

# 12

## 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
- Apply formatting to local and zoned dates and times

# 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):

- Does not support fluent API approach
- 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.
- The API should support a fluent API approach.
- 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

next method

```
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);
    }
}
```

TUESDAY

LocalDate objects are immutable – methods return a new instance.



# Working with LocalTime

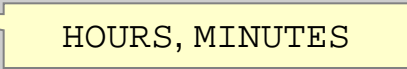
LocalTime stores the time within a day.

- 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);
    }
}
```



# 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?

# LocalDateTime: 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

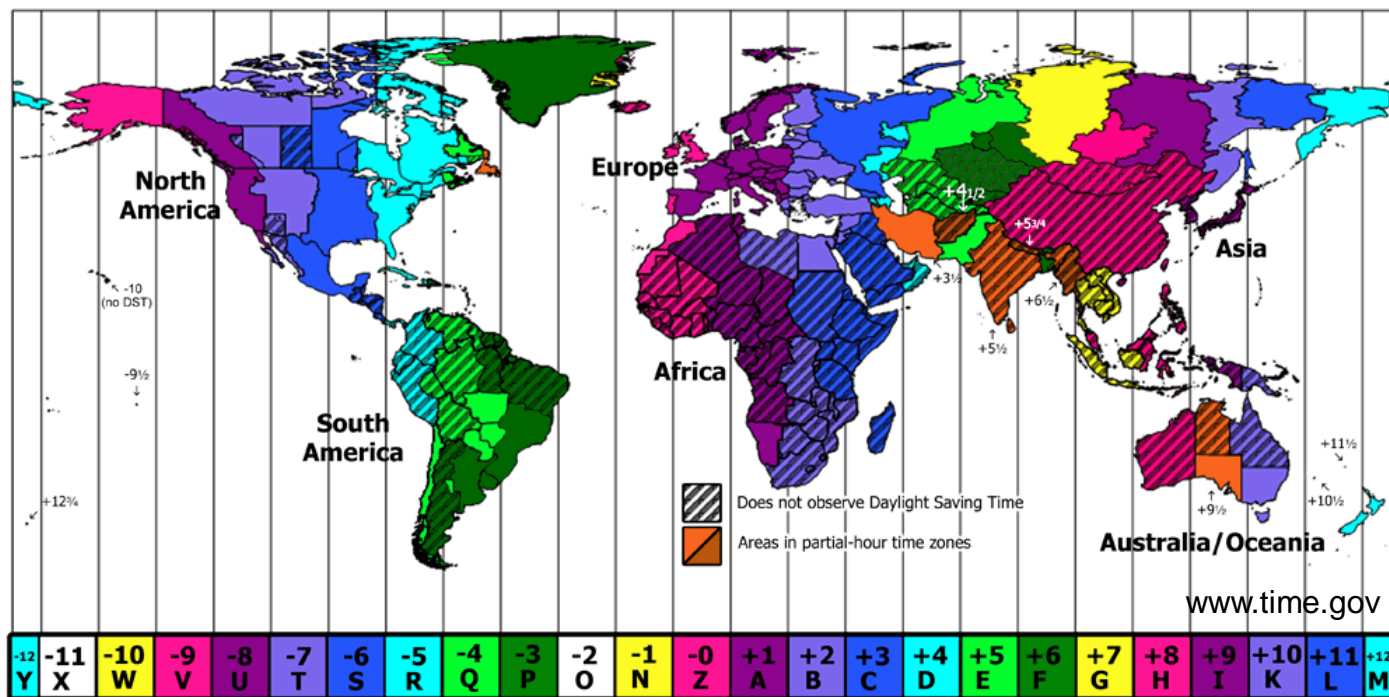
MARCH

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.



# Daylight Savings Time Rules

Time changes result in a local hour gap/overlap:

| Sunday, March 9, 2014<br>(New York) | Local time         | UTC Offset |
|-------------------------------------|--------------------|------------|
|                                     | 1:59:58 AM         | UTC-5h EST |
|                                     | 1:59:59 AM         | UTC-5h EST |
| Starting DST causes a one hour gap. | 2:00:00 -> 3:00:00 | UTC-4h EDT |
|                                     | 3:00:01 AM         | UTC-4h EDT |

| Sunday, November 2, 2014<br>(New York) | Local time         | UTC Offset |
|--|--------------------|------------|
|  | 1:59:58 AM         | UTC-4h EST |
|  | 1:59:59 AM         | UTC-4h EST |
| Ending DST causes a one hour overlap.  | 2:00:00 -> 1:00:00 | UTC-5h EDT |
|  | 1:00:01 AM         | UTC-5h EDT |

# 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);
```

- **ZoneRules:** Is the class used to determine offsets

# Creating ZonedDateTime Objects

- Stores LocalDateTime, ZoneId, and ZoneOffset

```
ZoneId USEast = ZoneId.of("America/New_York");
LocalDate date = LocalDate.of(2014, MARCH, 23);
LocalTime time = LocalTime.of(9, 30);
LocalDateTime dateTime = LocalDateTime.of(date, time);
ZonedDateTime courseStart = ZonedDateTime.of(date, time, USEast);
ZonedDateTime hereNow = ZonedDateTime.now(USEast).truncatedTo(MINUTES);
System.out.println("Here now:           " + hereNow);
System.out.println("Course start:        " + courseStart);
ZonedDateTime newCourseStart = courseStart.plusDays(2).minusMinutes(30);
System.out.println("New Course Start: " + newCourseStart);
```

```
Here now:           2014-02-19 T 17:00 -05:00[America/New_York]
Course start:       2014-03-23 T 09:30 -04:00[America/New_York]
New Course Start:  2014-03-25 T 09:00 -04:00[America/New_York]
```

Space added to make the  
fields more clear



# Working with ZonedDateTime Gaps/Overlaps

Given a meeting date the day before daylight savings (2AM on March 9<sup>th</sup>), 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.

# ZoneRules

- Each time zone (`ZoneId`) has a set of rules that are part of the JDK.
- Date or times that land on time changes can be determined by using the rules.

```
// Ask the rules if there was a gap or overlap
ZoneId USEast = ZoneId.of("America/New_York");
LocalDateTime lateNight = LocalDateTime.of(2014, MARCH, 9, 2, 30);
ZoneOffsetTransition zot = USEast.getRules().getTransition(lateNight);
if (zot != null) {
    if (zot.isGap()) System.out.println("gap");
    if (zot.isOverlap()) System.out.println("overlap");
}
```

- Given the code above, what will print?

# Working Across Time Zones

The `OffsetDateTime` class stores a `LocalDateTime` and `ZoneOffset`.

- This is useful for determining `ZonedDateTime`s across time zones.

```
LocalDateTime meeting = LocalDateTime.of(2014, JUNE, 13, 12, 30);
ZoneId SanFran = ZoneId.of("America/Los_Angeles");
ZonedDateTime staffCall = ZonedDateTime.of(meeting, SanFran);
OffsetDateTime = staffCall.toOffsetDateTime();
```

- The offset is used to calculate date/time using zone rules:

```
ZoneId London = ZoneId.of("Europe/London");
OffsetDateTime staffCallOffset = staffCall.toOffsetDateTime();
ZonedDateTime staffCallUK = staffCallOffset.atZoneSameInstant(London);
System.out.println("Staff call (Pacific) is at: " + staffCall);
System.out.println("Staff call (UK) is at:      " + staffCallLondon);
```

# Date and Time Methods

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

# Date and Time Amounts

- `Instant` – Stores an instant in time on the time-line
  - Useful for: timestamps, e.g. login events
  - Stored as seconds (`long`) and nanoseconds (`int`)
  - Methods used to compare before and after

```
Instant now = Instant.now();
Thread.sleep(0,1); // long milliseconds, int nanoseconds
Instant later = Instant.now();
System.out.println("now is before later? " + now.isBefore(later));
System.out.println("Now:      " + now);
System.out.println("Later:   " + later);
```

```
now is before later? true
Now:      2014-02-21 T 16:11:34.788 Z
Later:    2014-02-21 T 16:11:34.789 Z
```

`toString` includes  
nanoseconds to three digits

# Period

Period is a class that holds a date-based amount.

- Years, months, and days based on the ISO-8601 calendar
- Plus and minus work with a conceptual day, thus preserving daylight savings changes

```
Period oneDay = Period.ofDays(1);
System.out.println("Period of one day: " + oneDay);
LocalDateTime beforeDST = LocalDateTime.of(2014, MARCH, 8, 12, 00);
ZonedDateTime newYorkTime =
    ZonedDateTime.of(beforeDST, ZoneId.of("America/New_York"));
System.out.println("Before: " + newYorkTime);
System.out.println("After:  " + newYorkTime.plus(oneDayYear));
```

The time is preserved, because  
only "days" are added.

```
Period of one day: P1D
Before: 2014-03-08 T 12:00 -05:00[America/New_York]
After:  2014-03-09 T 12:00 -04:00[America/New_York]
```

# Duration

`Duration` is a class that stores a time-based amount.

- Time is measured in actual seconds and nanoseconds.
- Days are treated as 24 hours, and daylight savings is ignored.

```
Duration one24hourDay = Duration.ofDays(1);
System.out.println("Duration of one day: " + one24hourDay);
beforeDST = LocalDateTime.of(2014, MARCH, 8, 12, 00);
newYorkTime = ZonedDateTime.of(beforeDST, ZoneId.of("America/New_York"));
System.out.println("Before: " + newYorkTime);
System.out.println("After:  " + newYorkTime.plus(one24hourDay));
```

The time is not preserved because  
24 hours are added.

```
Duration of one day: PT24H
Before: 2014-03-08 T 12:00 -05:00[America/New_York]
After:  2014-03-09 T 13:00 -04:00[America/New_York]
```

# Calculating Between Days

TemporalUnit is an interface representing a unit of time.

- Implemented by the enum class ChronoUnit

```
import static java.time.temporal.ChronoUnit.*;

LocalDate christmas = LocalDate.of(2014, DECEMBER, 25);
LocalDate today = LocalDate.now();
long days = DAYS.between(today, christmas);
System.out.println("There are " + days + " shopping days til Christmas");
```

- Period also provides a between method

```
Period tilXMas = Period.between(today, christmas);
System.out.println("There are " + tilXMas.getMonths() +
                  " months and " + tilXMas.getDays() +
                  " days til Christmas");
```



# Making Dates Pretty

`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
- 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
- Apply formatting to local and zoned dates and times

# Practices

- Practice 12-1: Working with Local Dates and Times
- Practice 12-2: Working with Dates and Times Across Time Zones
- Practice 12-3: Formatting Dates