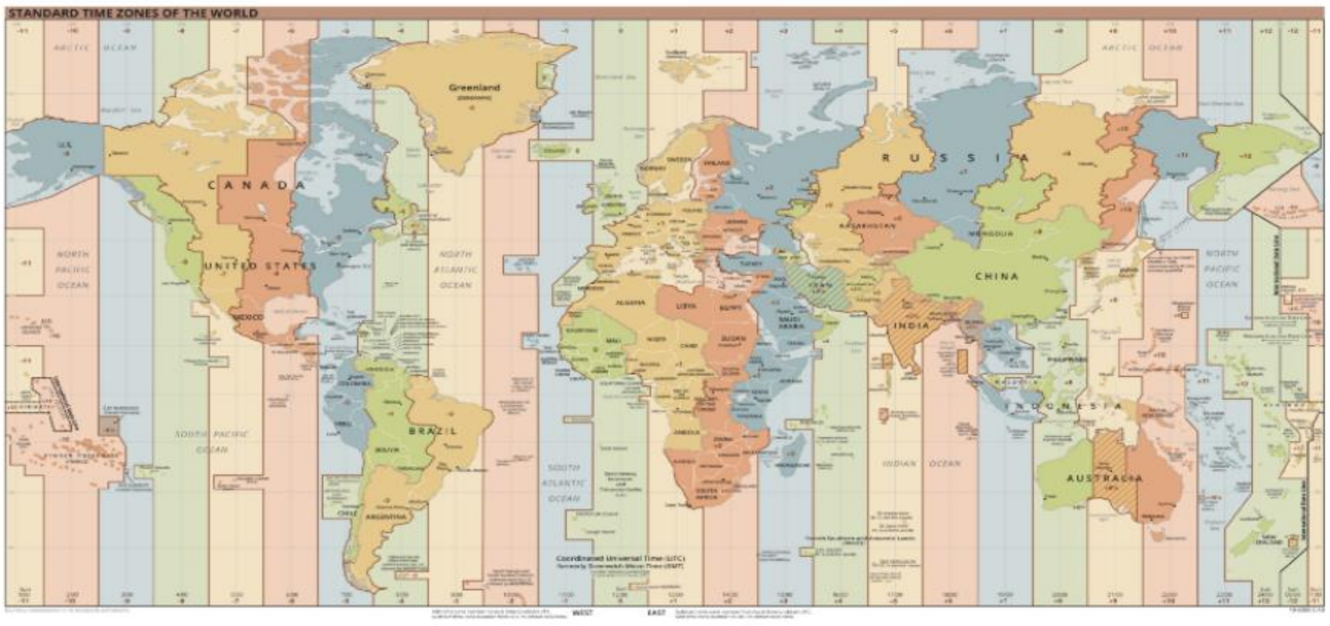
Date Time Api java 8

****

Java 8 Date & Time APIs



History:

Java 1.0: *java.util* package. Initially, **the** *"Date"* class was introduced. Instead of representing the actual date, it represented a specific instant in time with millisecond precision. With later releases, multiple classes were added to handle the date and time.

Cons:

1. Years starting from *1900* and months are *zero-index* based.
2. Formatting & parsing was inefficient and cannot format the date and time without using *java.text* package.
3. Weak input validation for Date constructor.
4. Not thread-safe.
5. Hard to deal with timezones.

Java 1.1: java.util.Calendar class has solved a few of the problems.

Pros:

1. ***Year Offset*** is managed by the ***Calendar*** class.
2. New Constants were added to deal with days & months.
3. ***java.text.DateFormat*** were introduced to parse the string dates.

Cons:

1. The calendar class is ***mutable*** which leads to thread safety problems.
2. java.text.DateFormat is not safe threaded.
3. Using and managing zoned date is *hard* and confusing even for some skilled java developers.

Problems remained:

1. Months are still ***zero-index*** based.
2. Difficult to deal with the calculation of intervals between two dates.
3. Zoned Date Time management is still hard to work.
4. Date class exists in two different packages(java.util, java.sql)

- **java.sql**: package deals with date formats that are suitable for databases.
- **java.util**: package deals with date formats that are suitable for java language.

*In 2005, the well-known third-party **JodaTime** was introduced by **joda.org** which has provided clear concepts and APIs for Java 5 to 7.*

Java 8 Date & Time packages:

- java.time
- java.time.chrono
- java.time.format
- java.time.temporal
- java.time.zone

Java.time Classes:

- Clock
- Duration
- Instant
- LocalDate
- LocalDateTime
- LocalTime
- MonthDay
- OffsetDateTime
- OffsetTime
- Period
- Year
- YearMonth
- ZonedDateTime
- ZoneId
- ZoneOffset

Common Methods for all Classes:

- **of** - static factory method
- **parse** - static factory method focussed on parsing
- **get** - gets the value of something
- **is** - checks if something is true
- **with** - the immutable equivalent of a setter
- **plus** - adds an amount to an object
- **minus** - subtracts an amount from an object
- **to** - converts this object to another type
- **at** - combines this object with another, such as date.atTime(time)

Java 8: java.time package was introduced with Date & Time Classes and are ***immutable*** and ***thread-safe***. The *java.time* package was greatly inspired by *JodaTime*.

Pros:

1. Rich set of APIs with consistent API design for easier development.
2. Flexible and Effective API.
3. Immutable & thread-safe.
4. *DateTimeFormatter* & *DateTimeFormatterBuilder* are introduced for formatting and parsing of dates.

LocalDate: Represents only the ***date*** in ISO format(**yyyy-MM-dd**).

LocalDate class:-

- An instance of current LocalDate can be created from the System Clock as below:

```
LocalDate localDate = LocalDate.now(); // 2022-03-02
```

- We can get a specific instance of a LocalDate by using factory methods(***of*** & ***parse***). For example, to create a LocalDate instance of 31st January 2022, it can be done either of the below ways:

```
LocalDate localDateWithOfFactory = LocalDate.of(2022,01,31);       //
2022-01-31

LocalDate localDateWithParseFactory = LocalDate.parse("2022-01-31");
// 2022-01-31
```

- To create ***future*** date (ahead of current date) events such as *nextDay, nextWeek, nextMonth, nextYear etc* can be done by *plusDays(), plusMonths() ,plusWeeks() , plusYears()* methods of LocalDate.

```
LocalDate currentDate = LocalDate.now(); // 2022-03-02
LocalDate nextLocalDate = currentDate.plusDays(1); // 2022-03-03
LocalDate nextMonth = currentDate.plusMonths(2); // 2022-05-02
LocalDate nextWeeks = currentDate.plusWeeks(2); // 2022-03-16
LocalDate nextYear = currentDate.plusYears(1); // 2023-03-02
```

- *ChronoUnit* is an alternate way to get an instance of future LocalDate. For example, getting a LocalDate of the next date(tomorrow) can be done by passing the *ChronoUnit.DAYS* enum to the plus constructor along with the days(1) ahead value.

```
LocalDate nextLocalDate = LocalDate.now().plus(1, ChronoUnit.DAYS); //
2022-03-03
```

- To create a ***historical*** date (behind of current date) events such as previous day, *previousWeek, previous year etc* can be done by *minusDays(), minusMonths(), minusYears()* methods of LocalDate.

```
LocalDate currentDate = LocalDate.now(); // 2022-03-02
LocalDate prevDay = currentDate.minusDays(1); // 2022-03-01
LocalDate prevMonth = currentDate.minusMonths(2); // 2022-01-02
LocalDate prevWeeks = currentDate.minusWeeks(3); // 2022-02-09
LocalDate prevYear = currentDate.minusYears(1); // 2021-03-02
```

- With ChronoUnit as well, it is possible to create historical date events.

```
LocalDate prevDay = LocalDate.now().minus(1, ChronoUnit.DAYS);     // 
2022-03-01
```

- LocalDate provides ***utility*** methods to obtain information such as *getDayOfWeek(), getDayOfMonth(), getMonth(), isLeapYear(), isAfter(), isBefore() etc.*

```
LocalDate localDate = LocalDate.parse("2022-01-31");

DayOfWeek dayOfWeek = localDate.getDayOfWeek(); // MONDAY
int dayOfMonth = localDate.getDayOfMonth(); // 31
int dayOfYear = localDate.getDayOfYear(); // 31

Month month = localDate.getMonth(); // JANUARY
int monthValue = localDate.getMonthValue(); // 1
int year = localDate.getYear(); // 2022

boolean isLeapYear = localDate.isLeapYear(); // false
int lengthOfMonth = localDate.lengthOfMonth(); // 31

LocalDate prevDay = LocalDate.parse("2022-01-30");
boolean isAfter = localDate.isAfter(prevDay); // true
boolean isBefore = prevDay.isBefore(localDate); // true
```

LocalTime class:-

LocalTime: Represents only the *time* in ISO format(**hh:mm:ss.nanos**).

- An instance of current LocalTime can be created from the System Clock as below:

```
LocalTime localTime = LocalTime.now(); // 20:32:36.268082
```

- We can get a specific instance of a LocalTime by using factory methods(*of* & *parse*). For example, to create a LocalTime instance of 11:10:30, it can be done either of the below ways:

```
LocalTime localTimeWithOf = LocalTime.of(11,10,30); // 11:10:30

LocalTime localTimeWithParse=LocalTime.parse("11:10:30"); //11:10:30
```

- To create ***future*** time (ahead of current time) events such as *nextHour, nextMinute, nextSecond, nextNano etc* can be done by *plusHours(), plusMinutes() ,plusSeconds() , plusNanos()* methods of LocalTime.

```
LocalTime currentTime = LocalTime.now(); // 20:32:36.268082
LocalTime nextHour = currentTime.plusHours(1); // 21:32:36.268082

LocalTime nextMinutes = currentTime.plusMinutes(10); //20:42:36.268082

LocalTime nextSeconds = currentTime.plusSeconds(10); //20:32:46.268082

LocalTime nextNanos = currentTime.plusNanos(1000); //20:32:36.268082
```

- With ChronoUnit as well, it is possible to create *future* time events such as below:

```
LocalTime next =  LocalTime.now().plus(1, ChronoUnit.HOURS);
//21:32:36.268082
```

- To create ***historical*** time (behind current time) events such as *previousHours, previousMinutes, previousSeconds etc* can be done by *minusHours(), minusMinutes(), minusSeconds()* methods of LocalTime.

```
LocalTime currentTime = LocalTime.now(); // 20:32:36.268082

LocalTime prevHour = currentTime.minusHours(1); // 19:32:36.268082

LocalTime prevMinutes = currentTime.minusMinutes(10);
//20:22:36.268082

LocalTime prevSeconds = currentTime.minusSeconds(10);
//20:32:26.268082
```

- With ChronoUnit as well, it is possible to create *future* time events such as below:

```
LocalTime prevHour = LocalTime.now().minus(1,ChronoUnit.HOURS);
```

- Finest time instances such as *hour, minute, second* can be fetched by *getHour(), getMinute(), getSecond() etc* methods of LocalTime.

```
LocalTime currentTime = LocalTime.now(); //20:32:36.268082
int getHour = currentTime.getHour(); //20
int getMinute = currentTime.getMinute(); //32
int getSecond = currentTime.getSecond(); //36
int getNano = currentTime.getNano(); //268082
```

- LocalTime provides ***utility*** methods to obtain information such as *isAfter(), isBefore() and constants like MAX, MIN, MIDNIGHT, NOON etc.*

```
LocalTime maxTime =  LocalTime.MAX; // 23:59:59.999999999
LocalTime minTime =  LocalTime.MIN; //00:00
LocalTime midnight  = LocalTime.MIDNIGHT; //00:00
LocalTime noon = LocalTime.NOON;//12:00

LocalTime localTimeOfSixThirty = LocalTime.parse("06:30");
LocalTime localTimeOfSevenThirty = LocalTime.parse("07:30");

boolean isBefore =
localTimeOfSixThirty.isBefore(localTimeOfSevenThirty);  // true

boolean isAfter localTimeOfSixThirty.isAfter(localTimeOfSevenThirty);
//false
```

LocalDateTime class:-

LocalDateTime: It is a combination of LocalDate and LocalTime and represents *date* and *time* without timezone in **YYYY-MM-DDThh:mm:ss** format.

Current Instance:

```
LocalDateTime localDateTime = LocalDateTime.now();
//2022-03-03T20:32:36.268082
```

Factory methods:

```
LocalDateTime specificLocalDateTime = LocalDateTime.of(2021,
Month.FEBRUARY,01,12,34); //2021-02-01T12:34

LocalDateTime specificLocalDateTime1 = LocalDateTime.parse("2021-02-
01T12:34:00"); //2021-02-01T12:34
```

- Utility APIS is also available to support addition and subtraction of specific time units like days, months , ,etc.

For example:

```
LocalDateTime currentInstance = LocalDateTime.now();

LocalDateTime nextYear = currentInstance.plusYears(1);
LocalDateTime nextHour = currentInstance.plusHours(1);
LocalDateTime prevYear = currentInstance.minusYears(1);
LocalDateTime prevHour = currentInstance.minusHours(1);
```

- Getter methods are also available to extract specific units similar to the date and time classes.

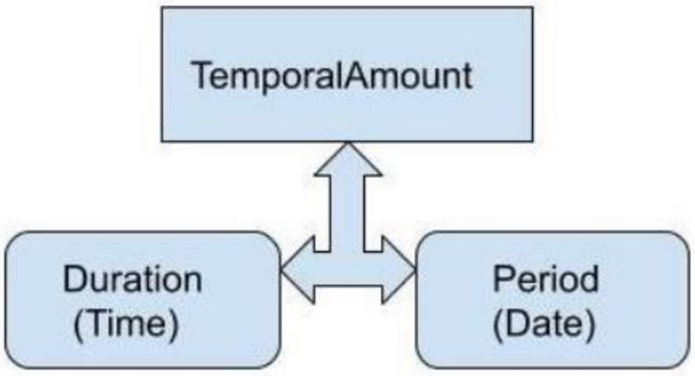
```
int dayOfMonth = currentTime.getDayOfMonth();

DayOfWeek dayOfWeek = currentTime.getDayOfWeek();
int dayOfYear = currentTime.getDayOfYear();
int monthValue = currentTime.getMonthValue();

int hour =  currentTime.getHour();
int minute = currentTime.getMinute();
```

Instant: It is a *specific* point in the continuous timeline. It represents the seconds passed since the Epoch time 1970–01–01T00:00:00Z.

TemporalAmount: Now java provides the TemporalAmount interface which represents an amount of time and it is implemented by two classes *Duration* and *Period*.



Duration &Period class:-

1. **Duration:** It is the amount of time in terms of seconds and nanoseconds. It has *utility* methods to get respective *hours, minutes, millis and nanos.*

```
LocalTime localTime = LocalTime.now();

LocalTime nextHour = localTime.plus(Duration.ofHours(1));
LocalTime prevMinute = localTime.minus(Duration.ofMinutes(1));
```

It also provides the *between()* to compute duration of two temporal objects

```
LocalTime time1 = LocalTime.of(11,12,34);
LocalTime time2 = LocalTime.of(10,10,10);

long seconds = Duration.between(time2, time1).getSeconds();
long hours = ChronoUnit.HOURS.between(time2,time1);
```

2) Period: It is the amount of time in terms of years, months, weeks and days. Period class is widely used to modify the values of a given date or to obtain the difference between two dates.

```
LocalDate currentDate = LocalDate.now();

LocalDate nextDates = currentDate.plus(Period.ofDays(5));
LocalDate prevWeek = currentDate.minus(Period.ofWeeks(1));

LocalDate date1 = LocalDate.parse("2022-03-31");
LocalDate date2 = LocalDate.parse("2021-01-01");

int days = Period.between(date2, date1).getDays();
int months = Period.between(date2, date1).getMonths();
int years = Period.between(date2, date1).getYears();
long chronoDays = ChronoUnit.DAYS.between(date2, date1);
```

Temporal Adjusters: It is a functional interface which has predefined static methods to adjust Temporal Objects.

e.g: Find the last day of the month, get next Tuesday etc.

```
LocalDateTime localDateTime = LocalDateTime.now();

LocalDate endOfMonth = localDateTime
                 .with(TemporalAdjusters.lastDayOfMonth());

LocalDate nextTue = localDateTime
                 .with(TemporalAdjusters.next(DayOfWeek.TUESDAY));
```

Handling Daylight Savings Time(DST): Until Java7, the *java.util.TimeZone* class was used together with *Calendar* class but the usage was not simple. With Java 8, various classes were introduced to deal with timezone, which was easier than before.

ZoneID: defines a unique id for a region/city. e.g: Europe/Rome.

```
ZoneId zoneId = ZoneId.of("Europe/Paris");
```

ZoneOffset represents a timezone with an offset from Greenwich/UTC. e.g: +03:00

ZonedDateTime: represents a date time as per ISO-8601 calendar system with timezone. e.g: 2020–06–10T08:00:15+01:00[Europe/Paris].

- Fetch all the available zone Ids like below:

```
Set<String> allZoneIds = ZoneId.getAvailableZoneIds();
```

• Conversion of **LocalDate to ZonedDateTime:**

```
LocalDateTime localDateTime = LocalDateTime.now();

ZonedDateTime zonedDateTime =
ZonedDateTime.of(localDateTime,ZoneId.of("Europe/Paris"));

OR

ZonedDateTime zonedDateTime = ZonedDateTime.parse("2022-02-
01T10:15:30+01:00[Europe/Paris]");
```

OffsetDateTime: represents a date-time from UTC/Greenwich as per the ISO-8601 calendar system with an *offset*. e.g: 2020–06–10T08:00:15+01:00 . It is another way to work with timezone and is an *immutable* representation of a date-time with an offset.

- Creation of ZonedDateTime With OffsetDateTime:

```
LocalDateTime localDateTime = LocalDateTime.of(2022, Month.FEBRUARY,
01, 06, 30);
ZoneOffset offset = ZoneOffset.of("+05:30");
OffsetDateTime offsetDateTime = OffsetDateTime.of(localDateTime,
offset);
```

- Get **Milliseconds** From LocalDateTime & ZonedDateTime:

```
//Millis from ZonedDateTime
ZonedDateTime zonedDateTime =
localDateTime.atZone(ZoneId.of("Asia/Kolkata"));

long millis = zonedDateTime.toInstant().toEpochMilli();

//Millis From LocalDateTime
LocalDateTime localDateTime = LocalDateTime.of(2022, Month.FEBRUARY,
01, 06, 30);

localDateTime.atOffset(ZoneOffset.of("+05:30")).toInstant().toEpochMil
li();
```

- Get **LocalDateTime** from milliseconds:

```
LocalDateTime localDateTime = LocalDateTime.ofInstant(
                               Instant.ofEpochMilli(1643694635000l),
                               ZoneId.of("Asia/Kolkata")
                              );
```

- Get **Date** from milliseconds:

```
LocalDate localDate = Instant.ofEpochMilli(1643694635000l)
                      .atZone(ZoneId.of("Asia/Kolkata"))
                      .toLocalDate();
```

ZoneRules: are the actual set of rules that define when the zone offset changes.

```
boolean resp = ZoneRules.of(ZoneOffset.of("-08:00"))
.isDaylightSavings(Instant.now());
```

Formatting & Parsing:

Formatting & Parsing:

- **2 ways:** DateTimeFormatter, DateTimeFormatterBuilder.

DateTimeFormatter: It is the substitute for old *java.text.DateFormat* has three popular methods:

1. *ofPattern(String pattern)*: creates a formatted using the specified pattern. It returns a DateTimeFormatter Instance.
2. *format(TemporalAccessor temporal)*: formats a date-time object using this formatter. It returns a string.
3. *parse(CharSequence text)*: fully parses the text producing the temporal object. It returns an instance of TemporalAccessor type.

DateTimeFormatterBuilder: It works on the builder pattern to build custom patterns.

- we can pass in formatting style either as SHORT, LONG or MEDIUM as part of the formatting option.

```
LocalDateTime localDateTime = LocalDateTime.of(2022, Month.JANUARY,
25, 6, 30);

String localDateString = localDateTime.
format(DateTimeFormatter.ISO_DATE);

localDateString = localDateTime.
format(DateTimeFormatter.ofPattern("yyyy/MM/dd"));

localDateString = localDateTime.
format(DateTimeFormatter.ofLocalizedDateTime(FormatStyle.MEDIUM)
.withLocale(Locale.UK));
```