>
handleValidationException(IllegalArgumentException ex) {
    return ResponseEntity.badRequest().body(new ErrorResponse(ex.getMessage()));
}
```

So when validateDiscountPercent throws:

```
throw new IllegalArgumentException("Discount percent must be between 0 and 100");
```

The user gets back:

```
{
  "message": "Discount percent must be between 0 and 100"
}
```

Clean, user-friendly, and 400. ✅

💡 One Recommendation Though

Since you're doing **manual validation in the service layer**, you can actually **remove** @Min/@Max from the DTO to avoid duplicate validation logic. Pick **one** approach:

Approach	**Where Validation Lives**	**Best For**
Jakarta annotations (@Min, @Max)	DTO layer	Simple constraint checks at the API boundary
Manual validation (your current approach)	Service layer	Business rule validation with custom logic

Since your service already validates with meaningful messages, the DTO annotations would be redundant. Your current approach is clean and consistent — just keep it as-is.

I believe Jakarta @min & @max are throwing different exceptions.
This test

```
@Test
    void checkout_RentalDaysLessThanOne_Returns400WithMessage() throws Exception {
        // Specification test 2 - modified to be invalid
        CheckoutRequest request = new CheckoutRequest("LADW", 0, 10, LocalDate.of(2020, 7, 2));

        when(checkoutService.checkout(anyString(), anyInt(), anyInt(), any(LocalDate.class)))
                .thenThrow(new IllegalArgumentException("Rental day count must be 1 or greater"));

        mockMvc.perform(post("/api/checkout")
                        .contentType(MediaType.APPLICATION_JSON)
                        .content(objectMapper.writeValueAsString(request)))
                .andExpect(status().isBadRequest())
                .andExpect(jsonPath("$.message").value("Rental day count must be 1 or greater"));
    }
```

Returns this assert error.
Request Body Invalid: must be greater than or equal to 1

```
JSON path "$.message"
Expected :Rental day count must be 1 or greater
Actual   :must be greater than or equal to 1
```

RouteLLM

You are exactly right. The reason your test is failing is because **Jakarta Bean Validation (@Min) is intercepting the request before it ever reaches your service.**

Here is what is happening:

1. **The Interception:** Because you have @Min(1) on your CheckoutRequest DTO and @Valid in your Controller, Spring validates the object immediately upon deserialization.
2. **The Exception:** It throws MethodArgumentNotValidException.
3. **The Handler:** Your handleMethodArgumentNotValid method catches it.
4. **The Message:** Since you didn't provide a custom message in the @Min annotation, it uses the default Jakarta message: "must be greater than or equal to 1".
5. **The Service:** Your service (and your when(...).thenThrow(...) mock) is **never even called** because the request was rejected at the Controller door.

How to fix it for clean, consistent messages

To make the Jakarta validation match your service logic and pass your test, you should define the message explicitly in the DTO.

1. Update your DTO

Add the message attribute to your @Min and @Max annotations to match the exact strings your test expects.

```
public class CheckoutRequest {

    @NotNull(message = "Tool code is required")
    private String toolCode;

    @NotNull(message = "Rental day count is required")
    @Min(value = 1, message = "Rental day count must be 1 or greater") // Match your test string
    private Integer rentalDayCount;
    @NotNull(message = "Discount percent is required")
    @Min(value = 0, message = "Discount percent must be between 0 and 100")
    @Max(value = 100, message = "Discount percent must be between 0 and 100")
    private Integer discountPercent;

    // ... rest of class
}
```

2. Why this is better for SOLID

By putting these messages in the DTO:

- **Fail Fast:** You reject bad data at the edge of your application (Controller) before wasting resources in the Service or Database layers.
- **Single Source of Truth:** You can actually **remove** the manual validateRentalDayCount methods from your Service. The DTO handles the "shape" and "constraints" of the data, while the Service handles "business logic" (like checking if the tool is actually in stock).

If you want to keep the Service validation

If you prefer the Service to do the validation (perhaps because the rules are dynamic), you should **remove** @Min and @Max from the DTO entirely.

If you remove them:

1. Spring won't catch the error at the Controller.
2. The request will reach the Service.
3. Your Mockito when(...).thenThrow(...) will trigger.
4. Your IllegalArgumentException handler will catch it.
5. The test will pass with your specific message.

Recommendation: Use the **DTO approach (Step 1)**. It is the standard "Spring Way" and keeps your service code much leaner.
