Case Study: Date and Time Handling

Leave Tracking System

Estimated time: 15 minutes

Let's explore some real-life applications of date and time handling related to this case study.

Date and time handling is crucial in real-world applications for scheduling, record-keeping, and coordinating activities. Calendar applications, appointment booking systems, and flight reservations all rely on date-time operations.

In your case study, your Leave Tracking System needs robust date-time handling to:

- Record when leave requests start and end
- Calculate leave durations
- Track approval timestamps
- Manage leave balances over time

```
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.Period;
```

You'll also need to complete the following date and time handling practical tasks:

- Import the necessary date and time classes
- Update the LeaveRequest class to use proper date types
- Replace string dates with LocalDate objects

Displaying Current Date and Time

In real life, applications often need to show the current date and time. Digital clocks, weather apps, and logging systems all display current timestamps.

In this case study, you will use the current date and time to track when employees create leave requests. Take a moment to review the following code:

```
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
public class LeaveRequest {
    // Other fields
    private LocalDateTime createdAt;

    public LeaveRequest(int requestId, Employee employee,
                        LocalDate startDate, LocalDate endDate, String reason) {
        // Initialize other fields
        // ...

        // Record creation time
        this.createdAt = LocalDateTime.now();
    }

    public String getCreationTimestamp() {
        DateTimeFormatter formatter =
            DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
        return createdAt.format(formatter);
    }
}
```

You'll also need to complete the following practical tasks related to displaying the current date and time.

- Add timestamps to track when leave requests are created
- Display the creation time in a user-friendly format
- Record when requests are approved or denied

Formatting Dates in Java

Different regions and organizations require specific date formats. For example, international websites adapt their displayed date formats to learners in different countries. You also might see that an organization's financial reports use specific date presentations based on local requirements for clients and regulatory reporting.

In this case study, you will need to format dates for different display needs. Consider the following code sample:

```
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
public class LeaveRequestFormatter {
    // Format for display in the UI
    public static String formatForDisplay(LocalDate date) {
        DateTimeFormatter formatter =
```

```
            DateTimeFormatter.ofPattern("MMMM d, yyyy"); // e.g., "April 15, 2023"
        return date.format(formatter);
    }

    // Format for database storage
    public static String formatForStorage(LocalDate date) {
        return date.format(DateTimeFormatter.ISO_LOCAL_DATE); // e.g., "2023-04-15"
    }

    // Format for reports
    public static String formatForReport(LocalDate date) {
        DateTimeFormatter formatter =
            DateTimeFormatter.ofPattern("dd/MM/yyyy"); // e.g., "15/04/2023"
        return date.format(formatter);
    }
}
```

As part of the date and time formatting process, you will need to complete the following practical tasks.

- Create methods to format dates for different contexts
- Use appropriate date formats for user interfaces
- Implement standard date formats for data storage

Working with LocalDate and LocalTime

Modern applications can use dates and times separately and often use them together. For example, project management tools track task due dates, time-tracking apps record work hours, and scheduling applications handle both dates and times.

In this case study, you will update the LeaveRequest class to use LocalDate properly. Review the following code:

```
import java.time.LocalDate;
public class LeaveRequest {
    private int requestId;
    private Employee employee;
    private LocalDate startDate;
    private LocalDate endDate;
    private String status;
    private String reason;

    public LeaveRequest(int requestId, Employee employee,
                        LocalDate startDate, LocalDate endDate, String reason) {
        this.requestId = requestId;
        this.employee = employee;
        this.startDate = startDate;
        this.endDate = endDate;
        this.status = "Pending";
        this.reason = reason;
    }

    // Calculate the number of days of leave
    public int getLeaveDuration() {
        // Using Period to calculate days between dates
        return java.time.Period.between(startDate, endDate).getDays() + 1;
    }
}
```

As part of working with LocalDate and LocalTime, you will need to complete the following practical tasks:

- Replace string date representations with LocalDate objects
- Implement methods to calculate the duration of leave requests
- Add validation to ensure end dates are not before start dates

Understanding Time Zones

Time zones are an integral part of real-life applications. Global applications, including airlines and logistics organizations, implement time zone calculations as an integral part of their business. International flight booking systems display departure times in local time zones, virtual meeting platforms convert event times to participants' local times, and global teams coordinate staff coverage across different time zones.

In this case study, you will implement time zone handling for leave requests within a company with offices in multiple countries. Review the following code sample as you formulate your plan to handle time zones.

```
import java.time.ZoneId;
import java.time.ZonedDateTime;
import java.time.format.DateTimeFormatter;
public class LeaveTrackingSystem {
    // Other fields and methods
    // ...

    public String getLocalApprovalTime(LeaveRequest request, String timeZone) {
        // Get the approval timestamp (stored in UTC)
```

```
        ZonedDateTime utcTime = request.getApprovalTimestamp();

        // Convert to the requested time zone
        ZonedDateTime localTime = utcTime.withZoneSameInstant(ZoneId.of(timeZone));

        // Format the time
        DateTimeFormatter formatter =
            DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss z");
        return localTime.format(formatter);
    }
}
```

You will also need to complete the following practical tasks to handle time zones:

- Store timestamps in a standard time zone (UTC)
- Convert timestamps to users' local time zones for display
- Add support for employees in different time zones

Calculating Differences Between Two Dates

Date calculations are essential for many day-to-day applications. Project management tools calculate deadlines, billing systems determine subscription periods, and human resource (HR) systems compute employee leave balances.

In this case study, you will calculate leave balances and consumption. Review the following code:

```
import java.time.LocalDate;
import java.time.Period;
import java.time.temporal.ChronoUnit;
public class LeaveCalculator {
    // Calculate business days between two dates (excluding weekends)
    public static int calculateBusinessDays(LocalDate startDate, LocalDate endDate) {
        int businessDays = 0;
        LocalDate currentDate = startDate;

        while (!currentDate.isAfter(endDate)) {
            // Check if it's not a weekend (Saturday or Sunday)
            if (currentDate.getDayOfWeek().getValue() < 6) {
                businessDays++;
            }
            currentDate = currentDate.plusDays(1);
        }

        return businessDays;
    }

    // Calculate days until leave expires
    public static long daysUntilExpiration(LocalDate leaveDate, int expirationDays) {
        LocalDate expirationDate = leaveDate.plusDays(expirationDays);
        return ChronoUnit.DAYS.between(LocalDate.now(), expirationDate);
    }
}
```

As part of calculating differences between two dates, you will need to complete the following practical tasks:

- Create methods to calculate business days between dates
- Implement leave expiration calculations
- Create reports showing leave usage over time periods

Parsing Dates from Strings

Applications often need to convert user input from sentences and other string content into date objects. Form submissions, data imports, and search filters all require date parsing.

In this case study, you will need to parse dates from user input and file imports. Reflect on the following code:

```
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.time.format.DateTimeParseException;
public class DateParser {
    // Parse a date with validation
    public static LocalDate parseInputDate(String dateString) {
        try {
            DateTimeFormatter formatter = DateTimeFormatter.ofPattern("MM/dd/yyyy");
            return LocalDate.parse(dateString, formatter);
        } catch (DateTimeParseException e) {
            System.out.println("Invalid date format. Please use MM/DD/YYYY.");
            return null;
        }
    }

    // Parse dates from CSV imports
```

```
    public static LocalDate parseFromCsv(String dateString) {
        try {
            // Try different common formats
            String[] formats = {"yyyy-MM-dd", "MM/dd/yyyy", "dd-MM-yyyy"};

            for (String format : formats) {
                try {
                    DateTimeFormatter formatter = DateTimeFormatter.ofPattern(format);
                    return LocalDate.parse(dateString, formatter);
                } catch (DateTimeParseException e) {
                    // Try next format
                }
            }

            throw new DateTimeParseException(
                "Could not parse date in any supported format", dateString, 0);
        } catch (DateTimeParseException e) {
            System.out.println("Error parsing date: " + dateString);
            return null;
        }
    }
}
```

To parse dates from strings you'll need to complete the following practical tasks:

- Create methods to safely parse dates from different input formats
- Implement validation to handle parsing errors
- Add support for different regional date formats

The real-life application of date and time handling is critical for accurate record-keeping and calculations. HR systems use date calculations for leave entitlements, accruals, and reporting periods.

What you can do in real life

In real life, consider the following actions when handling dates and times:

- Use appropriate date and time classes based on the application's specific needs
- Format dates according to user preferences, regional standards, and regulatory requirements
- Perform accurate calculations for differences between dates
- Handle time zones correctly when working with global applications
- Implement code that safely parses dates from user input using proper validation

Author(s)

Ramanujam Srinivasan