Les dates avec java 8

Introduction

Java 8 propose une nouvelle API `java.time` dans laquelle on trouve tout un ensemble de classes permettant de gérer les dates et les heures. Toutes ces classes sont finales, immuables et threadsafe.

Pour utiliser ces classes, il faut distinguer le « temps absolu » du « temps humain ».

Le temps absolu est défini de façon unique pour tous les utilisateurs du monde. En java 8, il correspond au nombre de nanosecondes écoulées depuis l'Unix Epoch (1er janvier 1970 à minuit au méridien de Greenwich).

Le « temps humain » correspond à la manière dont on utilise les heures et les dates de la façon usuelle et plus ou moins « locale ».

Les interfaces implémentées

En consultant la javadoc (java.time), on constate que toutes ces classes implémentent `Serializable` et `Comparable<…>`. Par défaut, la comparaison se fait de façon chronologique.

Créer une « date/heure »

Les nouvelles classes de l'API ne proposent pas de constructeurs. Pour créer un objet d'une de ces classes, il faut passer par des méthodes statique (par exemple, `now()` qui donne la « date/heure » correspondant à la « date/heure » actuelle).

La classe Instant

La classe `Instant` représente ce qu'on a appelé le « temps absolu ».

La classe Duration également fournie dans la nouvelle API représente une durée entre deux instants. Elle propose entre autres la méthode statique `between(…,…)` permettant de calculer la durée entre deux instants.

La classe LocalDate

La classe `LocalDate` permet de manipuler des dates usuelles (c.-à-d. exprimées sous la forme année, mois et jour).

On y trouve des méthodes pour :

- récupérer le jour, le mois ou l'année,… ;
- ajouter ou retirer un certain nombre de jours, semaines, mois, années ;
- …

On peut aussi obtenir l'écart entre deux dates (méthode `until(…)`) soit sous forme d'une `Period` (écart exprimé en année, mois, jour), soit en fonction d'une unité à préciser.

La classe LocalTime

La classe `LocalTime` permet de manipuler des heures de la journée (exprimées en heures (comprise entre 0 et 23), minutes, secondes et nanosecondes). Elle offre des méthodes analogues à celles de la classe `LocalDate` excepté qu'on ne peut calculer l'écart entre deux heures sous forme de `Period`. Par contre, comme pour la classe Instant, on peut calculer la durée entre deux `LocalTime` à l'aide de la méthode statique `between` de la classe `Duration`.

La classe LocalDateTime

La classe `LocalDateTime` permet quant à elle de manipuler un objet contenant une date et une heure toujours sans préciser le fuseau horaire dans lequel on se situe. Elle regroupe la plupart des fonctionnalités des classes `LocalDate` et `LocalTime`. Cependant, on ne peut calculer l'écart entre deux `LocalDateTime` qu'en fonction d'une unité à préciser. Il faut aussi être prudent car cette classe ne tient pas compte du changement d'heure.

La classe ZonedDateTime

La classe `ZonedDateTime` permet de manipuler un objet contenant une date et une heure en précisant le fuseau horaire. Cette classe tient compte de particularités locales telles que le changement d'heure éventuel (hiver/été). Cette classe offre en outre les mêmes fonctionnalités que la classe `LocalDateTime`.

Chaque fuseau horaire a un indicateur de type String. Cet indicateur est encapsulé dans un objet de type `ZoneId`. Il existe plusieurs méthodes statiques permettant de récupérer un objet de type `ZoneId`. Par exemple, la méthode staticque `systemDefault()` permet de récupérer le fuseau horaire du système et la méthode statique `of(String zoneId)` permet de créer un fuseau horaire sur base de l'identificateur (par exemple, "Europe/Paris"). La méthode statique `getAvailableZoneIds()` permet de récupérer tous les identificateurs autorisés.

La classe OffsetDateTime

La classe `OffsetDateTime` permet de manipuler un objet contenant une date et une heure en précisant le décalage horaire (positif ou négatif) par rapport au méridien de Greenwich. L'information concernant le décalage horaire est gardée dans un objet de type `ZoneOffset`.

Formatage des dates

La classe `DateTimeFormatter` fournit 3 types de formateurs pour afficher des « dates/heures » :

- Des formateurs standards prédéfinis (fournis par des constantes) utilisés en général pour les échanges entre logiciels.
- Des formateurs locaux prédéfinis permettant d'avoir des présentations plus usuelles des dates en tenant compte de spécificités locales. Les méthodes statiques `ofLocalizedDate`, `ofLocalizedTime` et `ofLocalizedDateTime` permettent de créer de tels formateurs. Il faut préciser en paramètre le `FormatStyle` (style) à utiliser. Il existe 4 styles prédéfinis que ce soit pour les heures ou les dates : `SHORT`, `MEDIUM`, `LONG` et `FULL`.

- Des formateurs suivant un pattern à définir. De tels formateurs sont créés à l'aidez de la méthode statique `ofPattern(String pattern)`.

Une fois qu'on possède un objet de classe `DateTimeFormatter`, l'appel de la méthode `format` sur cet objet avec en paramètre la date/heure à formater renvoie une chaîne de caractères au format voulu.

Le programme :

```
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.ZoneId;
import java.time.format.DateTimeFormatter;
import java.time.format.FormatStyle;

public class FormatDate {
   public static void main(String[] args) {
      LocalDate dateActuelle =  LocalDate.now();
      DateTimeFormatter dateFormater1 =
            DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL);
      System.out.println(dateFormater1.format(dateActuelle));
      DateTimeFormatter dateFormater2 =
            DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);
      System.out.println(dateFormater2.format(dateActuelle));
      LocalTime timeActuel = LocalTime.now();
      DateTimeFormatter timeFormater =
            DateTimeFormatter.ofLocalizedTime(FormatStyle.MEDIUM);
      System.out.println(timeFormater.format(timeActuel));
      LocalDateTime dateTimeActuelle = LocalDateTime.now();
      DateTimeFormatter dateTimeFormater1 =
            DateTimeFormatter.ofPattern("dd MMMM yyyy HH:mm:ss");
      System.out.println(dateTimeFormater1.format(dateTimeActuelle));
      DateTimeFormatter dateTimeFormater2 =
            DateTimeFormatter.ofLocalizedDateTime(FormatStyle.LONG);
      System.out.println(dateTimeFormater2.format(
            dateTimeActuelle.atZone(ZoneId.systemDefault())));
   }
}
```

aura pour sortie (par exemple) :

```
lundi 2 février 2015
2/02/15
14:49:59
02 février 2015 14:49:59
2 février 2015 14:49:59 CET
```

Remarque : certains formateurs ont besoin de connaître le fuseau horaire et certaines particularités locales afin de pouvoir formater la date/heure (comme c'est le cas pour le dernier formateur du programme).

Les dates avec java 7

Introduction

La gestion des dates se fait via la classe `java.util.GregorianCalendar` et l'interface `java.util.Calendar`. La classe `java.util.Date` est presque entièrement dépréciée.

Les interfaces implémentées

En consultant la javadoc, on remarque que `Calendar` implémente `Serializable`, `Cloneable et Comparable<Calendar>`.

Par défaut la comparaison se fait de façon croissante. Par exemple, le 2/03/2013 est plus grand que le 1/04/2012.

Formater des Dates

Pour afficher une date, il est souhaitable de formater l'affichage. Pour faire cela, on emploie `java.text.DateFormat` qui fournit entre autre les méthodes :

```
static DateFormat getDateInstance(int style);

static DateFormat getTimeInstance(int style);

static DateFormat getDateTimeInstance(int dateStyle, int timeStyle);
```

On les utilisera respectivement pour formater une date, une heure ou les deux. Les paramètres de style seront l'une des constantes `DateFormat.SHORT`, `DateFormat.MEDIUM` ou `DateFormat.LONG`.

Le formatage proprement dit se fera en appelant la méthode `format(Date date)` sur l'instance demandée.

Il existe aussi une classe `java.text.DateFormatSymbols` qui permet de définir les valeurs utilisées par les différents styles.

Le programme :

```
import java.text.DateFormat;
import java.util.Date;
import java.util.GregorianCalendar;

public class FormatDate {
     public static void main(String[] args) {
          GregorianCalendar aujoutdhui = new GregorianCalendar();
          Date date = aujou
tdhui.getTime();
          DateFormat df =
DateFormat.getDateTimeInstance(DateFormat.LONG,
DateFormat.SHORT);
```

```
            System.out.println(df.format(date));
      }
}
```

aura pour sortie (par exemple) :

```
19 avril 2004 8:22
```

On peut obtenir le même résultat en utilisant un `SimpleDateFormat`. Dans le code précédent on remplace la ligne

```
      DateFormat df = ...
```

par

```
SimpleDateFormat df = new SimpleDateFormat("dd MMMM yyyy HH:mm");
      …
```