Tutoriel sur les nouveautés du langage 8 : la nouvelle API Date et Time

Par Yohan Beschi (Soat) - Soat

Date de publication : 24 juin 2013

Dernière mise à jour : 17 mars 2014

Cet article s'intéresse à détailler la nouvelle API pour gérer les dates et l'heure. Il a été rédigé quand le JDK 8 n'était pas encore à l'état finalisé, il est possible qu'il y ait des différences entre les fonctionnalités décrites dans cet article et celles proposées dans la version finale.

Pour réagir au contenu de cet article, un espace de dialogue vous est proposé sur le forum **Commentez**.

I - Introduction

Dans la **partie précédente**, nous avons vu tout ce qu'apporte le Projet Lambda à Java. Mais Java 8, ce n'est pas seulement les lambdas et un zeste de programmation fonctionnelle. Ce sont aussi de nombreuses améliorations tout autant révolutionnaires. Il aura fallu plus de dix ans pour que les développeurs soient entendus, mais le jour du salut est bientôt là ; nous allons enfin avoir une API de manipulation de dates digne de ce nom qui va se substituer à JodaTime.

Pour cet article, le JDK 8 build b94 a été utilisé. Le dernier build peut être téléchargé depuis **jdk8.java.net**.

L'intégralité des sources est disponible via **Github**.

II - java.time - JSR 310

La nouvelle API Date and Time est l'une des fonctionnalités les plus attendues et ce depuis de nombreuses années. Avec les JDK 7 et antérieurs, grâce à la méthode statique **System.currentTimeMillis()**, il est possible de récupérer le temps écoulé, en millisecondes, depuis le 1er janvier 1970. Si vous préférez utiliser des objets, il y a la classe **java.util.Date**, dont la plupart des méthodes ont été dépréciées dans le JDK8. Quant à la classe **java.util.GregorianCalendar**, elle permet d'effectuer - entre autres - des opérations sur les dates, comme ajouter ou soustraire x heures. Dans l'ensemble, ces classes et ces méthodes ne sont pas pratiques à utiliser. De ce fait, l'ancienne API a été délaissée par les développeurs au profit de **JodaTime**.

Il est donc indiscutable que nous avions besoin de quelque chose de simple à utiliser, de performant et d'objets immuables, dans la continuité de JodaTime. Mais la JSR 310 n'est pas un copier/coller de JodaTime, elle en est seulement inspirée, pour les raisons évoquées dans ce **post**. Stephen Colebourne, le créateur de JodaTime est aussi coleader de la **JSR 310**.

II-A - La nouvelle API

Cette nouvelle API est basée sur deux différents modèles de conception du temps. Le temps Machine et le temps Humain. Pour une machine, le temps n'est qu'un entier augmentant depuis l'epoch (01 janvier 1970 00h00min00s0ms0ns). Pour un humain en revanche, il s'agit d'une succession de champs ayant une unité (année, mois, jours, heure, etc.).

Les principes architecturaux autour desquels a été conçue la nouvelle API sous les suivants.

- **Immuabilité et thread safety :** Toutes les classes centrales de l'API Date and Time sont immuables, ce qui nous assure de ne pas avoir à nous soucier de problèmes de concurrence. De plus, qui dit objets immuables, dit des objets simples à créer, à utiliser et à tester.
- **Chaînage :** les méthodes chaînables rendent le code plus lisible et elles sont aussi plus simples à apprendre. Quant aux méthodes de type factory (par exemple: `now()`, `from()`, etc.) elles sont utilisées en lieu et place de constructeurs.
- **Clareté :** Chaque méthode définie clairement ce qu'elle fait. De plus, hormis dans quelques cas particuliers, passer un paramètre nul à une méthode provoquera la levée d'un `NullPointerException`. Les méthodes de validation prenant des objets en paramètre et retournant un booléen retournent généralement false lorsque null est passé.
- **Extensibilité :** Le design pattern Stratégie utilisé à travers l'API permet son extension en évitant toute confusion. Par exemple, bien que les classes de l' API soient basées sur le système de calendrier ISO-8601, nous pouvons aussi utiliser les calendriers non-ISO - tel que le calendrier Impérial Japonais - qui sont inclus dans l'API, ou même créer notre propre calendrier.

II-B - Le temps Machine

Pour commencer, voyons deux classes associées au temps machine, **java.time.Instant** et **java.time.Duration**.

II-B-1 - java.time.Instant

La classe java.time.Instant représente un point relatif à l'epoch.

```
1. /*
2.  * org.isk.datetime.MachineTimeTest
3.  */
4. @Test
5. public void instant() {
6.     //---- Instant.EPOCH
7.     Assert.assertEquals("1970-01-01T00:00:00Z", Instant.EPOCH.toString());
8.     Assert.assertEquals(Instant.parse("1970-01-01T00:00:00Z"), Instant.EPOCH);
9.     Assert.assertEquals(Instant.ofEpochSecond(0), Instant.EPOCH);
10.
11.     //---- Instant.MIN
12.     Assert.assertEquals(Instant.parse("-1000000000-01-01T00:00:00Z"), Instant.MIN);
13.
14.     //---- Instant.MAX
15.     Assert.assertEquals(Instant.parse("+1000000000-12-31T23:59:59.999999999Z"), Instant.MAX);
16.
17.     //---- Few instance methods
18.     final Instant instant = Instant.now();
19.
20.     // prints the current time
21.     // e.g. 2013-05-26T21:37Z (Coordinated Universal Time)
22.     System.out.println("Instant.now() : " + instant);
23.
24.     // print the number of nano seconds
25.     System.out.println("Instant.now().getNano() : " + instant.getNano());
26.
27.     //-- Working with 2 instants
28.     // 2013-05-26T23:10:40Z & 1530-05-26T23:10:40Z
29.     final Instant instant20130526_231040 = Instant.parse("2013-05-26T23:10:40Z");
30.     final Instant instant15300526_231040 = Instant.parse("1530-05-26T23:10:40Z");
31.
32.     // 2013-05-26T23:10:40Z is After 1530-05-26T23:10:40Z
33.     Assert.assertTrue(instant20130526_231040.isAfter(instant15300526_231040));
34.
35.     // 2013-05-26T23:10:40Z is NOT Before 1530-05-26T23:10:40Z
36.     Assert.assertFalse(instant20130526_231040.isBefore(instant15300526_231040));
37.
38.     // 2013-05-26T23:10:40Z minus 1 hour (3600s)
39.     Assert.assertEquals(Instant.parse("2013-05-26T22:10:40Z"),
40.                         instant20130526_231040.minusSeconds(3600));
41. }
```

Source.

- **Instant.EPOCH :** représente l'epoch.
- **Instant.MIN :** représente la plus petite valeur (pour les dates avant Jésus-Christ).
- **Instant.MAX :** représente la plus grande valeur possible (31 décembre un trillion).
- **Instant.parse() :** retourne un objet de type Instant à partir d'une chaîne de caractère représentant une date au format ISO-8601. Si la chaîne de caractères ne représente pas une valeur valide, une exception de type **java.time.format.DateTimeParseException** sera levée (le standard ISO-8601 spécifie que la lettre “T” désigne l'heure qu'elle précède et “Z” une data UTC).
- **Instant.ofEpochSecond()/Instant.ofEpochMilliSecond() :** retourne un objet de type Instant à partir d'un offset de x secondes ou millisecondes par rapport à l'epoch.
- **Instant.now() :** retourne un objet de type Instant représentant la date UTC actuelle.

- **Instant.now().getNano() :** retourne le nombre de nanosecondes de la date actuelle retournée par Instant.now().

Les méthodes isAfter(), isBefore(), minusSeconds(), etc. font ce que leur nom indique.

La méthode toString() sur une instance d'un objet Instant retourne la date au format ISO-8601.

II-B-2 - java.time.Duration

La classe java.time.Duration représente une durée.

```
1. /*
2.  * org.isk.datetime.MachineTimeTest
3.  */
4. @Test
5. public void duration() {
6.     //---- Duration.ZERO
7.     Assert.assertEquals(Duration.parse("PT0S"), Duration.ZERO);
8.     Assert.assertEquals("PT0S", Duration.ZERO.toString());
9.
10.     //---- 2h 5min 30s 345ms = 7_530_345ms
11.     final Duration duration = Duration.ofMillis(7_530_345);
12.     Assert.assertEquals("PT2H5M30.345S", duration.toString());
13.     Assert.assertEquals(7530, duration.getSeconds());
14.     Assert.assertEquals(345000000, duration.getNano());
15.
16. }
```

Source.

- **Duration.ZERO :** représente une durée nulle.
- **Duration.parse() :** retourne un objet de type Duration à partir d'une chaîne de caractères représentant une durée au format ISO-8601. Si la chaîne de caractères ne représente pas une valeur valide, une exception de type java.time.format.DateTimeParseException sera levée (selon le standard ISO-8601 une durée commence P - pour Period- et est suivie d'une valeur comme 4DT11H9M8S - pour Days, Hours, Minutes et Seconds - où la date et l'heure sont séparées par la lettre “T” - pour Time). Le standard permet aussi de définir un nombre d'années et de mois, ce que ne permet pas la méthode parse().
- **Duration.ofMillis() :** retourne un objet de type Duration à partir d'une chaîne de caractères représentant une durée en millisecondes.

La classe Duration, à l'instar de Instant, possède différents getters et méthodes de manipulation telles que plusX(), minusX(), etc. où X correspond à Days, Hours, Minutes, etc. ainsi que withY() où Y correspond à Seconds ou Nanos (ces méthodes retournent une copie de la durée ajustée avec la valeur passée en paramètre).

Les classes Instant et Duration étant immuables, les méthodes plusX(), minusX(), withY(), etc. retournent toujours de nouvelles instances.

II-C - Le temps Humain

II-C-1 - LocalDate, LocalTime, et LocalDateTime

Les classes **java.time.LocalDate**, **java.time.LocalTime** et **java.time.LocalDateTime** représentent des dates et heures système sans indication du fuseau horaire.

Les morceaux de code ci-dessous étant suffisamment clairs, je vous invite à les lire ainsi que leurs commentaires associés.

II-C-1-a - Les constantes

Divers constantes sont disponibles :

```
1. /*
2.  * org.isk.datetime.HumanTimeTest
3.  */
4. @Test
5. public void constants() {
6.     //---- LocalDate
7.     // LocalDate.MIN & LocalDate.MAX
8.     Assert.assertEquals("-999999999-01-01", LocalDate.MIN.toString());
9.     Assert.assertEquals("+999999999-12-31", LocalDate.MAX.toString());
10.
11.     //---- LocalTime
12.     // LocalTime.MIN & LocalTime.MAX
13.     Assert.assertEquals("00:00", LocalTime.MIN.toString());
14.     Assert.assertEquals("23:59:59.999999999", LocalTime.MAX.toString());
15.
16.     // LocalTime.NOON & LocalTime.MIDNIGHT
17.     // There is no mention of AM and PM
18.     Assert.assertEquals("12:00", LocalTime.NOON.toString());
19.     Assert.assertEquals("00:00", LocalTime.MIDNIGHT.toString());
20.
21.     //---- LocalDateTime
22.     // LocalDateTime.MIN & LocalDateTime.MAX
23.     Assert.assertEquals("-999999999-01-01T00:00", LocalDateTime.MIN.toString());
24.     Assert.assertEquals("+999999999-12-31T23:59:59.999999999", LocalDateTime.MAX.toString());
25. }
```

Source.

II-C-1-b - La méthode statique now()

La méthode statique `now()` retourne la date du système avec le fuseau horaire pris en compte, sauf si indiqué autrement en paramètre.

```
1. /*
2.  * org.isk.datetime.HumanTimeTest
3.  */
4. @Test
5. public void now() {
6.     //---- LocalDate
7.     // Current System Date
8.     // e.g. 2013-05-26
9.     System.out.println("LocalDate.now(): " + LocalDate.now());
10.
11.     // Current UTC System Date
12.     // e.g. 2013-05-26
13.     System.out.println("LocalDate.now(Clock.systemUTC()): " +
 LocalDate.now(Clock.systemUTC()));
14.
15.     //---- LocalTime
16.     // Current System Time
17.     //e.g. 21:35:45.977
18.     System.out.println("LocalTime.now(): " + LocalTime.now());
19.
20.     // Current UTC System Time
21.     //e.g. 19:35:45.977
22.     System.out.println("LocalTime.now(Clock.systemUTC()): " +
 LocalTime.now(Clock.systemUTC()));
23.
```

```
24.     //---- LocalDateTime
25.     // Current System Date and Time
26.     //e.g. 2013-05-26T21:35:45.977
27.     System.out.println("LocalDateTime.now(): " + LocalDateTime.now());
28.
29.     // Current UTC System Date and Time
30.     //e.g. 2013-05-26T19:35:45.977
31.     System.out.println("LocalDateTime.now(Clock.systemUTC()): " +
 LocalDateTime.now(Clock.systemUTC()));
32. }
```

Source.

II-C-1-c - Les méthodes statiques de construction

Il est possible de construire des dates à partir de différents formats.

```
1. /*
2.  * org.isk.datetime.HumanTimeTest
3.  */
4. @Test
5. public void localDateStaticMethods() {
6.     // LocalDate from a string
7.     final LocalDate localDateStr = LocalDate.parse("2013-05-23");
8.     Assert.assertEquals("2013-05-23", localDateStr.toString());
9.
10.     // LocalDate from 3 integers (year, month, day)
11.     final LocalDate localDate = LocalDate.of(2013, 05, 26);
12.     Assert.assertEquals("2013-05-26", localDate.toString());
13.
14.     // LocalDate with an offset from epoch
15.     final LocalDate oneHundredDaysBeforeEpoch = LocalDate.ofEpochDay(-1000);
16.     Assert.assertEquals("1967-04-07", oneHundredDaysBeforeEpoch.toString());
17.
18.     // Copy of a LocalDate
19.     final LocalDate copyLocalDate = LocalDate.from(localDate);
20.     Assert.assertEquals("2013-05-26", copyLocalDate.toString());
21. }
```

Source.

```
1. /*
2.  * org.isk.datetime.HumanTimeTest
3.  */
4. @Test
5. public void localTimeStaticMethods() {
6.     // LocalTime from a string
7.     final LocalTime timeStr1 = LocalTime.parse("12:35");
8.     Assert.assertEquals("12:35", timeStr1.toString());
9.
10.     final LocalTime timeStr2 = LocalTime.parse("12:35:32.978");
11.     Assert.assertEquals("12:35:32.978", timeStr2.toString());
12.
13.     // LocalDate from 3 integers (hour, minute, second)
14.     // But can be 2 (hour, minute)
15.     // or 4 (hour, minute, second, nanoseconds)
16.     final LocalTime timeAsInts = LocalTime.of(10, 22, 17);
17.     Assert.assertEquals("10:22:17", timeAsInts.toString());
18.
19.     // LocalTime from a number of seconds after midnight
20.     final LocalTime oneHourAfterMidnight = LocalTime.ofSecondOfDay(3600);
21.     Assert.assertEquals("01:00", oneHourAfterMidnight.toString());
22.
23.     // Copy of a LocalTime
24.     final LocalTime copyLocalTime = LocalTime.from(timeAsInts);
25.     Assert.assertEquals("10:22:17", copyLocalTime.toString());
26. }
```

Source.

```
1. /*
2.  * org.isk.datetime.HumanTimeTest
3.  */
4. @Test
5. public void localDateTimeStaticMethods() {
6.     // LocalDateTime from a string
7.     final LocalDateTime localDateTimeStr = LocalDateTime.parse("2013-05-26T10:22:17");
8.     Assert.assertEquals("2013-05-26T10:22:17", localDateTimeStr.toString());
9.
10.     // LocalDate from 5 parameters (year, month, day, hour, minute, second)
11.     // But range from 5 to 7.
12.     // Note : Month is an enum.
13.     final LocalDateTime localDateTime = LocalDateTime.of(2013, Month.MAY, 26, 12, 05);
14.     Assert.assertEquals("2013-05-26T12:05", localDateTime.toString());
15.
16.     // LocalDateTime from a LocalDate and a LocalTime
17.     final LocalDate localDate = LocalDate.of(2013, 05, 26);
18.     final LocalTime localTime = LocalTime.of(12, 35);
19.     final LocalDateTime localDateTimeOfDateAndTime = LocalDateTime.of(localDate, localTime);
20.     Assert.assertEquals("2013-05-26T12:35", localDateTimeOfDateAndTime.toString());
21.
22.     // Copy of a LocalDateTime
23.     final LocalDateTime copyLocalDateTime = LocalDateTime.from(localDateTime);
24.     Assert.assertEquals("2013-05-26T12:05", copyLocalDateTime.toString());
25. }
```

Source.

II-C-1-d - Les méthodes d'instances

Les classes LocalDate, LocalTime et LocalDateTime ont différentes méthodes permettant de récupérer une partie de la date, de la tester et d'effectuer des opérations dessus.

```
1. /*
2.  * org.isk.datetime.HumanTimeTest
3.  */
4. @Test
5. public void localDateInstanceMethods() {
6.     final LocalDate localDate = LocalDate.of(2013, 05, 26);
7.
8.     // Year
9.     Assert.assertEquals(2013, localDate.getYear());
10.
11.     // Month
12.     Assert.assertEquals(Month.MAY, localDate.getMonth());
13.     Assert.assertEquals(5, localDate.getMonthValue());
14.
15.     // Day
16.     Assert.assertEquals(26, localDate.getDayOfMonth());
17.     Assert.assertEquals(DayOfWeek.SUNDAY, localDate.getDayOfWeek());
18.     Assert.assertEquals(146, localDate.getDayOfYear());
19.
20.     // Leap Year
21.     Assert.assertFalse(localDate.isLeapYear());
22.     Assert.assertTrue(LocalDate.of(2004, 05, 26).isLeapYear());
23.
24.     //---- Operations
25.     final LocalDate localDate2 = LocalDate.of(2013, 04, 26);
26.
27.     // Before, After, Equal, equals
28.     Assert.assertTrue(localDate.isAfter(localDate2));
29.     Assert.assertFalse(localDate.isBefore(localDate2));
30.     Assert.assertTrue(localDate.isEqual(LocalDate.of(2013, 05, 26)));
31.     Assert.assertTrue(localDate.equals(LocalDate.of(2013, 05, 26)));
32.
```

```
33.     // plus & minus
34.     Assert.assertEquals("2013-04-26", localDate.minusMonths(1).toString());
35.     Assert.assertEquals("2013-06-05", localDate.plusDays(10).toString());
36.
37.     // Adjusters
38.     Assert.assertEquals("2013-05-01",
 localDate.with(TemporalAdjuster.firstDayOfMonth()).toString());
39.     Assert.assertEquals("2013-05-10", localDate.withDayOfMonth(10).toString());
40. }
```

Source.

```
1. /*
2.  * org.isk.datetime.HumanTimeTest
3.  */
4. @Test
5. public void localTimeInstanceMethods() {
6.     final LocalTime localTime = LocalTime.of(12, 35, 25, 452_367_943);
7.
8.     // Hour, Minute, Second, Nanosecond
9.     Assert.assertEquals(12, localTime.getHour());
10.     Assert.assertEquals(35, localTime.getMinute());
11.     Assert.assertEquals(25, localTime.getSecond());
12.     Assert.assertEquals(452_367_943, localTime.getNano());
13.
14.     //---- Operations
15.     // Before, After, Equal, equals
16.     final LocalTime localTime2 = LocalTime.of(12, 35, 25, 452_367_942);
17.     Assert.assertTrue(localTime.isAfter(localTime2));
18.     Assert.assertFalse(localTime.isBefore(localTime2));
19.     Assert.assertTrue(localTime.equals(LocalTime.of(12, 35, 25, 452_367_943)));
20.
21.     /// plus & minus
22.     Assert.assertEquals("12:25:25.452367943", localTime.minusMinutes(10).toString());
23.     Assert.assertEquals("17:35:25.452367943", localTime.plusHours(5).toString());
24.
25.     // Adjusters
26.     Assert.assertEquals("05:35:25.452367943", localTime.withHour(5).toString());
27. }
```

Source.

```
1. /*
2.  * org.isk.datetime.HumanTimeTest
3.  */
4. @Test
5. public void localDateTimeInstanceMethods() {
6.     final LocalDate localDate = LocalDate.of(2013, 05, 26);
7.     final LocalTime localTime = LocalTime.of(12, 35, 25, 452_367_943);
8.     final LocalDateTime localDateTime = LocalDateTime.of(localDate, localTime);
9.
10.     // As LocalDate & LocalTime
11.     Assert.assertEquals("2013-05-26", localDateTime.toLocalDate().toString());
12.     Assert.assertEquals("12:35:25.452367943", localDateTime.toLocalTime().toString());
13.
14.     // Other methods are the same as for LocalDate and LocalTime
15. }
```

Source.

II-C-2 - ZoneId, ZoneOffset, ZonedDateTime, et OffsetDateTime

Un fuseau horaire est une zone de la surface de la Terre dans laquelle toutes les localités ont le même temps standard. Chaque fuseau horaire a un identifiant (par exemple Europe/Paris) et un décalage par rapport à UTC/Greenwich (comme +01:00) qui change lorsque l'heure d'été est en vigueur.

II-C-2-a - ZoneId

La classe **ZoneId** représente un identifiant de fuseau horaire et fournit des règles de conversion entre Instant et LocalDateTime.

```
1. /*
2.  * org.isk.datetime.HumanTimeTest
3.  */
4. @Test
5. public void zoneId() {
6.     final ZoneId zoneId = ZoneId.systemDefault();
7.     final ZoneRules zoneRules = zoneId.getRules();
8.     Assert.assertEquals("Europe/Paris", zoneId.toString());
9.     Assert.assertEquals("ZoneRules[currentStandardOffset=+01:00]", zoneRules.toString());
10.
11.     // DST in effect
12.     Assert.assertTrue(zoneRules.isDaylightSavings(Instant.parse("2013-05-26T23:10:40Z")));
13.     Assert.assertFalse(zoneRules.isDaylightSavings(Instant.parse("2013-01-26T23:10:40Z")));
14. }
```

Source.

II-C-2-b - ZoneOffset

ZoneOffset décrit un offset de fuseau horaire, qui est un temps (généralement en heures) par lequel un fuseau horaire diffère de Greenwich.

```
1. /*
2.  * org.isk.datetime.HumanTimeTest
3.  */
4. @Test
5. public void zoneOffset() {
6.     final ZoneOffset zoneOffset = ZoneOffset.of("+06:00");
7.     Assert.assertEquals("+06:00", zoneOffset.toString());
8.     Assert.assertEquals(21600, zoneOffset.getTotalSeconds());
9. }
```

Source.

II-C-2-c - ZonedDateTime

La classe **ZonedDateTime** représente une date avec un fuseau horaire au format ISO-8601 (par exemple 2012-05-26T10:15:30+02:00 Europe/Paris).

```
1. /*
2.  * org.isk.datetime.HumanTimeTest
3.  */
4. @Test
5. public void zonedDateTime() {
6.     final LocalDateTime localDateTime = LocalDateTime.parse("2013-05-26T10:22:17");
7.     final ZoneId zoneId = ZoneId.of("Europe/Paris");
8.     final ZonedDateTime zonedDateTime = ZonedDateTime.of(localDateTime, zoneId);
9.     Assert.assertEquals("2013-05-26T10:22:17+02:00[Europe/Paris]", zonedDateTime.toString());
10.
11.     final ZoneOffset zoneOffset = ZoneOffset.from(zonedDateTime);
12.     Assert.assertEquals("+02:00", zoneOffset.toString());
13. }
```

Source.

II-C-2-d - OffsetDateTime

La classe **OffsetDateTime** représente une date avec un offset par rapport à Greenwich, au format ISO-8601 (par exemple 2012-05-26T10:15:30+02:00) sans indication de localité.

Les classes ZonedDateTime et OffsetDateTime sont similaires, elles représentent des dates avec des offsets par rapport à Greenwich. Cependant, la classe ZonedDateTime permet d'identifier certaines ambiguïtés. Par exemple, lors d'un changement d'heure, une même heure peut apparaître deux fois à une heure de décalage. La classe OffsetDateTime ne prend pas en compte ce cas, contrairement à ZonedDateTime.

```
1. /*
2.  * org.isk.datetime.HumanTimeTest
3.  */
4. @Test
5. public void offsetDateTime() {
6.     final LocalDateTime localDateTime = LocalDateTime.parse("2013-05-26T10:22:17");
7.     final ZoneOffset zoneOffset = ZoneOffset.of("+02:00");
8.     final OffsetDateTime offsetDateTime = OffsetDateTime.of(localDateTime, zoneOffset);
9.     Assert.assertEquals("2013-05-26T10:22:17+02:00", offsetDateTime.toString());
10. }
```

Source.

Les méthodes of(), from() et with() sont comparables à celles que nous avons vues dans la partie précédente.

III - Conclusion

La nouvelle API Date and Time surmonte divers problèmes des anciennes APIs Date and Time. Elle est organisée autour du package principal java.time et de quatre sous-packages. Même si nous utiliserons le plus souvent les classes Instant, Duration, LocalDate, LocalTime, LocalDateTime, ZoneId, ZoneOffset, ZonedDateTime, et OffsetDateTime, il existe d'autres types qui méritent notre attention.

IV - Remerciements

Cet article a été publié avec l'aimable autorisation de la société **Soat**.

Nous tenons à remercier **ced** pour sa relecture attentive de cet article et **Mickaël Baron** pour la mise au gabarit.