Java Date & Time API

Objectives

After completing this lesson, you should be able to:

- Create and manage date-based events
- Create and manage time-based events
- Combine date and time into a single object
- Work with dates and times across time zones
- Manage changes resulting from daylight savings
- Define and create timestamps, periods, and durations

Why Is Date and Time Important?

In the development of applications, programmers often need to represent time and use it to perform calculations:

- The current date and time (locally)
- A date and/or time in the future or past
- The difference between two dates/time in seconds, minutes, hours, days, months, years
- The time or date in another country (time zone)
- The correct time after daylight savings time is applied
- The number of days in the month of February (leap years)
- A time duration (hours, mins, secs) or a period (years, months, days)

Previous Java Date and Time

Disadvantages of `java.util.Date` (`Calendar`, `TimeZone` & `DateFormat`):

- Instances are mutable – not compatible with lambda
- Not thread-safe
- Weakly typed calendars
- One size fits all

Java Date and Time API: Goals

- The classes and methods should be straightforward.
- Instances of time/date objects should be immutable. (This is important for lambda operations.)
- Use ISO standards to define date and time.
- Time and date operations should be thread-safe.
- The API should support strong typing, which makes it much easier to develop good code first. (The compiler is your friend!)
- `toString` will always return a human-readable format.
- Allow developers to extend the API easily.

Working with Local Date and Time

The `java.time` API defines two classes for working with local dates and times (without a time zone):

- `LocalDate`:
  - Does not include time
  - A year-month-day representation
  - `toString` – ISO 8601 format (YYYY-MM-DD)
- `LocalTime`:
  - Does not include date
  - Stores hours:minutes:seconds.nanoseconds
  - `toString` – (HH:mm:ss.SSSS)

Working with `LocalDate`

`LocalDate` is a class that holds an event date: a birth date, anniversary, meeting date, and so on.

- A date is a label for a day.
- `LocalDate` uses the ISO calendar by default.
- `LocalDate` does not include time, so it is portable across time zones.
- You can answer the following questions about dates with `LocalDate`:
  - Is it in the future or past?
  - Is it in a leap year?
  - What day of the week is it?
  - What is the day a month from now?
  - What is the date next Tuesday?

LocalDate: Example

```
import java.time.LocalDate;
import static java.time.temporal.TemporalAdjusters.*;
import static java.time.DayOfWeek.*;
import static java.lang.System.out;

public class LocalDateExample {

  public static void main(String[] args) {
    LocalDate now, bDate, nowPlusMonth, nextTues;
    now = LocalDate.now();
    out.println("Now: " + now);
    bDate = LocalDate.of(1995, 5, 23); // Java's Birthday
    out.println("Java's Bday: " + bDate);
    out.println("Is Java's Bday in the past? " + bDate.isBefore(now));
    out.println("Is Java's Bday in a leap year? " + bDate.isLeapYear());
    out.println("Java's Bday day of the week: " + bDate.getDayOfWeek());
    nowPlusMonth = now.plusMonths(1);
    out.println("The date a month from now: " + nowPlusMonth);
    nextTues = now.with(next(TUESDAY));
    out.println("Next Tuesday's date: " + nextTues);
  }
}
```

`next` method

`TUESDAY`

`LocalDate` objects are immutable – methods return a new instance.

Working with `LocalTime`

`LocalTime` stores the time within a day.

- Measured from midnight
- Based on a 24-hour clock (13:30 is 1:30 PM.)
- Questions you can answer about time with `LocalTime`
  - When is my lunch time?
  - Is lunch time in the future or past?
  - What is the time 1 hour 15 minutes from now?
  - How many minutes until lunch time?
  - How many hours until bedtime?
  - How do I keep track of just the hours and minutes?

LocalTime: Example

```
import java.time.LocalTime;
import static java.time.temporal.ChronoUnit.*;
import static java.lang.System.out;

public class LocalTimeExample {
  public static void main(String[] args) {
    LocalTime now, nowPlus, nowHrsMins, lunch, bedtime;
    now = LocalTime.now();
    out.println("The time now is: " + now);
    nowPlus = now.plusHours(1).plusMinutes(15);
    out.println("What time is it 1 hour 15 minutes from now? " + nowPlus);
    nowHrsMins = now.truncatedTo(MINUTES);
    out.println("Truncate the current time to minutes: " + nowHrsMins);
    out.println("It is the " + now.toSecondOfDay()/60 + "th minute");
    lunch = LocalTime.of(12, 30);
    out.println("Is lunch in my future? " + lunch.isAfter(now));
    long minsToLunch = now.until(lunch, MINUTES);
    out.println("Minutes til lunch: " + minsToLunch);
    bedtime = LocalTime.of(21, 0);
    long hrsToBedtime = now.until(bedtime, HOURS);
    out.println("How many hours until bedtime? " + hrsToBedtime);
  }
}
```

HOURS, MINUTES

Working with `LocalDateTime`

`LocalDateTime` is a combination of `LocalDate` and `LocalTime`.

- `LocalDateTime` is useful for narrowing events.
- You can answer the following questions with `LocalDateTime`:
  - When is the meeting with corporate?
  - When does my flight leave?
  - When does the course start?
  - If I move the meeting to Friday, what is the date?
  - If the course starts at 9 AM on Monday and ends at 5 PM on Friday, how many hours am I in class?

LocalTimeDate: Example

```
import java.time.*;
import static java.time.Month.*;
import static java.time.temporal.ChronoUnit.*;
import static java.lang.System.out;

public class LocalDateTimeExample {
  public static void main(String[] args) {
    LocalDateTime meeting, flight, courseStart, courseEnd;
    meeting = LocalDateTime.of(2014, MARCH, 21, 13, 30);
    out.println("Meeting is on: " + meeting);
    LocalDate flightDate = LocalDate.of(2014, MARCH, 31);
    LocalTime flightTime = LocalTime.of(21, 45);
    flight = LocalDateTime.of(flightDate, flightTime);
    out.println("Flight leaves: " + flight);
    courseStart = LocalDateTime.of(2014, MARCH, 24, 9, 00);
    courseEnd = courseStart.plusDays(4).plusHours(8);
    out.println("Course starts: " + courseStart);
    out.println("Course ends:   " + courseEnd);
    long courseHrs = (courseEnd.getHour() - courseStart.getHour()) *
                     (courseStart.until(courseEnd, DAYS) + 1);
    out.println("Course is:     " + courseHrs + " hours long.");
  }
}
```

LocalDateTime, LocalDate, LocalTime

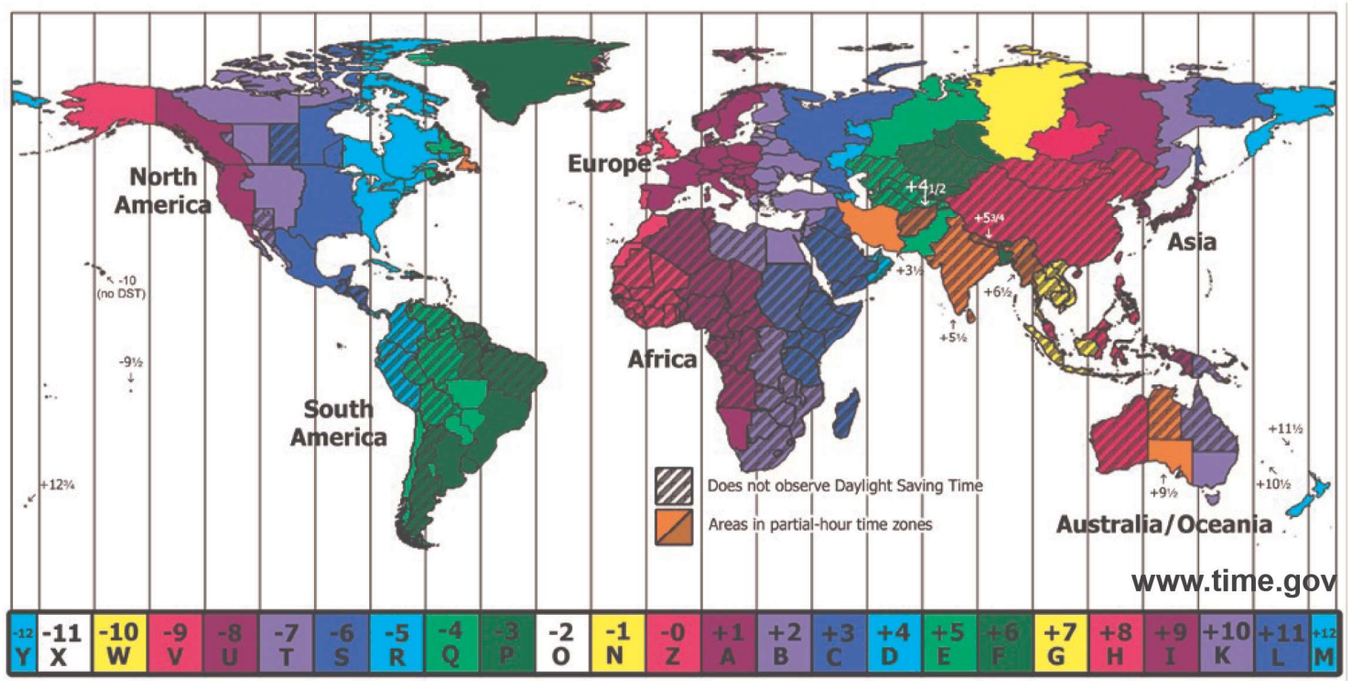
MARCH

Combine LocalDate and LocalTime objects.

Working with Time Zones

Time zones are geographic, but the time in a specific location is defined by the government in that location.

- When a country (and sometimes a state) observes changes (for daylight savings) varies.



Modeling Time Zones

- `ZoneId`: Is a specific location or offset relative to UTC

```
ZoneId nyTZ = ZoneId.of("America/New_York");
ZoneId EST = ZoneId.of("US/Eastern");
ZoneId Romeo = ZoneId.of("Europe/London");
```

- `ZoneOffset`: Extends `ZoneId`; specifies the actual time difference from UTC

```
ZoneOffset USEast = ZoneOffset.of("-5");
ZoneOffset Nepal = ZoneOffset.ofHoursMinutes(5, 45);
ZoneId EST = ZoneId.ofOffset("UTC", USEast);
```

Working with `ZonedDateTime` Gaps/Overlaps

Given a meeting date the day before daylight savings (2AM on March 9th), what happens if the meeting is moved out by a day?

```
// DST Begins March 9th, 2014
LocalDate meetDate = LocalDate.of(2014, MARCH, 8);
LocalTime meetTime = LocalTime.of(16, 00);
ZonedDateTime meeting = ZonedDateTime.of(meetDate, meetTime, USEast);
System.out.println("meeting time:     " + meeting);
ZonedDateTime newMeeting = meeting.plusDays(1);
System.out.println("new meeting time: " + newMeeting
```

```
meeting time:     2014-03-08 16:00 -05:00[America/New_York]
new meeting time: 2014-03-09 16:00 -04:00[America/New_York]
```

- The local time is not changed, and the offset is managed correctly.

Date and Time Methods

Prefix	Example	Use
`now`	`today = LocalDate.now()`	Creates an instance using the system clock
`of`	`meet = LocalTime.of(13, 30)`	Creates an instance by using the parameters passed
`get`	`today.get(DAY_OF_WEEK)`	Returns part of the state of the target
`with`	`meet.withHour(12)`	Returns a copy of the target object with one element changed
`plus, minus`	`nextWeek.plusDays(7)` `sooner.minusMinutes(30)`	Returns a copy of the object with the amount added or subtracted
`to`	`meet.toSecondOfDay()`	Converts this object to another type. Here returns `int` seconds.
`at`	`today.atTime(13, 30)`	Combines this object with another; returns a `LocalDateTime` object
`until`	`today.until`	Calculates the amount of time until another date in terms of the unit
`isBefore, isAfter`	`today.isBefore(lastWeek)`	Compares this object with another on the timeline
`isLeapYear`	`today.isLeapYear()`	Checks if this object is a leap year

`DateTimeFormatter` produces formatted date/times

- Using predefined constants, patterns letters, or a localized style

```
ZonedDateTime now = ZonedDateTime.now();
DateTimeFormatter formatter = DateTimeFormatter.ISO_LOCAL_DATE;
System.out.println(now.format(formatter));
formatter = DateTimeFormatter.ISO_ORDINAL_DATE;
System.out.println(now.format(formatter));
formatter = DateTimeFormatter.ofPattern("EEEE, MMMM dd, yyyy G, hh:mm a VV");
System.out.println(now.format(formatter));
formatter = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.MEDIUM);
System.out.println(now.format(formatter));
```

Predefined `DateTimeFormatter` constants

String pattern

Format style

```
2014-02-21
2014-052-05:00
Friday, February 21, 2014 AD, 03:51 PM America/New_York
Feb 21, 2014 3:51:51 PM
```

Year and day of the year

`FormatStyle.MEDIUM`

Using Fluent Notation

One of the goals of JSR-310 was to make the API fluent.

- Examples:

```
// Not very readable - is this June 11 or November 6th?
LocalDate myBday = LocalDate.of(1970, 6, 11);
// A fluent approach
myBday = Year.of(1970).atMonth(JUNE).atDay(11);

// Schedule a meeting fluently
LocalDateTime meeting = LocalDate.of(2014, MARCH, 25).atTime(12, 30);

// Schedule that meeting using the London timezone
ZonedDateTime meetingUK = meeting.atZone(ZoneId.of("Europe/London"));

// What time is it in San Francisco for that meeting?
ZonedDateTime earlyMeeting =
  meetingUK.withZoneSameInstant(ZoneId.of("America/Los_Angeles"));
```

Summary

In this lesson, you should have learned how to:

- Create and manage date-based events and manage time-based events
- Combine date and time into a single object
- Work with dates and times across time zones
- Manage changes resulting from daylight savings
- Define and create timestamps, periods and durations
- Apply formatting to local and zoned dates and times

Practice : Overview

- This practice covers the following topics:
  - Using SQL Developer
  - Selecting all data from different tables
  - Describing the structure of tables
  - Performing arithmetic calculations and specifying column names