07

Crie e manipule dados de calendários usando as classes do pacote java.time

Java 8 - Trabalhando com algumas classes da Java API

Crie e manipule dados de calendários usando as classes java.time.LocalDateTime, java.time.LocalDate, java.time.LocalTime, java.time.format.DateTimeFormatter?, java.time.Period

No Java 8, após uma longa história de sofrimento dos desenvolvedores com as classes `java.util.Date` e `java.util.Calendar`, foi introduzida uma nova API para manipulação de datas e horas.

Vamos entender quais classes e métodos foram incluídos, além de passar pelos detalhes que serão cobrados na prova.

As classes que serão cobradas são:

- `LocalDate` : representa uma data sem hora no formato `yyyy-MM-dd` (ano-mês-dia).
- `LocalTime` : representa uma hora no formato `hh:mm:ss.zzz` (hora:minuto:segundo.milissegundo).
- `LocalDateTime` : representa uma data com hora no formato `yyyy-MM-dd-HH-mm-ss.zzz` (ano-mês-dia-hora-minuto-segundo.milissegundo).
- `MonthDay` : representa um dia e mês, sem o ano.
- `YearMonth` : representa um mês e ano, sem o dia.
- `Period` : representa um período de tempo, em dia, mês e ano.

- `DateTimeFormatter` : classe que possui vários métodos para formatação.

Para utilizar essas classes, é necessário conhecer uma ou outra classe da API de `java.time` que não estão na lista da seção, sendo que também as veremos aqui.

Todas as classes do pacote `java.time` são imutáveis, ou seja, após serem instanciadas, seus valores não podem ser alterados, assim como a classe `String` . Portanto, lembre-se que todos os métodos que parecem modificar os valores das datas retornam novas instâncias com os valores alterados, enquanto o objeto original segue inalterado.

Criando objetos de data

Todas as classes que representam datas têm métodos similares para criação, como nos exemplos a seguir:

```
LocalTime currentTime = LocalTime.now(); // 09:05:03.244
LocalDate today = LocalDate.now(); // 2014-12-10
LocalDateTime now = LocalDateTime.now();
                        // 2014-12-10-09-05-03.244
```

COPIAR CÓDIGO

É possível escolher o fuso horário que será usado na criação das datas, passando como parâmetro para o método `now` um objeto do tipo `ZoneId` :

```
LocalTime time = LocalTime.now(ZoneId.of("America/Chicago"));
LocalDate date =
LocalDate.now(ZoneId.of("America/Sao_Paulo"));
```

```
LocalDateTime dateTime =
    LocalDateTime.now(ZoneId.of("America/Los_Angeles"));
```

COPIAR CÓDIGO

Caso queira representar uma data ou hora específica, usamos o método `of`. Cada um desses métodos possui versões sobrecarregadas, recebendo mais ou menos valores iniciais. Todos os tipos de datas que contêm meses possuem versões de `of`, que recebem tanto números inteiros quanto valores do ::enum:: `Month`.

Por exemplo, podemos criar uma representação do meio-dia:

```
LocalTime noon = LocalTime.of(12, 0);
```

COPIAR CÓDIGO

Para criar o natal de 2014 e de 2015:

```
LocalDate christmas2014 = LocalDate.of(2014, 12, 25);
LocalDate christmas2015 = LocalDate.of(2015, Month.DECEMBER,
25);
```

COPIAR CÓDIGO

Podemos representar qualquer natal:

```
MonthDay someChristmas = MonthDay.of(Month.DECEMBER, 31);
```

COPIAR CÓDIGO

Ainda com o método `of`, podemos criar um momento exato no tempo:

```
LocalDateTime someDate =
            LocalDateTime.of(2017, Month.JANUARY, 25, 13, 45);
```

COPIAR CÓDIGO

Ou ainda passar um dia e somente adicionar o horário:

```
LocalDate christmas2014 = LocalDate.of(2014, 12, 25);
LocalDateTime christmasAtNoon =
    LocalDateTime.of(christmas2014, meioDia);
```

COPIAR CÓDIGO

Passar um valor inválido para qualquer um dos campos (mês 13, por exemplo) lançará um `DateTimeException`.

Manipulando datas

Uma das decisões de ::design:: que guia a nova API de datas é a padronização dos nomes de métodos que têm o mesmo comportamento. Os nomes mais comuns são:

- `get`: obtém o valor de algo
- `is`: verifica se algo é verdadeiro
- `with`: lembra um ::setter::, mas retorna um novo objeto com o valor alterado

- `plus` : soma alguma unidade ao objeto, retorna um novo objeto com o valor alterado
- `minus` : subtrai alguma unidade do objeto, retorna um novo objeto com o valor alterado
- `to` : converte um objeto de um tipo para outro
- `at` : combina um objeto com outro

Veremos agora esses métodos na prática.

Extraindo partes de uma data

Para obter alguma porção de uma data, podemos usar os métodos precedidos por `get` :

```
LocalDateTime now = LocalDateTime.of(2014,12,15,13,0);
System.out.println(now.getDayOfMonth()); // 15
System.out.println(now.getDayOfYear();  // 349
System.out.println(now.getHour();       // 13
System.out.println(now.getMinute();     // 0
System.out.println(now.getYear();       // 2014
System.out.println(now.getDayOfWeek();  // MONDAY
System.out.println(now.getMonthValue(); // 12
System.out.println(now.getMonth();      // DECEMBER
```

COPIAR CÓDIGO

Além desses métodos, temos um método `get()` , que recebe como parâmetro uma implementação da interface `TemporalField` , geralmente `ChronoField` , e retorna um inteiro. Note que como estamos falando de um campo que será

retornado. Usamos `ChronoField`, um campo de tempo cujo valor queremos saber:

```
LocalDateTime now = LocalDateTime.of(2014,12,15,13,0);
// 15
System.out.println(now.get(ChronoField.DAY_OF_MONTH));
// 349
System.out.println(now.get(ChronoField.DAY_OF_YEAR));
// 13
System.out.println(now.get(ChronoField.HOUR_OF_DAY));
// 0
System.out.println(now.get(ChronoField.MINUTE_OF_HOUR));
// 2014
System.out.println(now.get(ChronoField.YEAR));
// 1 (MONDAY)
System.out.println(now.get(ChronoField.DAY_OF_WEEK));
// 12
System.out.println(now.get(ChronoField.MONTH_OF_YEAR));
```

COPIAR CÓDIGO

Nem todas as classes possuem todos os métodos. Por exemplo, objetos do tipo `LocalDate` não possuem a parte de horas, logo não há métodos como `getHour`. É necessário ficar atento ao tipo do objeto e a qual método está sendo chamado:

```
LocalDate d = LocalDate.now();
d.getHour(); //compile error, method not found.
```

COPIAR CÓDIGO

Comparações entre datas

Usamos os métodos que começam com `is` para realizar comparações entre as datas:

```
MonthDay day1 = MonthDay.of(1, 1); //01/jan
MonthDay day2 = MonthDay.of(1, 2); //02/jan

System.out.println(day1.isAfter(day2)); //false
System.out.println(day1.isBefore(day2)); //true
```

COPIAR CÓDIGO

Além de métodos de comparação, também existem aqueles para indicar se alguma porção da data é suportada pelo objeto:

```
LocalDate aprilFools = LocalDate.of(2015, 4, 1);
LocalDate foolsDay = LocalDate.of(2015, 4, 1);
// são equals?
System.out.println(aprilFools.isEqual(foolsDay)); //true
// este objeto suporta dias?
System.out.println(aprilFools.isSupported(
    ChronoField.DAY_OF_MONTH)); //true
// este objeto suporta horas?
System.out.println(aprilFools.isSupported(
    ChronoField.HOUR_OF_DAY)); //false
// posso fazer operações com dias?
System.out.println(aprilFools.isSupported(ChronoUnit.DAYS));
//true
// posso fazer operações com horas?
System.out.println(aprilFools.isSupported(ChronoUnit.HOURS));
//false
```

COPIAR CÓDIGO

Alterando as datas

Todos os objetos da nova API de datas são imutáveis, ou seja, não podem ter o seu valor alterado após a criação. Mas existem alguns que podem ser utilizados para obter versões modificadas destes objetos. Vamos começar com o método `with`, que é como um ::setter::, mas retornando um novo objeto em vez de alterar o valor do objeto atual:

```
LocalDate d = LocalDate.of(2015, 4, 1); //2014-04-01

d = d.withDayOfMonth(15).withMonth(3); //chaining
System.out.println(d); //2015-03-15
```

COPIAR CÓDIGO

Cada método `with` chamado retorna um novo objeto, com o valor modificado. O objeto original nunca tem seu valor alterado:

```
LocalDate d = LocalDate.of(2013, 9, 7);
System.out.println(d); // 2013-09-07
d.withMonth(12);
System.out.println(d); // 2013-09-07
```

COPIAR CÓDIGO

Lembre se que só é possível manipular partes da data em objetos que têm estas partes. O exemplo a seguir não compila pois "LocalTime does not have a day of month field".

```
LocalTime d = LocalTime.now();
```

```
d.withDayOfMonth(15); // compile error
```

COPIAR CÓDIGO

Caso o objetivo seja incrementar ou decrementar alguma parte da data, temos os métodos `plus` e `minus`:

```
LocalDate d = LocalDate.of(2013, 9, 7);
d = d.plusDays(1).plusMonths(3).minusYears(2);
System.out.println(d); // 2011-12-08
```

COPIAR CÓDIGO

Podemos adicionar os mesmos três meses usando uma `ENUM` de unidade de tempo. Cuidado que não estamos falando para alterar o campo `ChronoField.WEEK`. Isso seria errado, pois não queremos alterar um campo de semana, ou mesmo de dia. Queremos somar uma unidade de tempo, isto é, a API tem que se virar sozinha para adicionar os dias, meses etc. de acordo com o que nós pedirmos, mesmo se for ano bissexto etc. Portanto, não falamos o campo que desejamos alterar mas, sim, a unidade, `ChronoUnit`:

```
LocalDate d = LocalDate.of(2013, 9, 7);
d = d.plusWeeks(3).minus(3, ChronoUnit.WEEKS);
System.out.println(d); // 2013-09-07
```

COPIAR CÓDIGO

Para fixar se é `ChronoField` ou `ChronoUnit`, lembre-se que você deseja saber (`get`) o valor de um campo, então `ChronoField`, `.DAY` para o dia específico. Caso você deseje adicionar dias, para adicionar N dias usa-se `ChronoUnit.DAYS` (plural).

Caso você tente manipular uma data usando uma unidade não suportada, será lançada a exception `UnsupportedTemporalTypeException`:

```
LocalDate d = LocalDate.of(2013, 9, 7);
                        // UnsupportedTemporalTypeException
                        //LocalDate não suporta horas!
d = d.plus(3, ChronoUnit.HOURS);
System.out.println(d);
```

COPIAR CÓDIGO

A classe MonthYear

Atenção, a classe `MonthYear` não possui nenhum método para somar ou subtrair unidades de tempo, logo, ela não tem nenhum método `plus` ou `minus`, e também não possui um método `isSupported` que receba `ChronoUnit`.

Convertendo entre os diversos tipos de datas

A classe `LocalDateTime` possui métodos para converter esta data/hora em objetos que só possuem a data (`LocalDate`) ou que só possuem a hora (`LocalTime`):

```
LocalDateTime now = LocalDateTime.now();
LocalDate dateNow = now.toLocalDate(); // de DateTime para
Date
LocalTime timeNow = now.toLocalTime(); // de DateTime para
Time
```

COPIAR CÓDIGO

As classes também possuem métodos para combinar suas partes e criar um novo objeto modificado:

```
LocalDateTime now = LocalDateTime.now();
LocalDate dateNow = now.toLocalDate(); // de DateTime para
Date
LocalTime timeNow = now.toLocalTime(); // de DateTime para
Time

// de Date para DateTime
LocalDateTime nowAtTime1 = dateNow.atTime(timeNow);
// de Time para DateTime
LocalDateTime nowAtTime2 = timeNow.atDate(dateNow);
```

COPIAR CÓDIGO

Trabalhando com a API legada

Para tornar a API legada compatível com a API nova, foram introduzidos vários métodos nas antigas classes `java.util.Date` e `java.util.Calendar` .

O exemplo a seguir converte uma `java.util.Date` em `LocalDateTime` usando a Timezone padrão do sistema:

```
Date d = new Date();
Instant i = d.toInstant();
LocalDateTime ldt1 =
    LocalDateTime.ofInstant(i, ZoneId.systemDefault());
```

COPIAR CÓDIGO

O próximo exemplo transforma um `Calendar` pelo mesmo processo:

```
Calendar c = Calendar.getInstance();
Instant i = c.toInstant();
LocalDateTime ldt2 = LocalDateTime.ofInstant(i,

ZoneId.systemDefault());
```

COPIAR CÓDIGO

Repare que para fazer a conversão usamos como intermediário a classe `Instant`, que representa o número de milissegundos desde `01/01/1970`. Também podemos usar essa classe para fazer o caminho de volta:

```
Date d = new Date();
Instant i = d.toInstant();
LocalDateTime ldt1 =
    LocalDateTime.ofInstant(i, ZoneId.systemDefault());

Instant instant = ldt1.toInstant(ZoneOffset.UTC);
Date date = Date.from(instant);
```

COPIAR CÓDIGO

Cálculos de intervalo de tempo com datas

Quando necessitamos realizar algum tipo de cálculo envolvendo duas datas, podemos usar as classes `Duration`, `Period` e o método `between` da classe `ChronoUnit`:

`Duration` é a classe de mais baixo nível, usada para manipular objetos do tipo `Instant`. O exemplo a seguir soma 10 segundos ao instante atual:

```
Instant now = Instant.now(); // agora
Duration tenSeconds = Duration.ofSeconds(10); // 10 segundos
Instant t = now.plus(tenSeconds); // agora mais 10 segundos
```

COPIAR CÓDIGO

O próximo exemplo mostra como pegar o intervalo em segundos entre dois instantes:

```
Instant t1 = Instant.EPOCH; // 01/01/1970 00:00:00
Instant t2 = Instant.now();
long secondsSinceEpoch = Duration.between(t1,
t2).getSeconds();
```

COPIAR CÓDIGO

Note que `Duration` só tem a opção de `getSeconds`; não existem métodos do tipo `getDays` etc.

`ChronoUnit` é uma das classes mais versáteis, pois permite ver a diferença entre duas datas em várias unidades de tempo:

```
LocalDate birthday = LocalDate.of(1983, 7, 22);
LocalDate base = LocalDate.of(2014, 12, 25);

// 31 anos no total
System.out.println(ChronoUnit.YEARS.between(birthday, base));
```

```
// 377 meses no total
System.out.println(ChronoUnit.MONTHS.between(birthday, base));
// 11479 dias no total
System.out.println(ChronoUnit.DAYS.between(birthday, base));
```

COPIAR CÓDIGO

Já a classe `Period` pode ser usada para fazer cálculos de intervalos, quebrando as unidades de tempo do maior para o menor. Vamos tentar calcular a idade de uma pessoa:

```
LocalDate birthday = LocalDate.of(1983, 7, 22);
LocalDate base = LocalDate.of(2014, 12, 25);

Period lifeTime = Period.between(birthday, base);

System.out.println(lifeTime.getYears()); // 31 anos
System.out.println(lifeTime.getMonths()); // 5 meses
System.out.println(lifeTime.getDays()); // 3 dias
```

COPIAR CÓDIGO

Formatando e convertendo em texto

Para formatar a impressão de nossas datas, usamos a classe `DateTimeFormatter`, do pacote `java.time.format`. Ele segue o mesmo padrão da clássica `SimpleDateFormat`.

```
LocalDate birthday = LocalDate.of(1983, 7, 22);
```

```
DateTimeFormatter formatter =
    DateTimeFormatter.ofPattern("yyyy MM dd");
System.out.println(formatter.format(birthday)); // 1983 07 22
```

COPIAR CÓDIGO

Também podemos passar o formatador como parâmetro para o método `format` dos objetos de data:

```
LocalDate birthday = LocalDate.of(1983, 7, 22);
DateTimeFormatter formatter =
    DateTimeFormatter.ofPattern("yyyy MM dd");
System.out.println(birthday.format(formatter)); // 1983 07 22
```

COPIAR CÓDIGO

Já para transformar um texto em uma data, usamos o `DateTimeFormater` juntamente com o método `parse` da classe que desejamos instanciar:

```
DateTimeFormatter formatter =
    DateTimeFormatter.ofPattern("dd/MM/yyyy");
LocalDate d = LocalDate.parse("23/04/1986",formatter);
System.out.println(formatter.format(d)); // 23/04/1986
```

COPIAR CÓDIGO

Caso usemos algum caractere não suportado ou passemos uma data no formato inválido, será lançada uma `DateTimeParseException` :

```
DateTimeFormatter formatter =
    DateTimeFormatter.ofPattern("dd/MM/yyyy");
LocalDate d =
    LocalDate.parse("23/15/1986",formatter);
    // throws DateTimeParseException
System.out.println(formatter.format(d)); // 23/04/1986
```

COPIAR CÓDIGO