Conheça a nova API de datas do Java 8

Postado dia 08/04/2014 por Alexandre Aquiles e Rodrigo Ferreira em Programação 39

Manipular datas no Java sempre foi algo trabalhoso. No Java 1.0 havia apenas a classe `Date`, que era complicada de usar e não funcionava bem com internacionalização. Com o lançamento do Java 1.1, surgiu a classe abstrata `Calendar`, com muito mais recursos, porém com mutabilidade e decisões de design questionáveis.

A partir do Java 8, que foi lançado recentemente, há uma nova API de datas disponível no pacote java.time. Essa API é uma excelente adição às bibliotecas padrão do Java e já vinha sendo desenvolvida desde 2007.

Essa nova API foi baseada na famosa biblioteca JodaTime, que era a salvação para lidar com datas até então. Já falamos sobre como usar o JodaTime para resolver problemas difíceis. A nova API não é exatamente igual ao JodaTime, já que vários detalhes conceituais e de implementação foram melhorados.

Um dos principais conceitos dessa nova API é a separação de como dados temporais são interpretados em duas categorias: a dos computadores e a dos humanos.

Datas para computadores

Para um computador, o tempo é um número que cresce a cada instante. No Java, historicamente era utilizado um long que representava os milissegundos desde 01/01/1970 às 00:00:00. Na nova API, a classe `Instant` é utilizada para representar esse número, agora com precisão de nanossegundos.

```
Instant agora = Instant.now();
System.out.println(agora); //2014-04-08T10:02:52.036Z (formato ISO-8601)
```

Podemos usar um `Instant`, por exemplo, para medir o tempo de execução de um algoritmo.

```
Instant inicio = Instant.now();
rodaAlgoritmo();
Instant fim = Instant.now();

Duration duracao = Duration.between(inicio, fim);
long duracaoEmMilissegundos =
duracao.toMillis();
```

Observe que utilizamos a classe `Duration`. Essa classe serve para medir uma quantidade de tempo em termos de nanossegundos. Você pode obter essa quantidade de tempo em diversas unidades chamando métodos como `toNanos`, `toMillis`, `getSeconds`, etc.

Datas para humanos

Já para um humano, há uma divisão do tempo em anos, meses, dias, semanas, horas, minutos, segundos e por aí vai. Temos ainda fusos horários, horário de verão e diferentes calendários.

Várias questões surgem ao considerarmos a interpretação humana do tempo. Por exemplo, no calendário judaico, um ano pode ter 13 meses. As classes do pacote `java.time` permitem que essas interpretações do tempo sejam definidas e manipuladas de forma precisa, ao contrário do que acontecia ao usarmos `Date` ou `Calendar`.

Temos, por exemplo, a classe `LocalDate` que representa uma data, ou seja, um período de 24 horas com dia, mês e ano definidos.

```
LocalDate hoje = LocalDate.now();
System.out.println(hoje); //2014-04-08 (formato ISO-
8601)
```

Um `LocalDate` serve para representarmos, por exemplo, a data de emissão do nosso RG, em que não nos importa as horas ou minutos, mas o dia todo.

Podemos criar um `LocalDate` para uma data específica utilizando o método `of`:

```
LocalDate emissaoRG = LocalDate.of(2000, 1,
15);
```

Note que utilizamos o valor 1 para representar o mês de Janeiro. Poderíamos ter utilizado o enum `Month` com o valor JANUARY. Há ainda o enum `DayOfWeek`, que representa os dias da semana.

Para calcularmos a duração entre dois `LocalDate`, devemos utilizar um `Period`, que já trata anos bissextos e outros detalhes.

```
LocalDate homemNoEspaco = LocalDate.of(1961, Month.APRIL, 12);
LocalDate homemNaLua = LocalDate.of(1969, Month.MAY, 25);

Period periodo = Period.between(homemNoEspaco, homemNaLua);

System.out.printf("%s anos, %s mês e %s dias",
   periodo.getYears() , periodo.getMonths(), periodo.getDays());
   //8 anos, 1 mês e 13 dias
```

Já a classe `LocalTime` serve para representar apenas um horário, sem data específica. Podemos, por exemplo, usá-la para representar o horário de entrada no trabalho.

```
LocalTime horarioDeEntrada = LocalTime.of(9, 0);
System.out.println(horarioDeEntrada); //09:00
```

A classe `LocalDateTime` serve para representar uma data e hora específicas. Podemos representar uma data e hora de uma prova importante ou de uma audiência em um tribunal.

```
LocalDateTime agora = LocalDateTime.now();
LocalDateTime aberturaDaCopa = LocalDateTime.of(2014, Month.JUNE, 12, 17,
0);
System.out.println(aberturaDaCopa); //2014-06-12T17:00 (formato ISO-8601)
```

Datas com fuso horário

Para representarmos uma data e hora em um fuso horário específico, devemos utilizar a classe `ZonedDateTime`.

```
ZoneId fusoHorarioDeSaoPaulo = ZoneId.of("America/Sao_Paulo");
ZonedDateTime agoraEmSaoPaulo =
ZonedDateTime.now(fusoHorarioDeSaoPaulo);
```

Com um `ZonedDateTime`, podemos representar, por exemplo, a data de um voo.

```
ZoneId fusoHorarioDeSaoPaulo = ZoneId.of("America/Sao_Paulo");
ZoneId fusoHorarioDeNovaYork = ZoneId.of("America/New_York");

LocalDateTime saidaDeSaoPauloSemFusoHorario =
    LocalDateTime.of(2014, Month.APRIL, 4, 22, 30);
LocalDateTime chegadaEmNovaYorkSemFusoHorario =
    LocalDateTime.of(2014, Month.APRIL, 5, 7, 10);

ZonedDateTime saidaDeSaoPauloComFusoHorario =
    ZonedDateTime.of(saidaDeSaoPauloSemFusoHorario,
fusoHorarioDeSaoPaulo);
System.out.println(saidaDeSaoPauloComFusoHorario); //2014-04-04T22:30-
03:00[America/Sao_Paulo]

ZonedDateTime chegadaEmNovaYorkComFusoHorario =
    ZonedDateTime.of(chegadaEmNovaYorkSemFusoHorario,
fusoHorarioDeNovaYork);
System.out.println(chegadaEmNovaYorkComFusoHorario); //2014-04-05T07:10-
04:00[America/New_York]

Duration duracaoDoVoo =
    Duration.between(saidaDeSaoPauloComFusoHorario,
chegadaEmNovaYorkComFusoHorario);
System.out.println(duracaoDoVoo); //PT9H40M
```

Se calcularmos de maneira ingênua a duração do voo, teríamos 8:40. Porém, como há uma diferença entre os fusos horários de São Paulo e Nova York, a duração correta é 9:40. Repare que a API já faz o tratamento de fusos horários distintos.

Outro cuidado importante que devemos ter é em relação ao horário de verão. No fim do horário de verão, por exemplo, a mesma hora existe duas vezes!

```
ZoneId fusoHorarioDeSaoPaulo = ZoneId.of("America/Sao_Paulo");

LocalDateTime fimDoHorarioDeVerao2013SemFusoHorario =
    LocalDateTime.of(2014, Month.FEBRUARY, 15, 23, 00);

ZonedDateTime fimDoHorarioVerao2013ComFusoHorario =
    fimDoHorarioDeVerao2013SemFusoHorario.atZone(fusoHorarioDeSaoPaulo);
System.out.println(fimDoHorarioVerao2013ComFusoHorario); //2014-02-
15T23:00-02:00[America/Sao_Paulo]

ZonedDateTime maisUmaHora =
```

```
    fimDoHorarioVerao2013ComFusoHorario.plusHours(1);
System.out.println(maisUmaHora); //2014-02-15T23:00-
03:00[America/Sao_Paulo]
```

Repare no código anterior que, mesmo aumentando uma hora, o horário continuou 23:00. Entretanto, observe que o fuso horário foi alterado de -02:00 para -03:00.

Datas e meses importantes

Existem também as classes `MonthDay`, que deve ser utilizada para representar datas importantes que se repetem todos os anos, e `YearMonth`, que deve ser utilizada para representar um mês inteiro de um ano específico.

```
MonthDay natal = MonthDay.of(Month.DECEMBER, 25);
YearMonth copaDoMundo2014 = YearMonth.of(2014, Month.JUNE);
```

Formatando datas

O `toString` padrão das classes da API utiliza o formato ISO-8601. Se quisermos definir o formato de apresentação da data, devemos utilizar o método `format`, passando um `DateTimeFormatter`.

```
LocalDate hoje = LocalDate.now();
DateTimeFormatter formatador =
    DateTimeFormatter.ofPattern("dd/MM/yyyy")
;
hoje.format(formatador); //08/04/2014
```

O enum `FormatStyle` possui alguns formatos pré-definidos, que podem ser combinados com um `Locale`.

```
LocalDateTime agora = LocalDateTime.now();
DateTimeFormatter formatador = DateTimeFormatter
    .ofLocalizedDateTime(FormatStyle.SHORT)
    .withLocale(new Locale("pt", "br"));
agora.format(formatador); //08/04/14 10:02
```

Manipulando datas

Todas as classes mencionadas possuem diversos métodos que permitem manipular as medidas de tempo. Por exemplo, podemos usar o método `plusDays` da classe `LocalDate` para aumentarmos um dia:

```
LocalDate hoje = LocalDate.now();
LocalDate amanha = hoje.plusDays(1);
```

Outro cálculo interessante é o número de medidas de tempo até uma determinada data, que podemos fazer através do método `until`. Para descobrir o número de dias até uma data, por exemplo, devemos passar `ChronoUnit.DAYS` como parâmetro.

```
MonthDay natal = MonthDay.of(Month.DECEMBER, 25);
LocalDate natalDesseAno = natal.atYear(Year.now().getValue());
long diasAteONatal = LocalDate.now()
        .until(natalDesseAno, ChronoUnit.DAYS);
```

Podemos utilizar a interface `TemporalAdjuster` para definir diferentes maneiras de manipular as medidas de tempo. É interessante notar que essa é uma interface funcional, permitindo o uso de lambdas.

A classe auxiliar `TemporalAdjusters` já possui diversos métodos que agem como factories para diferentes implementações úteis de `TemporalAdjuster`. Podemos, por exemplo, descobrir qual é a próxima sexta-feira.

```
TemporalAdjuster ajustadorParaProximaSexta =
TemporalAdjusters.next(DayOfWeek.FRIDAY);
LocalDate proximaSexta = LocalDate.now().with(ajustadorParaProximaSexta);
```

Imutabilidade e Testabilidade

Se você adicionar um dia a um `LocalDate`, as informações de data não serão alteradas.

```
LocalDate hoje = LocalDate.now(); //2014-04-08
hoje.plusDays(1);
System.out.println(hoje); //2014-04-08 (ainda é hoje, e não
amanhã!)
```

Na verdade, qualquer método que alteraria o objeto retorna uma referência a um novo objeto com as informações alteradas.

```
LocalDate hoje = LocalDate.now();
LocalDate amanha = hoje.plusDays(1);
boolean mesmoObjeto = hoje == amanha; //false, já que é
imutável
```

Isso vale para todas as classes do pacote `java.time`, que são imutáveis e, por isso, são thread-safe e mais fáceis de dar manutenção.

Um outro ponto importante da API é a melhor testabilidade com o uso da classe `Clock`.

Trabalhando com código legado

Não poderemos mudar todo o nosso código existente para trabalhar com o poder do pacote `java.time` de uma hora pra outra. Por isso, o Java 8 trouxe alguns pontos de interoperabilidade entre os antigos `Date` e `Calendar` e a nova API.

```
Calendar calendar = Calendar.getInstance();
Instant instantAPartirDoCalendar = calendar.toInstant();
Date dateAPartirDoInstant = Date.from(instantAPartirDoCalendar);
Instant instantAPartirDaDate = dateAPartirDoInstant.toInstant();
```

Além disso, classe abstrata `Calendar` ganhou um builder, que possibilita a criação de uma instância de maneira fluente.

```
Calendar calendario =
        new Calendar.Builder()
           .setDate(2014, Calendar.APRIL, 8)
           .setTimeOfDay(10, 2, 57)
           .setTimeZone(TimeZone.getTimeZone("America/Sao_Paulo"))
           .setLocale(new Locale("pt", "br"))
           .build();
```

A nova API de datas do Java 8 é bastante extensa, possuindo diversos outros recursos interessantes. Com certeza, é uma adição muito bem-vinda ao Java, que vai facilitar bastante o trabalho de manipulação de datas.