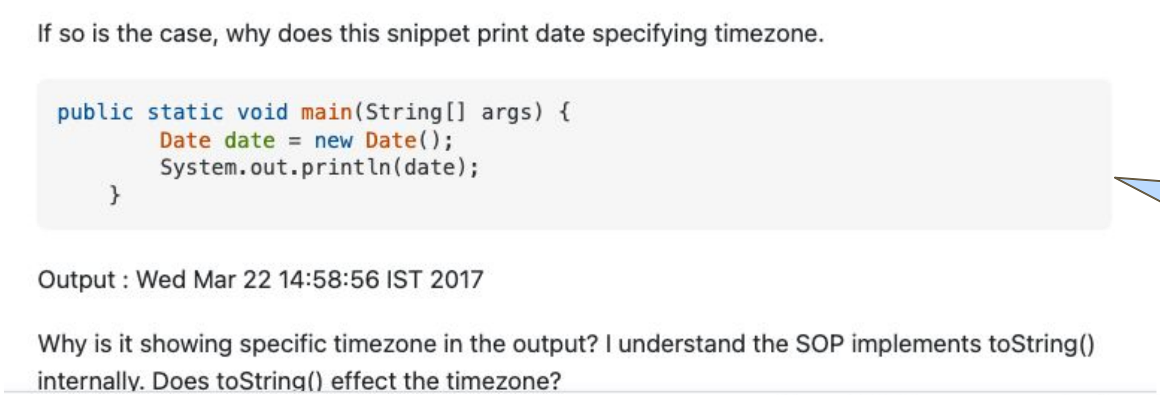
Java 8 Datetime API

Microlearning, 4 Sep 2023

Why do we need Java 8 Datetime API?



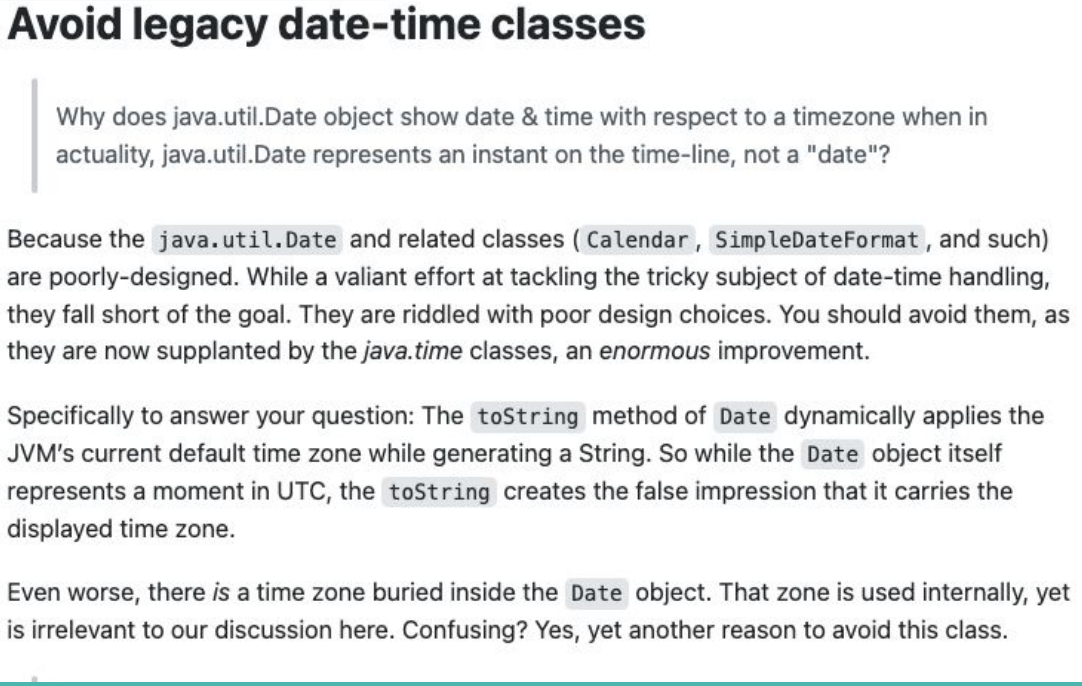
If so is the case, why does this snippet print date specifying timezone.

```
public static void main(String[] args) {
        Date date = new Date();
        System.out.println(date);
    }
```

Output : Wed Mar 22 14:58:56 IST 2017

Why is it showing specific timezone in the output? I understand the SOP implements toString() internally. Does toString() effect the timezone?

new Date() represents a moment in **UTC**, why does .toString() print out **my timezone information**?



Avoid legacy date-time classes

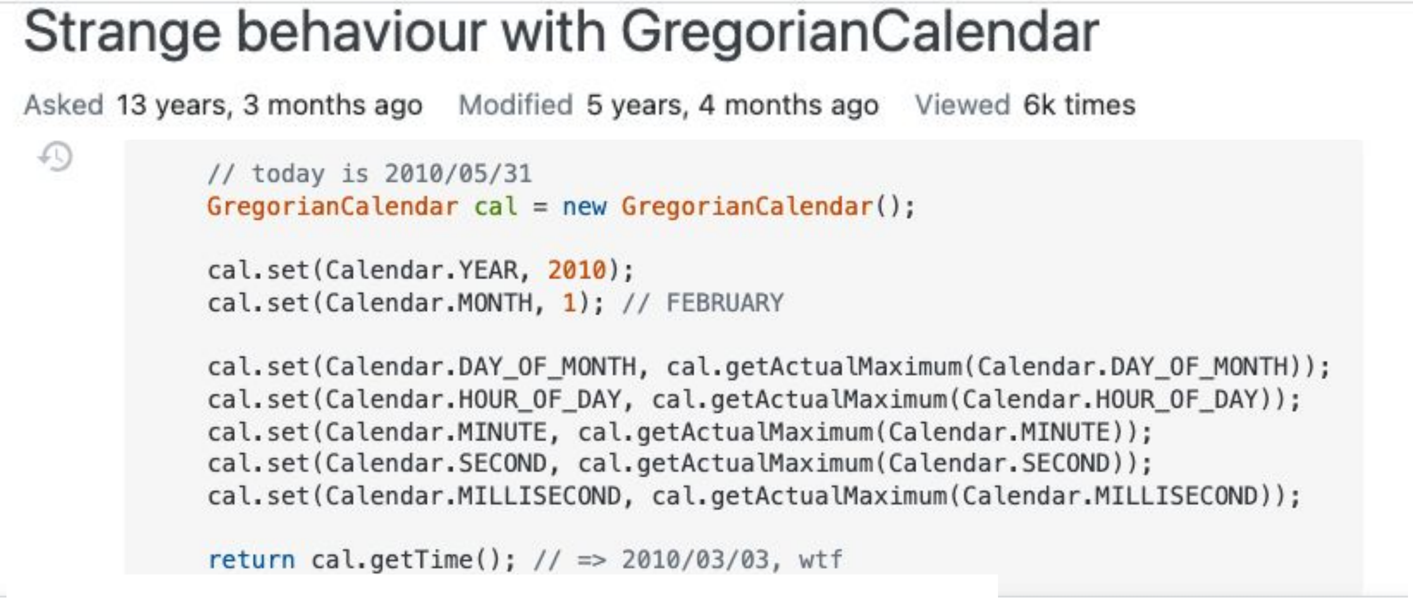
> Why does java.util.Date object show date & time with respect to a timezone when in actuality, java.util.Date represents an instant on the time-line, not a "date"?

Because the `java.util.Date` and related classes (`Calendar`, `SimpleDateFormat`, and such) are poorly-designed. While a valiant effort at tackling the tricky subject of date-time handling, they fall short of the goal. They are riddled with poor design choices. You should avoid them, as they are now supplanted by the *java.time* classes, an *enormous* improvement.

Specifically to answer your question: The `toString` method of `Date` dynamically applies the JVM's current default time zone while generating a String. So while the `Date` object itself represents a moment in UTC, the `toString` creates the false impression that it carries the displayed time zone.

Even worse, there *is* a time zone buried inside the `Date` object. That zone is used internally, yet is irrelevant to our discussion here. Confusing? Yes, yet another reason to avoid this class.

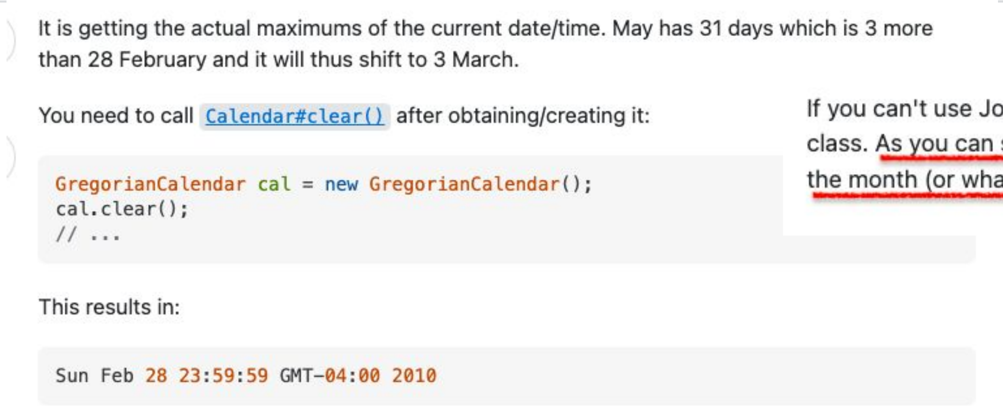
https://stackoverflow.com/a/48739458



Strange behaviour with GregorianCalendar

Asked 13 years, 3 months ago Modified 5 years, 4 months ago Viewed 6k times

```
// today is 2010/05/31
GregorianCalendar cal = new GregorianCalendar();

cal.set(Calendar.YEAR, 2010);
cal.set(Calendar.MONTH, 1); // FEBRUARY

cal.set(Calendar.DAY_OF_MONTH, cal.getActualMaximum(Calendar.DAY_OF_MONTH));
cal.set(Calendar.HOUR_OF_DAY, cal.getActualMaximum(Calendar.HOUR_OF_DAY));
cal.set(Calendar.MINUTE, cal.getActualMaximum(Calendar.MINUTE));
cal.set(Calendar.SECOND, cal.getActualMaximum(Calendar.SECOND));
cal.set(Calendar.MILLISECOND, cal.getActualMaximum(Calendar.MILLISECOND));

return cal.getTime(); // => 2010/03/03, wtf
```



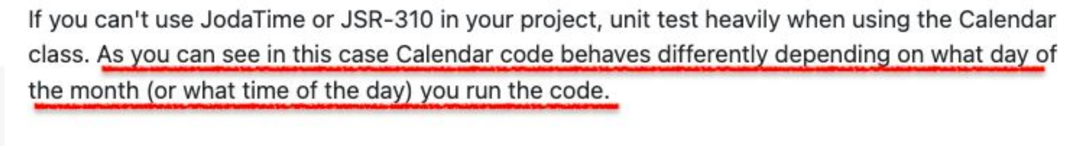
It is getting the actual maximums of the current date/time. May has 31 days which is 3 more than 28 February and it will thus shift to 3 March.

You need to call `Calendar#clear()` after obtaining/creating it:

```
GregorianCalendar cal = new GregorianCalendar();
cal.clear();
// ...
```

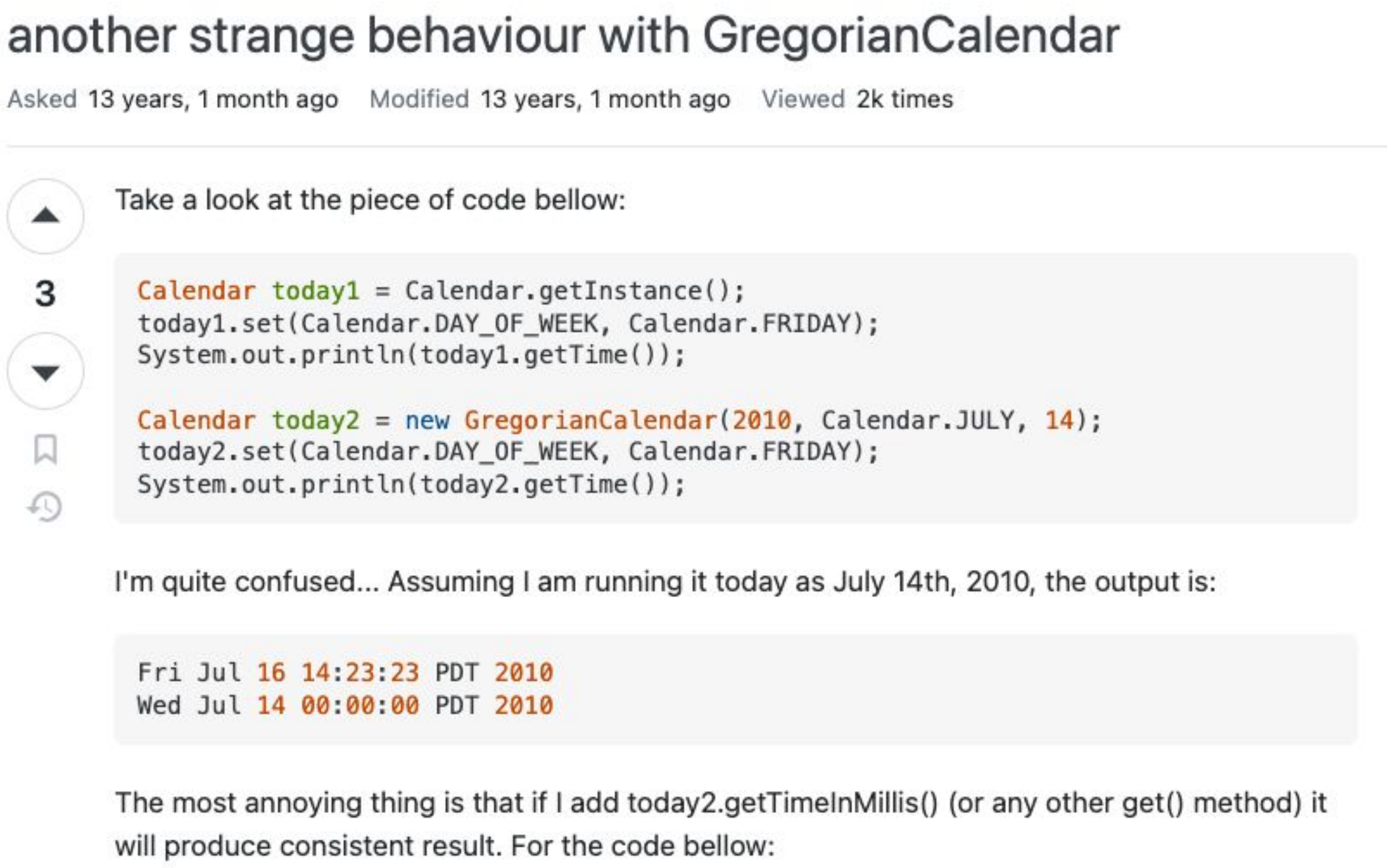
This results in:

```
Sun Feb 28 23:59:59 GMT-04:00 2010
```



If you can't use JodaTime or JSR-310 in your project, unit test heavily when using the Calendar class. As you can see in this case Calendar code behaves differently depending on what day of the month (or what time of the day) you run the code.

https://stackoverflow.com/questions/2944585/strange-behaviour-with-gregoriancalendar



another strange behaviour with GregorianCalendar

Asked 13 years, 1 month ago Modified 13 years, 1 month ago Viewed 2k times

3

Take a look at the piece of code bellow:

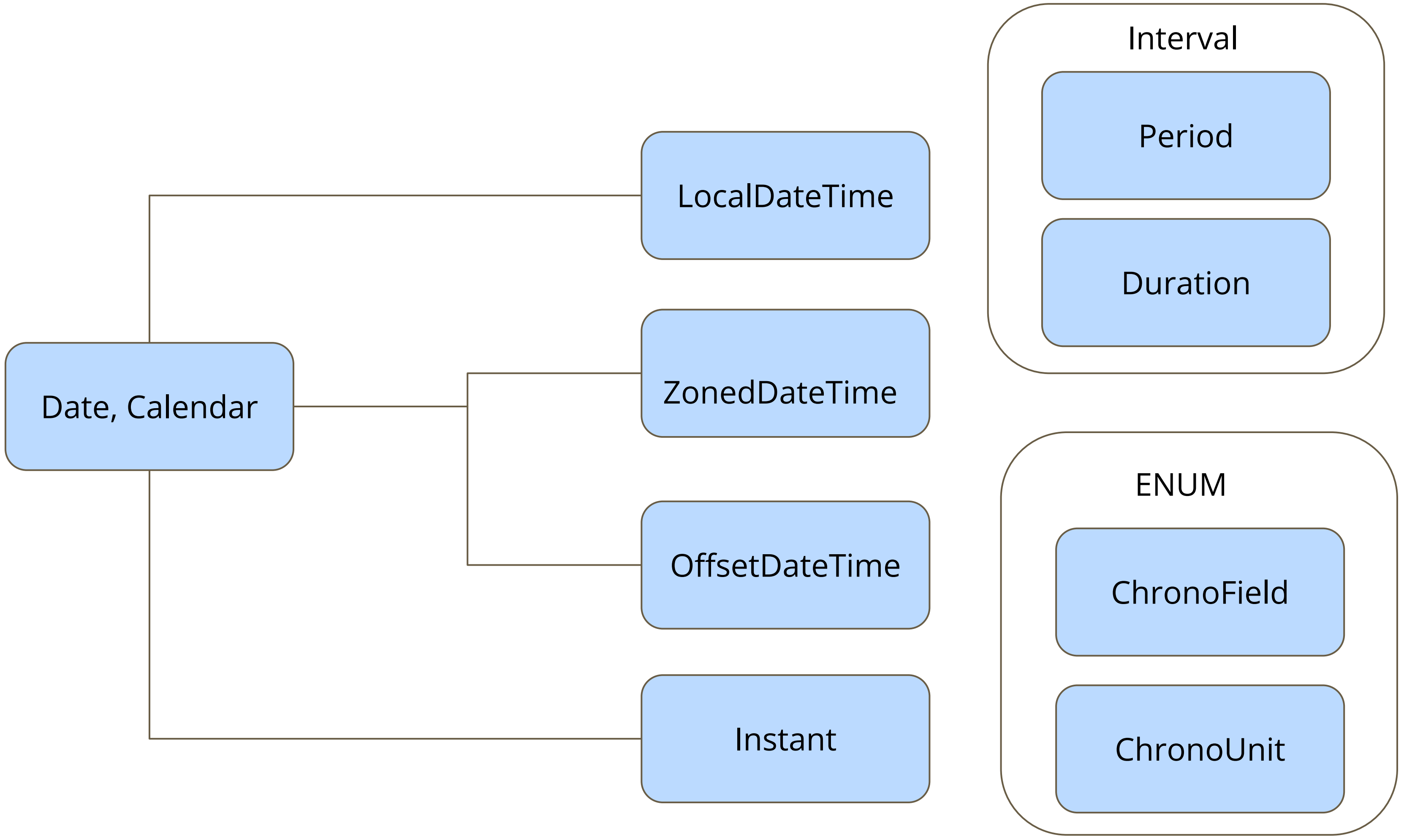
```
Calendar today1 = Calendar.getInstance();
today1.set(Calendar.DAY_OF_WEEK, Calendar.FRIDAY);
System.out.println(today1.getTime());

Calendar today2 = new GregorianCalendar(2010, Calendar.JULY, 14);
today2.set(Calendar.DAY_OF_WEEK, Calendar.FRIDAY);
System.out.println(today2.getTime());
```

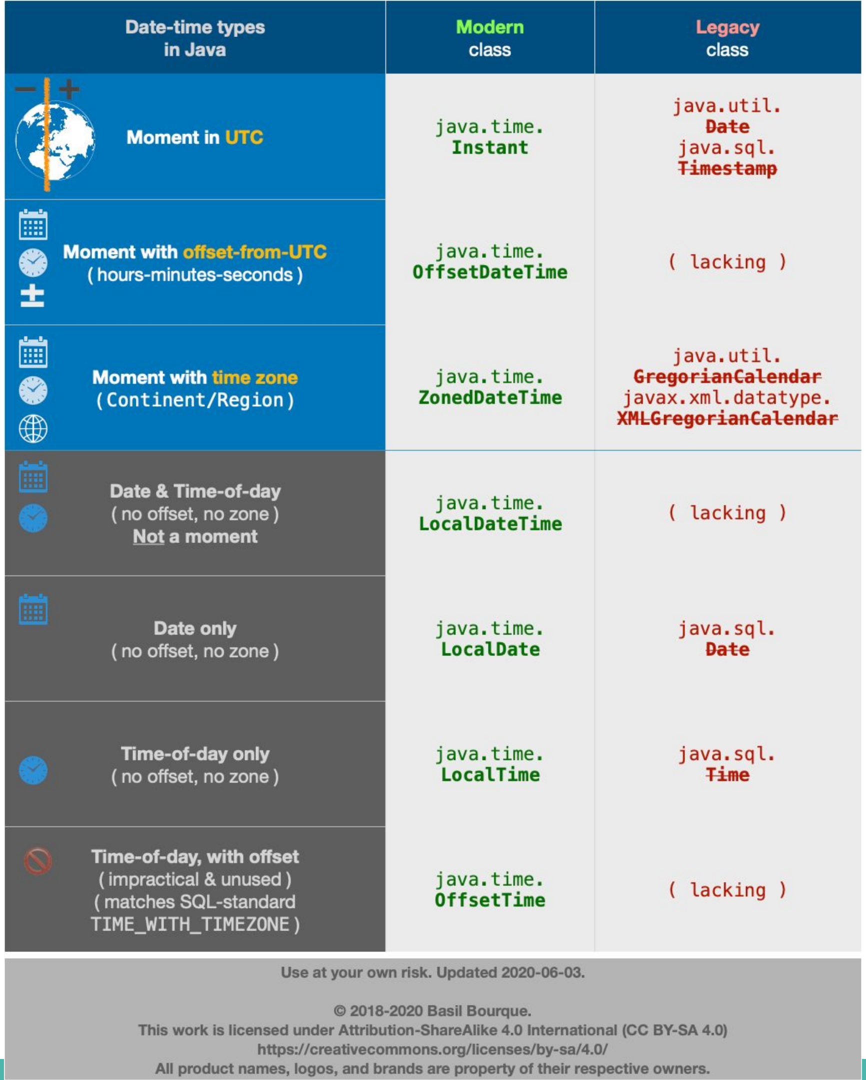
I'm quite confused... Assuming I am running it today as July 14th, 2010, the output is:

```
Fri Jul 16 14:23:23 PDT 2010
Wed Jul 14 00:00:00 PDT 2010
```

The most annoying thing is that if I add today2.getTimeInMillis() (or any other get() method) it will produce consistent result. For the code bellow:

https://stackoverflow.com/questions/3250733/another-strange-behaviour-with-gregoriancalendar


Date, Calendar
LocalDateTime
ZonedDateTime
OffsetDateTime
Instant
Interval
Period
Duration
ENUM
ChronoField
ChronoUnit



Date-time types in Java	Modern class	Legacy class
Moment in UTC	java.time. Instant	java.util. ~~Date~~ java.sql. ~~Timestamp~~
Moment with offset-from-UTC (hours-minutes-seconds)	java.time. OffsetDateTime	(lacking)
Moment with time zone (Continent/Region)	java.time. ZonedDateTime	java.util. ~~GregorianCalendar~~ javax.xml.datatype. ~~XMLGregorianCalendar~~
Date & Time-of-day (no offset, no zone) Not a moment	java.time. LocalDateTime	(lacking)
Date only (no offset, no zone)	java.time. LocalDate	java.sql. ~~Date~~
Time-of-day only (no offset, no zone)	java.time. LocalTime	java.sql. ~~Time~~
Time-of-day, with offset (impractical & unused) (matches SQL-standard TIME_WITH_TIMEZONE)	java.time. OffsetTime	(lacking)

LocalDateTime, ZonedDateTime, OffsetDateTime

	Data	Leap Year	Daylight Saving
LocalDateTime.now()	2023-09-03T13:44:09.982	?	NO
ZonedDateTime.now()	2023-09-03T13:44:09.982+07:00[Asia/Bangkok]	Yes	YES, but only if it is created with Zone information NO if it is created with UTC offset
OffsetDateTime.now()	2023-09-03T13:44:09.982+07:00	Yes	NO

Instant.now().atOffset(ZoneOffset.UTC)
Instant.now()
	2023-09-03T06:44:09.982Z

Instant

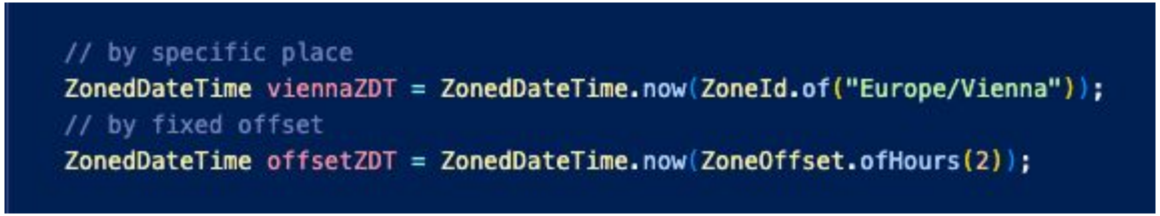
- A single moment in time in the UTC timezone

“This class models a single instantaneous point on the timeline. This might be used to record event **timestamps** in the application.”

* The number of seconds(-ish) as elapsed from the epoch reference point: 1 Jan 1970

ZonedDateTime

- Can be created by using
    - UTC Offset (**ZonedOffset)**
    - Continent/City format e.g. Europe/Vienna" or "Australia/Sydney" (**ZoneId**)



```
// by specific place
ZonedDateTime viennaZDT = ZonedDateTime.now(ZoneId.of("Europe/Vienna"));
// by fixed offset
ZonedDateTime offsetZDT = ZonedDateTime.now(ZoneOffset.ofHours(2));
```

LocalDateTime vs Instant

`Instant` stores timestamps in a Coordinated Universal Time (UTC) format and provides a machine-facing, or internal, time view. It is suitable for database storage, business logic, data exchange, and serialization scenarios. `LocalDateTime`, `OffsetDateTime`, and `ZonedDateTime` include time zone or seasonal information and also provide a human-friendly time view to input and output data to users. When the same time is output to different users, their values are different. For example, the shipping time of an order is shown to the buyer and seller in different local times. These three classes can be considered as external-facing tools, rather than the internal work part of the application.

In short, `Instant` is better for backend services and databases, while `LocalDateTime` and its cohorts are better for frontend services and displays. The two are in theory interchangeable but in reality serve different functions.

OffsetDateTime

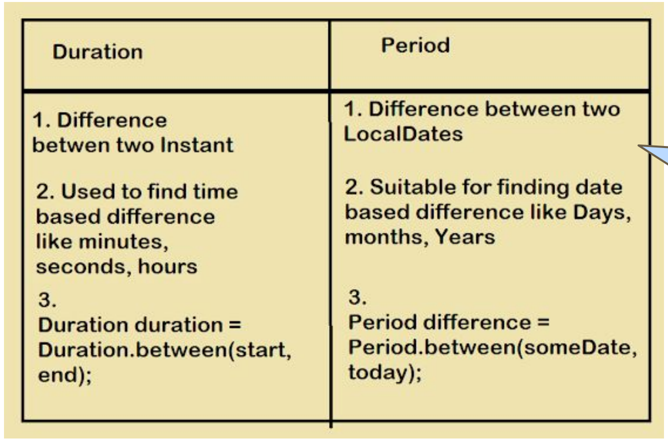
It is intended that **ZonedDateTime** or **Instant** is used to model data in simpler applications. This [**OffsetDateTime**] class may be used when modeling date-time concepts in more detail, or when communicating to a database or in a network protocol.

Duration

- works with **time**-based units
  - second, minute, hour
  - generally used with **Instant** (timestamp)
- Duration.toMinutes(), toSeconds(), toMills()

Period

- works with **date**-based units
  - day, month, year
  - only accepts **LocalDate** object
- Period.getMonths(), getDays(), and getYears()

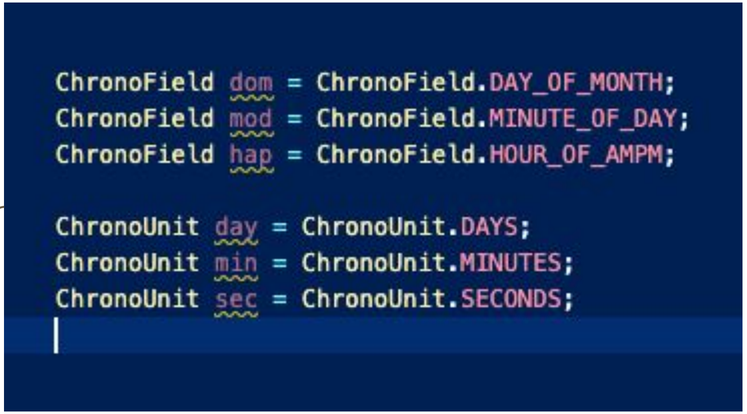


Duration	Period
1. Difference betwen two Instant	1. Difference between two LocalDates
2. Used to find time based difference like minutes, seconds, hours	2. Suitable for finding date based difference like Days, months, Years
3. Duration duration = Duration.between(start, end);	3. Period difference = Period.between(someDate, today);

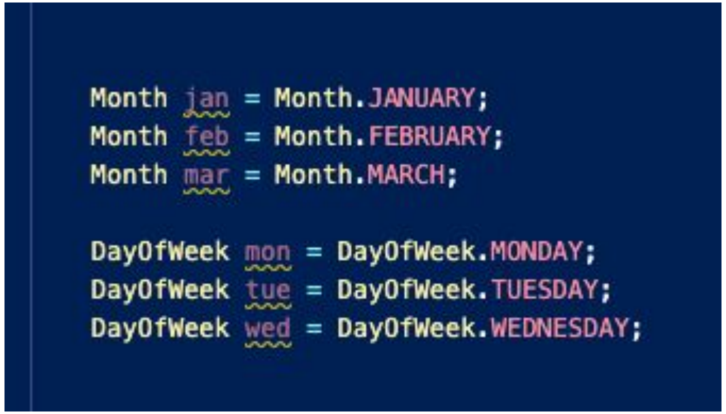
Does not work with Timezone or DST!

Use ZonedDateTime.plus..(..), ZonedDateTime.until(..) instead

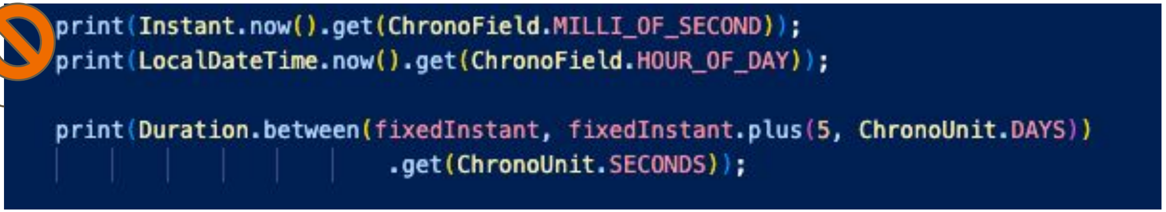
ChronoField, ChronoUnit, Other ENUMs



```java
ChronoField dom = ChronoField.DAY_OF_MONTH;
ChronoField mod = ChronoField.MINUTE_OF_DAY;
ChronoField hap = ChronoField.HOUR_OF_AMPM;

ChronoUnit day = ChronoUnit.DAYS;
ChronoUnit min = ChronoUnit.MINUTES;
ChronoUnit sec = ChronoUnit.SECONDS;
```



```java
Month jan = Month.JANUARY;
Month feb = Month.FEBRUARY;
Month mar = Month.MARCH;

DayOfWeek mon = DayOfWeek.MONDAY;
DayOfWeek tue = DayOfWeek.TUESDAY;
DayOfWeek wed = DayOfWeek.WEDNESDAY;
```



```java
print(Instant.now().get(ChronoField.MILLI_OF_SECOND));
print(LocalDateTime.now().get(ChronoField.HOUR_OF_DAY));

print(Duration.between(fixedInstant, fixedInstant.plus(5, ChronoUnit.DAYS))
                        .get(ChronoUnit.SECONDS));
```

LocalDateTime Coding Exercise:

#1 Does LocalDateTime take into account leap year?

ZonedDateTime Coding Exercise:

#1 If I watch the Ring tape on 28 Oct 2023 in Vienna and I shall die in 7 days, when will I die if the demon considers 7 days as

- 24*7 hours
- exactly 7 days

ZonedDateTime Coding Exercise:

#2 If a package gets picked up in Vienna (GMT+2) today at 8 AM and will be delivered to Sydney (GMT+10) taking exactly 5 hours and 30 minutes on a super-express international flight, when will the package arrive in Sydney?

OffsetDateTime Coding Exercise:

#1 Print me the current UTC datetime from

- Instant converted from ZonedDateTime (Vienna, Bangkok)
- Instant UTC

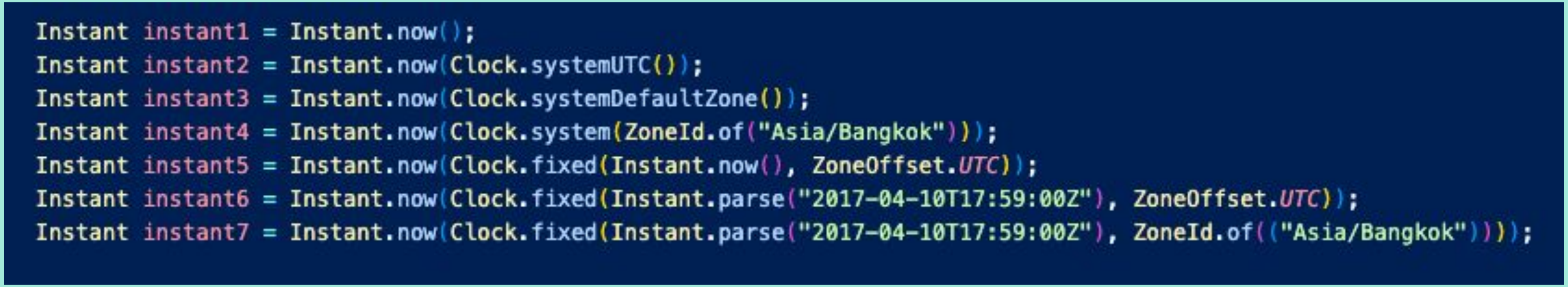
Instant Coding Exercise:

#1 Check if a token has expired using Instant given:

```
String tokenExpires = "2023-09-04T07:06:25.092Z";
```

Homework: Using Clock

#1 Why should we use **Instant.now(Clock)** or **LocalDateTime.now(Clock)** instead of **Instant.now()** or **LocalDateTime.now()** in unit tests and when?
#2 What is the difference between each Instant/timestamp created:



```java
Instant instant1 = Instant.now();
Instant instant2 = Instant.now(Clock.systemUTC());
Instant instant3 = Instant.now(Clock.systemDefaultZone());
Instant instant4 = Instant.now(Clock.system(ZoneId.of("Asia/Bangkok")));
Instant instant5 = Instant.now(Clock.fixed(Instant.now(), ZoneOffset.UTC));
Instant instant6 = Instant.now(Clock.fixed(Instant.parse("2017-04-10T17:59:00Z"), ZoneOffset.UTC));
Instant instant7 = Instant.now(Clock.fixed(Instant.parse("2017-04-10T17:59:00Z"), ZoneId.of(("Asia/Bangkok"))));
```