Instant

The java.time package provides us with classes for working with chronographic units.

Instant class: a **date-time unit** suitable for capturing event timestamps during the application execution.

Creational methods

It provides static methods for creating the date and time in the **ISO 8601** format.

Instant values are stored in a long variable storing the seconds counting from the **Java epoch** and an int variable storing the nanoseconds of a second.

Epoch is a common date-time representation format counting the date and time relative to January 1, 1970 (midnight UTC/GMT) which looks like 1970-01-01T00:00:00Z in the ISO 8601 format.

You can obtain this value with the Instant.EPOCH static field.

```
Instant epoch = Instant.EPOCH;
System.out.println(epoch); // 1970-01-01T00:00:00Z
```

In the code above the last Z stands for the time-zone ID, which is zero in our case. So, it shows us the epoch time of the GMT0 time zone

So, in Java, when working with the Instant class, you specify the date before or after the epoch time counting in seconds and nanoseconds.

exmaple:

```
long posValue = 1234567890L;
long negValue = -1234567890L;

// Adding specified milliseconds to epoch
Instant milli = Instant.ofEpochMilli(posValue); // 1970-01-15T06:56:07.890Z
// Adding specified seconds to epoch
Instant second = Instant.ofEpochSecond(posValue); // 2009-02-13T23:31:30Z
// Adding specified seconds and nanoseconds to epoch
Instant secondNano = Instant.ofEpochSecond(posValue, 123L); //
2009-02-13T23:31:30.000000123Z
// Adding specified seconds and nanoseconds to epoch. Version 2
Instant nanoRounded = Instant.ofEpochSecond(posValue, 1_000_000_123L); //
2009-02-13T23:31:31.000000123Z
```

```
// Subtracting specified milliseconds from epoch
Instant milli = Instant.ofEpochMilli(negValue); // 1969-12-17T17:03:52.110Z
// Subtracting specified seconds from epoch
Instant second = Instant.ofEpochSecond(negValue); // 1930-11-18T00:28:30Z
// Subtracting specified seconds and nanoseconds from epoch
Instant secondNano = Instant.ofEpochSecond(negValue, -150L); //
1930-11-18T00:28:29.999999850Z
// Subtracting specified seconds and nanoseconds from epoch. Version 2
Instant nanoRounded = Instant.ofEpochSecond(negValue, -1_000_000_150L); //
1930-11-18T00:28:28.999999850Z
```

Although ofEpochSecond(long epochSecond, long nanoAdjustment) accepts a long type variable, it rounds each **1bln** nanoseconds to a second and adds to the epochSecond, and the rest is stored in an int variable

here will be many situations where you will need to obtain an Instant unit for a given time zone. In such cases you can use the ZoneId class to specify the zone:

```
Instant instant = Instant.ofEpochSecond(1234567890L);

System.out.println(instant); // 2009-02-13T23:31:30Z
System.out.println(instant.atZone(ZoneId.of("GMT+4"))); //
2009-02-14T03:31:30+04:00[GMT+04:00]
System.out.println(instant.atZone(ZoneId.of("+04:00"))); //
2009-02-14T03:31:30+04:00
System.out.println(instant.atZone(ZoneId.of("Asia/Yerevan"))); //
2009-02-14T03:31:30+04:00[Asia/Yerevan]
System.out.println(instant.atZone(ZoneId.systemDefault())); //
2009-02-14T03:31:30+04:00[Asia/Yerevan]

System.out.println(ZoneId.systemDefault().getId()); // Asia/Yerevan
System.out.println(ZoneId.systemDefault().getRules()); //
ZoneRules[currentStandardOffset=+04:00]
```

If you don't know your time zone **ID** or the **offset** (the difference between a specified time zone and GMT0), the last two lines from the code above will help you. In our case, it shows the Armenia/Yerevan time zone.
The next method for creating Instant units is Instant.parse(), which creates a unit object by accepting a text and parsing it to the Instant type.

```
Instant instant = Instant.parse("2009-02-14T03:31:30Z");

System.out.println(instant); // 2009-02-14T03:31:30Z
System.out.println(instant.atZone(ZoneId.of("GMT+4"))); //
```

2009-02-14T07:31:30+04:00[GMT+04:00]

Operational methods

We will start with the simple isBefore()/isAfter() pair of methods comparing the chronological order of two units.

```
Instant instant1 = Instant.ofEpochSecond(123456L);
Instant instant2 = Instant.ofEpochSecond(123456789L);

System.out.println(instant1.isAfter(instant2)); // false
System.out.println(instant1.isBefore(instant2)); // true
```

Besides these two methods returning a boolean result of the comparison, the class implements the compareTo() method from the Comparable interface:

```
Instant instant1 = Instant.ofEpochSecond(123456L);
Instant instant2 = Instant.ofEpochSecond(123456789L);

System.out.println(instant1.compareTo(instant2)); // -1
```

This class doesn't implement the addTo() and subtractFrom() methods that are implemented in the Period and Duration classes, but it implements some others: minus(), plus() and their "subversions", such as minusSeconds(), plusSeconds(), and so on.

```
Instant instant = Instant.ofEpochSecond(123456L);

System.out.println(instant); // 1970-01-02T10:17:36Z

System.out.println(instant.minus(Period.ofDays(1))); // 1970-01-01T10:17:36Z
System.out.println(instant.minus(Duration.ofDays(1))); //
1970-01-01T10:17:36Z
System.out.println(instant.minus(1, ChronoUnit.DAYS)); //
1970-01-01T10:17:36Z

System.out.println(instant.plus(Period.ofDays(1))); // 1970-01-03T10:17:36Z
System.out.println(instant.plus(Duration.ofDays(1))); // 1970-01-03T10:17:36Z
System.out.println(instant.plus(1, ChronoUnit.DAYS)); // 1970-01-03T10:17:36Z

System.out.println(instant); // 1970-01-02T10:17:36Z
```

These two methods have limitations concerning supported unit types. They accept units that are smaller than days (inclusive).

```
Instant instant = Instant.ofEpochSecond(123456L);

System.out.println(instant.minus(Period.of(123, 12, 3))); //
```

```
UnsupportedTemporalTypeException
System.out.println(instant.plus(1, ChronoUnit.WEEKS)); //
UnsupportedTemporalTypeException

System.out.println(instant.plus(Period.ofWeeks(1))); // 1970-01-09T10:17:36Z
```

Here, all lines except the last one will throw an exception.

In the provided code snippet, instant.plus(Period.ofWeeks(1)) adds a Period of one week to the Instant object instant. This operation does not throw an error because Period objects only represent date-based amounts, such as years, months, and days, and do not consider time components like hours, minutes, or seconds.

This class has one more interesting method to calculate the difference between two units. It shows the time until another Instant unit is in the form of the specified unit:

```
Instant instant = Instant.ofEpochSecond(100200300L);
System.out.println(Instant.EPOCH.until(instant, ChronoUnit.DAYS)); // 1159
System.out.println(Instant.EPOCH.until(instant, ChronoUnit.HOURS)); // 27833
```

Get operations

There are three different methods for this purpose, which will show different results depending on how you created the Instant unit. We will consider one case using Instant.ofEpochSecond():

```
Instant ofEpochSecond = Instant.ofEpochSecond(123456L, 789L);
System.out.println(ofEpochSecond); // 1970-01-02T10:17:36.000000789Z
System.out.println(ofEpochSecond.getEpochSecond()); // 123456

System.out.println(ofEpochSecond.get(ChronoField.MICRO_OF_SECOND)); // 0
System.out.println(ofEpochSecond.get(ChronoField.MILLI_OF_SECOND)); // 0
System.out.println(ofEpochSecond.get(ChronoField.NANO_OF_SECOND)); //
789

System.out.println(ofEpochSecond.getLong(ChronoField.INSTANT_SECONDS))
; // 123456
System.out.println(ofEpochSecond.getLong(ChronoField.MICRO_OF_SECOND))
; // 0
System.out.println(ofEpochSecond.getLong(ChronoField.MILLI_OF_SECOND));
// 0
System.out.println(ofEpochSecond.getLong(ChronoField.NANO_OF_SECOND));
// 789
```

The first getEpochSecond() returns a long value storing the unit seconds. The other two perform similarly, but the get() method returns the value of the

specified unit as an int, while getLong() returns a long.
Since ofEpochSecond() doesn't use
milliseconds, get(ChronoField.MICRO_OF_SECOND) and
get(ChronoField.MILLI_OF_SECOND) return zero.

The same method calls with the same arguments wouldn't return zero if
the Instant unit was created using the ofEpochMilli() method.

Note that both the get() and getLong() methods accept only
those ChronoField units you can see in the code above. Passing other types will
cause an UnsupportedTemporalTypeException.

Instant vs LocalDateTime

Although Instant and LocalDateTime are date-time units, they are completely different in their nature. Instant is a representation of a moment on a timeline relative to an epoch.

LocalDateTime is a representation of a calendar date and daytime combination. The first one stores its value in seconds and nanoseconds, while the second one stores it as a pair of LocalDate and LocalTime objects.

Finally, LocalDateTime doesn't contain any information regarding a time zone, but if you create its object with the now() method, it will be created depending on your system's default zone. On the other hand, Instant contains that information and shows the GMT0 time-stamp by default in any case.

Instant instant = Instant.now(); // System time zone independent, shows GMT0
LocalDateTime dateTime = LocalDateTime.now(); // System time zone dependent