Date and Time in Java

eazy bytes

In Java, there are two distinct APIs for representing Date and Time values: one relies on classes within the **java.util, java.sql** package, while the other leverages the **java.time** package.

Legacy Date API

java.util.Date
java.util.Calendar
java.util.GregorianCalendar
java.text.SimpleDateFormat
java.sql.Date

New Date API introduced in Java 8

java.time.LocalDate
java.time.LocalDateTime
java.time.Instant
java.time.Duration
java.time.Period
java.time.format.DateTimeFormatter

Why a new Date API introduced in Java 8 ?

eazy bytes

The introduction of the java.time package in Java 8 was driven by several issues and limitations associated with the older date and time classes (java.util.Date, java.util.Calendar, SimpleDateFormat, etc.) that were part of the pre-Java 8 standard library. Here are some of the problems with the older date and time classes and the reasons behind the introduction of java.time:

Design Flaws: The java.util.Date class had design flaws, including ambiguity in its representation (it represented both a point in time and a date) and poor handling of time zones. This led to unpredictable behavior and made it difficult to work with dates and times accurately.

```
Date date = new Date(); //Tue Sep 26 16:07:42 IST 2030
```

Like in the screenshot, if we use java.util.Date to represent current date, but the value will have both Date and Time value. This creates confusion few developers

With legacy Date API, the year start from 1900 whereas the months start at index 0. Suppose if you want to represent a date of 26 Sep 2100, you need to create an instance of date using the code below. This is not very developer friendly

```
Date date = new Date(200,8,26); //Sun Sep 26 00:00:00 IST 2100
```

The issues with java.util.Date were addressed by introducing new classes, methods and deprecating some old ones. However, the replacement class, java.util.Calendar, had its own design flaws, leading to error-prone code. The coexistence of both classes often confused developers about which one to use.

Why a new Date API introduced in Java 8 ?

eazy bytes

Immutability and Thread Safety: The older java.util.Date and java.util.Calendar classes were mutable, which meant that their state could be changed after creation. This mutability could lead to synchronization issues in multithreaded applications. In contrast, the java.time classes are immutable, making them thread-safe by design.

Limited Date and Time Precision: java.util.Date had limited precision, as it could only represent time in milliseconds, which is inadequate for many modern applications that require nanosecond-level precision. The **java.time.Instant** class in java.time addresses this issue by representing time with nanosecond precision.

Inadequate Time Zone Handling: The older classes lacked proper support for time zones and daylight saving time (DST). Handling time zone-related issues often required complex and error-prone code. java.time provides comprehensive support for time zones, making it easier to work with different time zones and DST changes.

Verbosity and Complexity: Formatting and parsing dates and times using SimpleDateFormat in java.text was verbose and error-prone. java.time introduced the DateTimeFormatter class, which offers a simpler and more flexible way to format and parse date and time values.

In summary, the java.time package in Java 8 was introduced to address the shortcomings of the older date and time classes, providing a more robust, accurate, and developer-friendly API for handling date and time operations in Java. This new package aimed to simplify date and time manipulation while also addressing issues related to immutability, precision, time zones, and API design.

Should I learn Legacy Date API as well ?

eazy bytes

Offcourse YES. The legacy Date-Time API has been around for over 15 years. It is possible that you will encounter legacy datetime classes while working with existing applications. The legacy datetime classes have been retrofitted to work seamlessly with the new classes. When you write new code, use the new Date-Time API classes. When you receive objects of legacy classes as input, convert the legacy objects into new datetime objects, and use the new Date-Time API.

The Date Class

An object of the Date class represents an instant in time. A Date object stores the number of milliseconds elapsed since the epoch, midnight January 1, 1970, UTC.

A Date object works with a 1900-based year. When you call the setYear() method of this object to set the year as 2100, you will need to pass 200 (2100 – 1900 = 200). Its getYear() method returns 200 for the year 2100. Months in this class range from 0 to 11 where January is 0, February is 2 … and December is 11.

```
// Create a new Date object
Date currentDate = new Date();
// Get the milliseconds value of the current date
long millis = currentDate.getTime();
// Represent Sun Sep 26 00:00:00 IST 2100
Date futureDate = new Date(200,8,26);
```

Converting Between Date and Milliseconds

```
Date currentDate = new Date();
// Get milliseconds since the epoch
long milliseconds = currentDate.getTime();
// Convert milliseconds back to Date
Date convertedDate = new Date(milliseconds);
```

Comparing Dates

```
Date currentDate = new Date(); // Current date and time
// Add one day in milliseconds
Date futureDate = new Date(currentDate.getTime() + 86400000);
boolean isAfter = futureDate.after(currentDate); // true
```

Java.util.Date examples

eazy bytes

Mutating a Date (Not Recommended)

```
Date currentDate = new Date();
currentDate.setYear(123); // Mutating the year (avoid this)
currentDate.setDate(9); // Mutating the year (avoid this)
```

Formatting a Date

```
Date currentDate = new Date(); // Current date and time
SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
String formattedDate = dateFormat.format(currentDate);
```

Parsing a Date String

```
String dateString = "2100-09-26 14:30:00";
SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
Date parsedDate = dateFormat.parse(dateString);
```

java.sql.Date

eazy bytes

java.util.Date and java.sql.Date are both classes in Java used for representing dates and times, but they have different purposes and characteristics, primarily related to how they are used in the context of Java applications:

- **Purpose:** java.util.Date class is part of the core Java platform and is used for general-purpose date and time representation. It stores both date and time information. Where as java.sql.Date class is part of the Java Database Connectivity (JDBC) API and is primarily used for interacting with databases. It only stores date information and is designed for compatibility with SQL databases that have date columns.

- **Date vs. Date and Time:** java.util.Date stores both date and time information, including milliseconds since the Unix epoch (January 1, 1970, 00:00:00 GMT). java.sql.Date stores only date information (year, month, day) without time information.

- **Inheritance:** java.util.Date extends java.util.Object, which is the root class for all Java classes. java.sql.Date extends java.util.Date but overrides some methods to ensure that only date information is stored and retrieved when interacting with databases.

- **Usage:** java.util.Date is used for general-purpose date and time operations in Java applications and is not specific to database interactions. java.sql.Date is used when working with JDBC for database operations where a date-only representation is required. It's typically used for binding and retrieving date values to/from SQL databases.

- **Compatibility:** java.util.Date is not specifically designed for database interactions, so you may need to perform additional conversions when working with databases. java.sql.Date is specifically designed for compatibility with SQL databases, making it more straightforward to work with date columns in database queries and updates.

java.util.GregorianCalendar examples

eazy bytes

The Calendar Class

Calendar is an abstract class. An abstract class cannot be instantiated. The GregorianCalendar class is a concrete class, which inherits the Calendar class. It provides enhanced date and time handling capabilities over java.util.Date, including support for different calendar systems and time zones.

In GregorianCalendar, the year can be mentioned as it is with out worrying about substracting or adding from 1900.

The month part of a date ranges from 0 to 11. That is, January is 0, February is 1, and so on. It is easier to use the constants declared for months and the other date fields in the Calendar class rather than using their integer values. For example, you should use the Calendar.JANUARY constant to represent the January month in your program instead of a 0.

TimeZone information can be passed while creating the object of GregorianCalendar.

```
// Create a GregorianCalendar for the current date and time in the default
time zone.
GregorianCalendar calendar = new GregorianCalendar();

// Fri Jan 01 11:30:45 IST 2100
GregorianCalendar futureDate = new GregorianCalendar(2100, Calendar.JANUARY,
1, 11, 30, 45);

// Fri Jan 01 00:00:00 IST 2100
GregorianCalendar futureCalendar = new GregorianCalendar(2100,0,1);

// You can also specify a time zone.
GregorianCalendar newYorkCalendar = new
GregorianCalendar(TimeZone.getTimeZone("America/New_York"));

// Get current date in India
GregorianCalendar indianDate = new
GregorianCalendar(TimeZone.getTimeZone("GMT+05:30"));
```

Java.util.GregorianCalendar examples

eazy bytes

Getting Date and Time Components

```
GregorianCalendar calendar = new GregorianCalendar();
int year = calendar.get(Calendar.YEAR);
int month = calendar.get(Calendar.MONTH);
int day = calendar.get(Calendar.DAY_OF_MONTH);
int hour = calendar.get(Calendar.HOUR_OF_DAY);
int minute = calendar.get(Calendar.MINUTE);
int second = calendar.get(Calendar.SECOND);
int millisecond = calendar.get(Calendar.MILLISECOND);
```

Setting Date and Time Components

```
GregorianCalendar calendar = new GregorianCalendar();
calendar.set(Calendar.YEAR, 2023);
calendar.set(Calendar.MONTH, Calendar.SEPTEMBER);
calendar.set(Calendar.DAY_OF_MONTH, 26);
calendar.set(Calendar.HOUR_OF_DAY, 14);
calendar.set(Calendar.MINUTE, 30);
calendar.set(Calendar.SECOND, 0);
calendar.set(Calendar.MILLISECOND, 0);
```

Adding and Subtracting Time Units

```
GregorianCalendar calendar = new GregorianCalendar();
calendar.add(Calendar.HOUR, 2);
calendar.add(Calendar.DAY_OF_MONTH, -1);
```

Checking for Leap Year

```
GregorianCalendar calendar = new GregorianCalendar();
boolean isLeapYear = calendar.isLeapYear(2048);
```

Formatting Date and Time

```
GregorianCalendar calendar = new GregorianCalendar();
SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
String formattedDate = dateFormat.format(calendar.getTime());
```

Introduction to new Date & Time API

eazy bytes

With all the limitations that Java util Date/Calendar has Developers started using third party date and time libraries, such as Joda Time• For these reasons, Java development team decided to provide high quality date and time support in the native Java API. As a result, **Java 8** integrates many of the Joda Time features in the **java.time** package

The new java.time package has the below important classes to deal with Date & Time,

java.time package

LocalDate
LocalDateTime
LocalTime
ZonedDateTime
OffsetDateTime
Instant
Duration
Period

Enums
DayOfWeek
Month

java.time.temporal package

Temporal
TemporalAccessor
TemporalAdjuster
TemporalQuery

Enums
ChronoField
ChronoUnit

java.time.format package

DateTimeFormatter

java.time Date & Time API

eazy bytes

LocalDate represents a day, LocalTime represents a time, LocalDateTime represents a day and time, and ZonedDateTime represents a day and time in a specific time zone.

LocalDate represents a date without a time, such as "2100-01-01"

LocalTime represents a time without a date, such as "15:53:46".

LocalDateTime represents a date and time without a time zone, such as "2100-01-01T15:53:46".

ZonedDateTime represents a date and time with a time zone, such as "2100-01-01T15:53:46IST".

In the Date-Time API, classes for representing date, time, and datetime include a **now()** method that provides the current date, time, or datetime. The following code snippet demonstrates how to create datetime objects to represent date, time, and their combination, both with and without specifying a time zone.

```
LocalDate dateOnly = LocalDate.now();
LocalTime timeOnly = LocalTime.now();
LocalDateTime dateTime = LocalDateTime.now();
ZonedDateTime dateTimeWithZone = ZonedDateTime.now();
```

The of() methods in java.time Date & Time API

eazy bytes

The **of()** method in the java.time package is a static factory method available in various classes within the package, such as LocalDate, LocalTime, LocalDateTime, ZonedDateTime, and more. This method is used to create instances of date and time objects with specific values.

The **of()** method typically takes arguments to specify the desired date and time components, such as year, month, day, hour, minute, second, and nanosecond, depending on the class. The method then returns an instance of the respective class initialized with the specified values. Here are some common use cases of the of() method:

```
// Creates a LocalDate for Mar 18, 2015
LocalDate date = LocalDate.of(2015, 3, 18);
// Creates a LocalTime for 2:30 PM
LocalTime time = LocalTime.of(14, 30);
// Creates a LocalDateTime for Mar 18, 2015, 2:30 PM
LocalDateTime dateTime = LocalDateTime.of(2015, 3, 18, 14, 30);
// Creates a ZonedDateTime for 2015-03-18T14:30-04:00[America/New_York]
ZoneId newYorkTimeZone = ZoneId.of("America/New_York");
ZonedDateTime zonedDateTime = ZonedDateTime.of(2015, 3, 18, 14, 30, 0, 0,
newYorkTimeZone);
```

The of() method ensures that the values provided are valid for the corresponding date and time components. For example, it will check that the month is between 1 and 12, the day is within the valid range for that month, and so on.

There are other ofXxx() methods as well available where Xxx represer description of the parameters. Here are the few examples,

```
// Give 1000 day from epoch 1970-01-01 which is 1972-09-27
LocalDate someDate = LocalDate.ofEpochDay(1000);
// Give 100th day in the year 2000 which is 2000-04-09
LocalDate dayIn2000 = LocalDate.ofYearDay(2000, 100);
```

The from() & withXxx() methods in java.time Date & Time API

eazy bytes

A from() method is a static factory method, similar to an of() method, but it's used to create a datetime object by deriving its values from a specified argument. Unlike an of() method, a from() method involves converting the provided argument.

Here's an example code snippet that demonstrates how to derive a LocalDate from a LocalDateTime:

```
// 2015-03-18T22:30
LocalDateTime someDateTime = LocalDateTime.of(2015, 3, 18, 22, 30);
LocalDate derivedDate = LocalDate.from(someDateTime); // 2015-03-18
```

Many classes in the Date-Time API are designed to be immutable, meaning they cannot be modified once created. Instead of having setXxx() methods to change their fields, these classes provide withXxx() methods. A withXxx() method returns a new copy of the object with the specified field modified.

For example, if you have a LocalDate object and want to change its year, you can use the withYear(int newYear) method provided by the LocalDate class. You can also chain multiple withXxx() method calls to create a new object with several field changes. In this example, we change both the year and the month of a LocalDate. Here are the examples,

```
LocalDate date1 = LocalDate.of(2015, Month.MARCH, 18);//2015-03-18
LocalDate date2 = date1.withYear(2030);//2030-03-18
LocalDate date3 = date1.withYear(2030).withMonth(10);//2030-10-18
```

This approach ensures that the original objects remain unchanged (immutable) while allowing you to work with modified copies of the objects.

The toXxx() & atXxx() methods in java.time Date & Time API

eazy bytes

The **toXxx()** methods in the Date-Time API are used to convert an object to a related type. These methods allow you to extract specific information or transform one type of date-time object into another related type. Here are some examples of using toXxx() methods:

```
LocalDateTime localDateTime = LocalDateTime.of(2015, 3, 18, 14, 15);
LocalTime localTime = localDateTime.toLocalTime(); // 14:15
LocalDate localDate = localDateTime.toLocalDate(); // 2015-03-18
long epochDays = localDate.toEpochDay();  // 16512
```

An **atXxx()** method in the Date-Time API allows you to construct a new datetime object from an existing one by adding supplementary information. To understand the distinction between using an atXxx() method and a withXxx() method, consider that the former enables the creation of a new type of object with additional details, whereas the latter allows you to make a copy of an object while altering its existing fields.

For instance, suppose you start with the date "2015-03-18." If you wish to generate a new date, "2015-06-18," by changing only the month, you would utilize a withXxx() method. However, if you want to create a datetime like "2015-03-18T18:20" by adding the time "18:20," you would employ an atXxx() method.

```
LocalDate localDate = LocalDate.of(2015, 3, 18); //2015-03-18
// Using atStartOfDay() to add the start of the day (00:00)
LocalDateTime startOfDay = localDate.atStartOfDay(); //2015-03-18T00:00
// Using atTime() to add the time 15:30
LocalDateTime localDateTime = localDate.atTime(18, 20); //2015-03-18T18:20
```

The getXxx(), plusXxx() and minusXxx() methods in java.time Date & Time API

eazy bytes

A **getXxx()** method in the Date-Time API is used to retrieve a specific element or component of a datetime object. For example, the getYear() method in the LocalDate class allows you to extract the year part of a date. Here's how you can obtain the year, month, and day from a LocalDate object:

```
LocalDate localDate = LocalDate.of(2015, 3, 18);
// Get the year
int year = localDate.getYear(); // 2015
// Get the month
Month month = localDate.getMonth(); // Month.MARCH
// Get the day of the month
int day = localDate.getDayOfMonth(); // 18
LocalDateTime localDateTime = LocalDateTime.now();
int hour = localDateTime.getHour(); // Return hour value
```

The Date-Time API provides **plusXxx()** and **minusXxx()** methods that enable you to create copies of date and time objects by adding or subtracting specific values. Here are some examples:

The plusDays(long days) method in the LocalDate class adds a specified number of days to a LocalDate object and returns a copy with the new date.
The plusMonths(int months) method adds a specified number of months.
The plusWeeks(long weeks) method adds a specified number of weeks.
Similarly, the minusXxx() methods subtract values in a similar fashion.

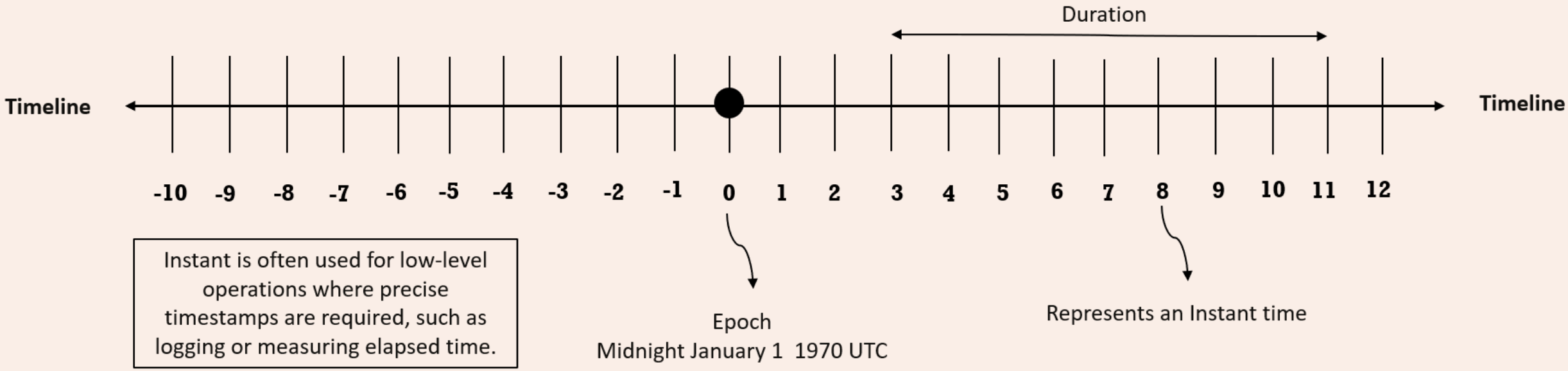
```
LocalDate localDate = LocalDate.of(2015, 3, 18);  // 2015-03-18
// Adding 6 days
LocalDate ld1 = localDate.plusDays(6); // 2015-03-24
// Adding 6 months
LocalDate ld2 = localDate.plusMonths(6); // 2015-09-18
// Adding 2 weeks
LocalDate ld3 = localDate.plusWeeks(2); // 2015-04-01
// Subtracting 9 months
LocalDate ld4 = localDate.minusMonths(9); // 2014-06-18
// Subtracting 7 years
LocalDate ld5 = localDate.minusYears(7); // 2008-03-18
```

Instant & Duration in java.time Date & Time API

eazy bytes

For humans we use Date and time, but for machines the time is calculated based on number of seconds passed from the Unix epoch time, set by convention to midnight of January 1 1970 UTC. So to represent them we have **Instant** class

Duration can be used to identify the duration of time between two instances, times and date Times The difference will be given in terms of hours, minutes, seconds and nano seconds



An instance of the Instant class signifies a precise point in time along a timeline. This timeline represents time in simplified UTC (Coordinated Universal Time) with nanosecond precision. In this representation, the time interval, or duration, between two consecutive points on the timeline is precisely one nanosecond.

This timeline designates January 1, 1970, at midnight UTC (1970-01-01T00:00:00Z) as the starting point, known as the epoch. Instants occurring after this epoch are assigned positive values, while instants preceding it are given negative values. The instant exactly at the epoch holds a value of zero. This system allows for accurate timestamping and measurement of time intervals with nanosecond precision.

Instant & Duration in java.time Date & Time API

eazy bytes

Both Instant and LocalDateTime represent date and time in Java, but they have fundamental differences in how they do so:

Feature	Instant	LocalDateTime
Represents	Specific moment in time (UTC)	Date and time (time zone not specified)
Internal representation	Seconds and nanoseconds since Unix epoch	Year, month, day, hour, minute, second
Time zone dependence	Independent	No time zone information
Use cases	Timestamps, durations, time zone independent	Displaying user-friendly dates/times

When to use which:

Use Instant when you need to represent a specific point in time, regardless of location or time zone. This is ideal for timestamps, durations, and calculations.

Use LocalDateTime when you need to display dates and times to users in their local time zone or when the specific moment itself isn't critical. However, remember that LocalDateTime alone doesn't represent a unique moment and needs additional context (like a time zone) to be fully defined.

Instant & Duration in java.time Date & Time API

eazy bytes

Few example methods under Instant & Duration along with their description and output,

```java
// An instant: 1000 seconds from the epoch
Instant i1 = Instant.ofEpochSecond(1000); // 1970-01-01T00:16:40Z
// Get the current instant
Instant i2 = Instant.now();
// Gets the number of seconds from the Java epoch of 1970-01-01T00:00:00Z.
long seconds = i2.getEpochSecond();
// Returns the nanoseconds within the second,
int nanoSeconds = i2.getNano();
// A duration of 5 days
Duration d1 = Duration.ofDays(5); // PT120H
// A duration of 10 minutes
Duration d2 = Duration.ofMinutes(10); //PT10M
Duration d3 = Duration.ofSeconds(30); // PT30S
Duration d4 = Duration.ofSeconds(-13); // PT-13S
Duration d5 = d3.plus(d4); // PT17S
Instant i3 = i1.plus(d1); // 1970-01-06T00:16:40Z
Instant i4 = i1.minus(d2); // 1970-01-01T00:06:40Z
boolean isBefore = i3.isBefore(i4); // true
boolean isAfter = i3.isAfter(i4); // false
```

Period in java.time Date & Time API

eazy bytes

You should use **Duration** to measure the difference in time like the difference between two Instant (milliseconds from epoch), but use **Period** to calculate the difference between dates. Here are some key points about Period and how it differs from Duration:

Units of Measurement: Duration represents time-based durations measured in terms of seconds, minutes, hours, and nanoseconds. It deals with precise time intervals. Period, on the other hand, represents calendar-based durations measured in terms of years, months, and days. It deals with differences in terms of calendar units.

Representation: Duration is typically represented in the ISO-8601 format, such as PT10H30M for 10 hours and 30 minutes. Period is represented in the ISO-8601 period format, such as P1Y2M3D for 1 year, 2 months, and 3 days.

Use Cases: Use Duration when you need to work with precise time intervals, such as measuring the time between two Instant timestamps.
Use Period when you need to represent a duration in terms of years, months, and days, such as calculating the difference between two LocalDate instances.

Few example methods of Period along with their description and output,

```
LocalDate localDate1 = LocalDate.of(2013,1,1); // 2013-01-01
LocalDate localDate2 = LocalDate.of(2015,3,18); // 2015-03-18
Period period = Period.between(localDate1,localDate2); // P2Y2M17D

Period oneYearTwoMonths = Period.of(1,2,0); // P1Y2M
Period threeDays = Period.ofDays(3); // P3D
LocalDate localDate3 = localDate1.plus(oneYearTwoMonths); // 2014-03-01
```

The multipliedBy(), dividedBy() & negated() methods in java.time Date & Time API

eazy bytes

In the Date-Time API, concepts like multiplication, division, and negation don't apply to date and time objects, as they represent specific points or intervals on the timeline. These operations are meaningful for types like Duration and Period, which represent amounts of time. Below are the example usage of these methods,

```
Duration originalDuration = Duration.ofMinutes(30); // PT30M
Duration multipliedDuration = originalDuration.multipliedBy(3); // PT1H30M
Duration dividedDuration = originalDuration.dividedBy(2); // PT15M
Duration negatedDuration = originalDuration.negated(); // PT-30M

Period originalPeriod = Period.ofDays(3); // P3D
Period multipliedPeriod = originalPeriod.multipliedBy(3); // P9D
Period negatedPeriod = originalPeriod.negated(); // P-3D
```

These methods are particularly useful when you need to perform simple arithmetic operations on durations, periods such as scaling a duration up or down by a factor, or changing the direction of a duration. They provide a convenient way to work with time-based intervals and adjust them according to your requirements in a clean and readable manner.

The truncatedTo() method

eazy bytes

The truncatedTo() method in the Java java.time.Duration class allows you to obtain a modified copy of a duration with time units smaller than the specified unit "truncated" or dropped. It's important to note that the temporal unit you specify must be equal to or smaller than days (i.e., DAYS, HOURS, MINUTES, SECONDS, or smaller). Attempting to use larger units like WEEKS or YEARS will result in a runtime exception.

Here's a practical example of how to use this method:

```
// Create a duration of PT678H56M19S
Duration d=Duration.ofDays(28).plusHours(6).plusMinutes(56).plusSeconds(19);
// PT672H
Duration daysTruncated = d.truncatedTo(ChronoUnit.DAYS);
// PT678H
Duration hoursTruncated = d.truncatedTo(ChronoUnit.HOURS);
// PT678H56M
Duration minutesTruncated = d.truncatedTo(ChronoUnit.MINUTES);
```

Please note the same truncatedTo(TemporalUnit unit) method also exists in the LocalTime and Instant classes as well.

java.time.ZoneId, ZoneOffset, ZonedDateTime, OffsetDateTime & OffsetTime

eazy bytes

ZoneId: **ZoneId** represents a time zone identifier, which can be a geographical region or a fixed offset from UTC (Coordinated Universal Time). It is used to define time zone rules and obtain **ZonedDateTime** instances for a specific time zone. ZonedDateTime also provides inbuilt functions, to convert a given date from one time-zone to another.

```java
// Get the set of all time zone IDs.
Set<String> allZones = ZoneId.getAvailableZoneIds();
ZoneId zone = ZoneId.of("Asia/Kolkata");
ZoneId destZone = ZoneId.of("America/Chicago");
// 2023-09-28T15:04:20.222928+05:30[Asia/Kolkata]
ZonedDateTime date = ZonedDateTime.now(zone);
//2023-09-28T04:34:20.222928-05:00[America/Chicago]
ZonedDateTime destDate = date.withZoneSameInstant(destZone);
```

```java
// Represents a +02:00 offset
ZoneOffset offset = ZoneOffset.ofHours(2);
// Represents a -08:00 offset
ZoneOffset destOffset = ZoneOffset.ofHours(-8);
// 2023-09-28T11:45:13.396952+02:00
ZonedDateTime date = ZonedDateTime.now(offset);
// 2023-09-28T01:45:13.396952-08:00
ZonedDateTime destDate = date.withZoneSameInstant(destOffset);
// 2023-09-28T11:45:13.397781+02:00
OffsetDateTime offsetDateTime = OffsetDateTime.now(offset);
// 2023-09-28T01:45:13.397781-08:00
OffsetDateTime destOffsetDateTime =
offsetDateTime.withOffsetSameInstant(destOffset);
```

ZoneOffset: ZoneOffset represents a fixed offset from UTC, typically used in cases where a specific time zone is not required. It is defined by an offset in hours and minutes from UTC, such as +02:00 or -05:00. OffsetDateTime represents a date and time with a fixed offset from UTC. Unlike ZonedDateTime, it doesn't contain information about time zone rules, daylight saving time changes, or geographical regions.

Both **ZonedDateTime** and **OffsetDateTime** have their places in applications depending on whether you need to work with time zone-specific data or fixed offsets. Choose the one that aligns with your specific requirements and use cases.

java.time.ZoneId, ZoneOffset, ZonedDateTime, OffsetDateTime & OffsetTime

eazy bytes

A ZoneId object combines a zone offset with the regulations governing changes to the zone offset during periods of Daylight Saving Time. It's important to note that not all time zones adhere to Daylight Saving Time. To simplify your grasp of ZoneId, you can conceptualize it as follows:

ZoneId = ZoneOffset + ZoneRules

OffsetTime is a class used to represent a specific time of day with a fixed offset from UTC (Coordinated Universal Time). It includes information about both the time of day and the offset from UTC, such as +02:00 or -05:00. OffsetTime is particularly useful when you need to work with time values that are not tied to a specific date or time zone but still require knowledge of the offset from UTC.

```
// 15:30+02:00
OffsetTime offsetTime = OffsetTime.of(15, 30, 0, 0, ZoneOffset.ofHours(2));
int hour = offsetTime.getHour(); // 15
int minute = offsetTime.getMinute(); // 30
int second = offsetTime.getSecond(); // 0
ZoneOffset offset = offsetTime.getOffset(); // +02:00
OffsetTime laterTime = offsetTime.plusHours(2); // 17:30+02:00
boolean isBefore = offsetTime.isBefore(laterTime); // true
```

In summary, you should choose the appropriate class based on your specific requirements. If you need to work with time zones and handle daylight saving time, use ZonedDateTime. If you only need to deal with fixed offsets, use OffsetDateTime. ZoneId and ZoneOffset provide the means to specify time zones and offsets, respectively, that can be used with these date-time classes.

Non-ISO Calendars in java.time Date & Time API

eazy bytes

In the Java java.time package, the primary calendar system used is the ISO calendar, which is based on the Gregorian calendar and is widely used internationally. However, the java.time package also provides support for non-ISO calendar systems through the java.time.chrono package. This allows you to work with alternative calendar systems such as the Thai Buddhist calendar, Hijrah calendar, Minguo calendar, and Japanese calendar, which are used in specific regions and cultures.

```
JapaneseDate japaneseDateNow = JapaneseDate.now(); // Japanese Reiwa 5-09-28
LocalDate isoNow = LocalDate.now(); // 2023-09-28

// Convert Japanese date to ISO date and vice versa
JapaneseDate jpd = JapaneseDate.from(isoNow); // Japanese Reiwa 5-09-28
LocalDate isoNow2 = LocalDate.from(japaneseDateNow); // 2023-09-28
```

For each of the non-ISO calendar systems available in Java, there is a pair of classes: one for the calendar system itself and another for representing dates within that system.

The XxxChronology class stands for the specific calendar system (e.g., Japanese or Hijrah) and provides the rules and calculations for that calendar. It includes a constant named INSTANCE that holds a singleton instance of that particular chronology.

For instance, if you're working with the Hijrah calendar system, you'll use the HijrahChronology class to define the rules and characteristics of the Hijrah calendar. Similarly, the HijrahDate class represents a date within the Hijrah calendar system.

Formatting Dates and Times in java.time Date & Time API

eazy bytes

Formatting dates and times in Java using the java.time package involves using the **DateTimeFormatter** class. DateTimeFormatter provides a flexible way to convert date and time objects into formatted strings and parse strings into date and time objects.

Below are the examples on how to format dates and times using **DateTimeFormatter,**

```
LocalDate localDate = LocalDate.of(2015,3,18);
LocalTime localTime = LocalTime.of(15,30,0);
LocalDateTime localDateTime = LocalDateTime.of(2015,3,18,15,30,0);

DateTimeFormatter dateFormatter = DateTimeFormatter.ofPattern("yyyy/MM/dd");
String formattedDate = dateFormatter.format(localDate);

DateTimeFormatter timeFormatter = DateTimeFormatter.ofPattern("h:mm a");
String formattedTime = timeFormatter.format(localTime);

DateTimeFormatter dateTimeFormatter = DateTimeFormatter.ofPattern("dd/MM/yyyy h:mm a");
String formattedDateTime = dateTimeFormatter.format(localDateTime);

String formattedDate1 = localDate.format(DateTimeFormatter.ISO_LOCAL_DATE);

DateTimeFormatter germanFormatter = DateTimeFormatter.ofPattern("d. MMMM yyyy", Locale.GERMAN);
String formattedDateGermany = germanFormatter.format(localDate);
```

Parsing String Dates and Times in java.time API

eazy bytes

Parsing involves the conversion of a string into a date, time, or datetime object. Similar to formatting, parsing is accomplished using a `**DateTimeFormatter**`.

Below are the examples on how to parse String dates and times using **DateTimeFormatter,**

```
DateTimeFormatter dateFormatter = DateTimeFormatter.ofPattern("yyyy-MM-dd");
DateTimeFormatter timeFormatter = DateTimeFormatter.ofPattern("HH:mm:ss");
DateTimeFormatter dateTimeFormatter = DateTimeFormatter.ofPattern("dd/MM/yyyy HH:mm:ss");

String dateString = "2015-03-18";
String timeString = "15:30:00";
String dateTimeString = "18/03/2015 15:30:00";

LocalDate localDate = LocalDate.parse(dateString,dateFormatter);
LocalTime localTime = LocalTime.parse(timeString,timeFormatter);
LocalDateTime localDateTime = LocalDateTime.parse(dateTimeString,dateTimeFormatter);
```