Datas e horas

Conceitos fundamentais e as APIs do Java

Casa do Código

HUGO KOTSUBO

Sumário

ISBN

Impresso e PDF: 978-85-7254-005-6

EPUB: 978-85-7254-006-3

MOBI: 978-85-7254-007-0

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

Agradecimentos

A Nathalia, eterna companheira, por todo o apoio, paciência e compreensão, durante o longo processo de escrita deste livro.

A Stephen Colebourne, por ter criado duas das minhas APIs de data favoritas.

E a Arthur David Olson, pois sem ele estaríamos literalmente perdidos no tempo.

Sobre o autor

Hugo Kotsubo é Bacharel em Ciência da Computação pela USP e desenvolvedor Java desde 2000, com foco em aplicações web. Possui certificação SCJP e atualmente trabalha com Java no front-end e às vezes no back-end. Também flerta com outras linguagens (Python, Ruby, Shell Script, Scala) e é fascinado pela complexidade - nem sempre aparente - que envolve datas e horas.

Já escreveu muito código para lidar com datas, e entre um bug e outro, aprendeu bastante a respeito disso. Código que só falha no horário de verão, ou que depende do fuso horário configurado no servidor, entre outras situações estranhas que só as datas e horas nos proporcionam, são ao mesmo tempo fontes de inspiração e raiva, de motivação para entender mais e indignação por serem tão desnecessariamente complicadas.

Prefácio

Toda aplicação precisa, em algum momento, lidar com datas. Seja com os casos mais simples, como uma data de nascimento no cadastro de usuários, até os mais complexos, como um rastreamento completo de todos os eventos ocorridos com determinado produto, desde a sua confecção até a venda, com data e hora exatas em que cada etapa ocorreu, além de geração de relatórios das vendas do último mês, e assim por diante.

Inicialmente, pensamos ser algo simples, afinal, datas e horas são coisas tão banais no cotidiano que nem nos damos conta da complexidade escondida por trás destes conceitos. E é aí que mora o perigo. Ao tratar algo complexo com uma abordagem simplista, corremos o risco de não pensar em todos os erros que podem acontecer. Simplesmente porque nem sequer imaginamos que eles possam ocorrer.

Um exemplo: imagine uma rede varejista que tem sua matriz em São Paulo, mas possui um e—commerce que entrega em todo o Brasil. Suponha que um gerente pediu um relatório de todas as compras feitas no site, em janeiro de 2018, em todo o país. Uma das compras foi feita por um cliente que estava em Manaus, no dia 31 de janeiro de 2018, às 23:00 (horário local). Porém, esta compra não aparece no relatório.

Depois de muita investigação, descobre-se que a aplicação que extraiu o relatório está convertendo todas as datas para o fuso horário de São Paulo. E 31 de janeiro de 2018, às 23:00 em Manaus é equivalente a 1º de **fevereiro** de 2018, às 01:00 em São Paulo (lembre-se de que São Paulo estava em horário de verão neste dia, daí a diferença de 2 horas). Por isso, aquela compra não está aparecendo no relatório de janeiro.

Muitas vezes nem é culpa de quem desenvolve - ao menos não diretamente - pois, dependendo da API utilizada, essas conversões acontecem internamente, "por debaixo dos panos", sem que o

desenvolvedor sequer perceba. E podemos até debater quem é o real culpado: quem desenvolve, por não conhecer os detalhes intrínsecos que envolvem a programação com datas, ou os criadores da API utilizada, por não terem feito algo mais fácil de usar e entender? Mas o fato é que no fim das contas o problema deve ser corrigido. Claro que há uma decisão de negócio envolvida (considerar o horário do cliente ou o horário de São Paulo?), mas, uma vez tomada esta decisão, o desenvolvedor deve saber como alterar seu código para que siga a regra desejada.

Para saber exatamente o que fazer, é essencial ter um conhecimento maior sobre todos os conceitos e detalhes envolvendo datas e horas. Sem saber como funcionam os fusos horários e demais conceitos, e como sua linguagem ou framework trabalha com eles, não é possível escrever um código confiável. Como eu sei que estou considerando este ou aquele fuso horário? Como meu código vai se comportar no exato momento em que ocorre a mudança para o horário de verão? Como eu sei que estou usando a data correta?

Este livro tratará estas e muitas outras questões envolvendo datas. A primeira parte explicará todos os conceitos importantes do ponto de vista de quem desenvolve o software. Portanto, serão desconsiderados conceitos como relatividade e outros não diretamente relacionados à manipulação de datas em aplicações. Ao mesmo tempo, essa parte terá algo a oferecer a todos que trabalham com desenvolvimento, já que não foca em nenhuma tecnologia ou linguagem específica.

No restante do livro cobriremos as APIs da linguagem Java, começando com `java.util.Date` e `java.util.Calendar` , e por fim a API `java.time` , introduzida no Java 8, e que é **muito** superior à API antiga, em vários aspectos que serão explicados ao longo do livro.

Muitos argumentam que, por existir uma nova API, não é mais necessário aprender `Date` e `Calendar` . Apesar de concordar que as aplicações devam usar o `java.time` sempre que possível, ainda há

muito código legado e bibliotecas antigas — porém muito utilizadas — que dependem de `Date` e `Calendar`, e é importante saber como elas funcionam para que a migração seja feita da forma correta. Por serem limitadas e cheias de problemas, há muito código que usa estas classes de maneiras não convencionais – justamente para tentar contornar suas limitações, ou até mesmo por desconhecimento – ou que não se comportam de maneira muito intuitiva, por isso nem sempre é trivial migrar para a nova API.

E se você não programa em Java, ainda assim poderá aproveitar um pouco da segunda e terceira partes do livro. Apesar de terem muito código, estes capítulos também aprofundarão alguns conceitos e discutirão casos de uso e situações práticas que não aparecem na primeira parte.

Pré-requisitos e convenções

A primeira parte deste livro é conceitual e não depende de nenhuma tecnologia específica, portanto não há nenhum conhecimento técnico prévio como pré-requisito. Para o restante, é necessário o mínimo de familiaridade com Java, para que os códigos possam ser entendidos. Assume-se que você saiba pelo menos o básico da linguagem.

Conforme já foi dito, se você não programa em Java, ainda assim poderá aproveitar um pouco da segunda e terceira partes do livro. Haverá bastante código, mas também terá algumas explicações conceituais, aprofundando ou reforçando o que foi visto na primeira parte.

A maior parte dos códigos deste livro foi testada com o JDK `1.8.0_131`. Quando não houver uma indicação explícita da versão utilizada, deve-se assumir que foi esta. Há alguns códigos que foram testados nas versões `1.7.0_51` e `10.0.1`. Se houver um exemplo específico que foi testado em alguma destas versões, isto será indicado no texto.

As JVMs possuem as seguintes configurações:

```
-Duser.country=BR -Duser.language=pt -Duser.timezone=America/Sao_Paulo
```

Isso quer dizer que o *locale* padrão, retornado por `java.util.Locale.getDefault()`, é `pt_BR` (Português Brasileiro), e o timezone padrão, retornado por `java.util.TimeZone.getDefault()`, é `America/Sao_Paulo`. Ao longo do livro será explicado como estas configurações podem afetar o comportamento das APIs.

Todo o código está disponível no GitHub:
https://github.com/hkotsubo/java-datetime-book/

Convenções

Muitas classes não serão chamadas o tempo todo pelo seu *fully qualified name* (ou "nome completo"). Exemplos: `java.util.Date` e `java.util.Calendar` serão chamadas de `Date` e `Calendar` , a menos que haja algum motivo para usar o nome completo, por exemplo para diferenciar de `java.sql.Date` .

As duas APIs a serem explicadas serão algumas vezes chamadas de "API antiga" e "API nova", entre outros nomes, conforme tabela a seguir:

Nome da API	Outros nomes	Classes
`java.time`	API nova, JSR-310	Todo o pacote `java.time`
`Date`	API antiga, `Calendar` , API legada	`java.util.Date` , `java.util.Calendar` , `java.text.SimpleDateFormat` , `java.util.TimeZone` , mais as classes `Date` , `Time` e `Timestamp` do pacote `java.sql`

Vários exemplos usarão código que retorna a data e/ou a hora atual. Porém, esta é uma informação que muda o tempo todo e os valores ficariam diferentes a cada vez que o código rodasse. Por isso, assume-se que a data/hora atual corresponde a **4 de maio de 2018, às 17:00 em São Paulo**. É um valor arbitrário, sem nenhum significado especial, usado apenas para que os exemplos tenham um mesmo valor e não fiquem confusos. Quando o valor retornado for diferente, será informado no texto.

Além disso, alguns termos em inglês serão usados, por não haver um equivalente em português, ou quando a tradução não for adequada. Quando cada termo for introduzido, haverá uma explicação e justificativa para esta escolha.

Conceitos fundamentais de data e hora

Em um primeiro momento, todos pensam que programar com datas é algo fácil, mas infelizmente não é bem assim. Data e hora em programação é um dos domínios mais difíceis de se lidar, e mais difícil ainda é aceitar e entender este fato - sei que provavelmente você está achando que estou exagerando, só para "vender meu peixe", mas infelizmente datas são mais difíceis do que deveriam.

Afinal, usamos datas e horas frequentemente no nosso dia a dia, e de maneira tão natural, que nunca temos problemas com elas. Parece algo tão simples, que nem nos damos conta da complexidade existente, e simplesmente assumimos que vai ser fácil e saímos codificando sem parar para pensar no assunto.

E muitas APIs não ajudam, pois também partem desta mesma premissa e muitas vezes assumem abordagens simplificadas que nem sempre funcionam da maneira correta.

Esta primeira parte pretende explicar em detalhes todos os conceitos envolvendo datas e horas, e mostrar toda a complexidade que se esconde por trás delas do ponto de vista da programação. Ao final, espero que você consiga entender melhor por que aquele seu código que mexe com datas não funciona direito no horário de verão, ou por que o cálculo automático da idade às vezes retorna um ano a mais (ou a menos), entre outras situações estranhas que só as datas são capazes de nos proporcionar.

Talvez não seja possível fornecer uma receita que resolva exatamente aquele bug específico no seu código – e nem tenho a pretensão de resolver todos os problemas do mundo da programação – mas espero que, pelo menos, você consiga entender por que aquele erro ocorre, como debugá-lo melhor para saber exatamente qual data está sendo utilizada, em vez de usar alguma

gambiarra aleatória sem dominar o que de fato está acontecendo, e com isso, que você esteja mais apto a resolvê-lo.

Capítulo 1
Que dia é hoje? Que horas são?

Conforme você vai perceber ao longo do livro, a resposta para estas – e muitas outras perguntas – é "depende". Apesar de não parecer, datas e horas são mais complicadas do que parecem e, como são coisas tão comuns no nosso dia a dia, não paramos para pensar na complexidade por trás delas.

Um dos motivos é que sempre pensamos em data e hora do ponto de vista local. Por exemplo, algumas respostas válidas para as perguntas acima seriam:

- Que dia é hoje? 4 de maio de 2018.
- Que horas são? 5 da tarde.

Quando eu pergunto "que dia é hoje" ou "que horas são", sempre está implícito que eu me refiro ao local onde estou – a propósito, estou em São Paulo, então as respostas acima referem-se ao fuso horário usado nesta cidade. Mas se neste mesmo instante eu perguntar para um amigo meu que mora na Califórnia, ele responderá que é 4 de maio, 1 da tarde (mesmo dia, hora diferente). E se, ainda neste mesmo instante, eu perguntar para outro amigo meu que mora no Japão, ele responderá que lá já é dia 5 de maio, e provavelmente ele vai estar de mau humor, porque ainda são 5 da manhã em Tóquio.

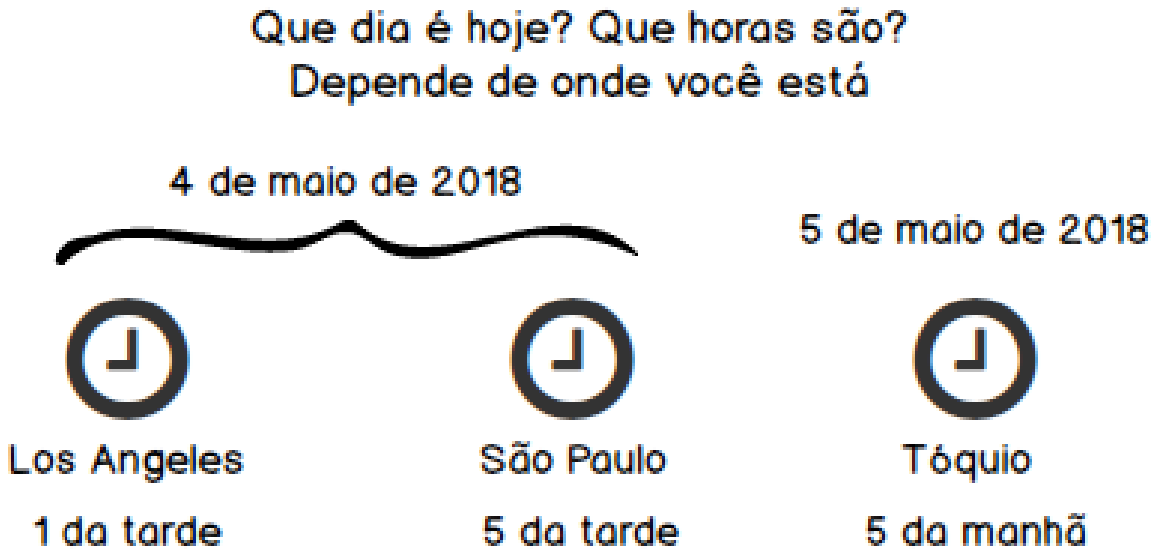

Figura 1.1: Neste exato momento, em cada parte do mundo, a data e hora atual não serão as mesmas

No dia a dia, costumamos trabalhar com a data e hora local, sendo que o "local" está sempre implícito. "Se eu pergunto que dia é hoje ou que horas são, é **óbvio** que eu me refiro ao lugar onde estamos", você vai pensar. E, de fato, para nós é muito prático que tais informações sejam implícitas. Na verdade, fazemos até mais do que isso, pois muitas vezes omitimos ainda mais informações. A pergunta "Que dia é hoje?" pode ter mais de uma resposta válida:

- 4 de maio de 2018
- 4 de maio – omitimos o ano, pois assumimos que é o ano atual
- Dia 4 – já está implícito que nos referimos ao mês e ano atual
- Sexta-feira – também é uma resposta válida, afinal só me interessa saber quanto falta para o fim de semana

O mesmo vale para as horas:

- Preciso saber a hora exata ou posso arredondar os minutos? (17 horas ou 17:02, faz diferença? Depende do contexto.)
- Preciso saber até os segundos (e milésimos de segundo)? Geralmente a hora e minuto são suficientes, mas há situações

em que saber os segundos é necessário/desejado.

Enfim, no nosso cotidiano, ao lidar com datas e horas, usamos uma abordagem mais, digamos, simplificada, na qual muita informação pode ser omitida pois já está implícita de acordo com o contexto.

Mas não é assim que computadores trabalham, e é aí que começam os problemas.

O computador pode enxergar a mesma data/hora de hoje (4 de maio de 2018 às 17:00 em São Paulo) como um número gigante: `1525464000000` . Nos próximos capítulos explicaremos melhor o que é este número e o que ele significa, mas por ora fiquemos apenas com este importantíssimo conceito: esse número é um valor *absoluto*, no sentido de que é o mesmo para todos, não importando onde você está. Todos os computadores do mundo que tenham verificado a data e hora atual, naquele mesmo instante, obtiveram este resultado.

O que acontece é que, para cada lugar do mundo, este mesmo número corresponde a uma data e hora diferente. O instante (o número gigante `1525464000000`) é o mesmo para todos, mas a data e hora **local** não (em São Paulo e Los Angeles o dia é o mesmo mas o horário não, e em Tóquio até o dia é diferente – mas o número gigante é o mesmo para todos, tanto faz se o computador está no Brasil, nos EUA, ou no Japão).

Existem, portanto, duas formas diferentes de se "ver" o tempo:

- Do ponto de vista humano: usa valores para dia, mês, ano, hora, minuto e segundo. Depende do local onde você está (fuso horário), e também pode variar de acordo com:
    - regras deste local: se tem horário de verão ou não, por exemplo;
    - regras do calendário utilizado: por exemplo: de acordo com o calendário *Umm al-Qura*, uma das variantes do calendário islâmico, a data de 4 de maio de 2018

corresponde ao dia 18 do mês 8 (chamado de *Shaʿbān*) do ano de 1439 (dois valores diferentes para o mesmo dia);
- Do ponto de vista das máquinas: usa valores absolutos, como o número gigante `1525464000000` . É o mesmo para todos, não importando o local ou calendário utilizado.

Como a programação é basicamente a tradução entre os dois pontos de vista (humano e máquina), um desenvolvedor tem que saber como ambos funcionam e, principalmente, como traduzir corretamente de um para outro.

1.1 Data e hora local

Quando eu digo que hoje é 4 de maio de 2018 e são 17:00, estou dizendo a data e hora do meu ponto de vista. Estes são os valores atuais para o lugar onde estou no momento – no caso, São Paulo – por isso podemos dizer que esta é minha data e hora **local**.

Se alguém em Tóquio disser que tinha um compromisso no dia 4 de maio de 2018 às 17:00 no horário local do Japão, não quer dizer que este compromisso é agora. Afinal, enquanto aqui em São Paulo são 17:00 do dia 4, em Tóquio já são 05:00 do dia 5, logo, o compromisso já ocorreu 12 horas atrás.

Portanto, uma informação como "4 de maio de 2018 às 17:00", sem nenhuma indicação do lugar ao qual ela se refere, não pode ser traduzida para um momento único. Em cada fuso horário do mundo, esta data e hora ocorre em um instante diferente, e tudo depende do lugar em questão, que pode ser conhecido ou não.

Algumas APIs chamam esta informação (4 de maio de 2018 às 17:00) de data/hora local (em inglês, *local date/time*), no sentido de que o fuso horário não é conhecido, e portanto ela pode se referir a qualquer local – assim, não se deve assumir que ela se refere a um lugar específico.

Mas o termo "local" também é usado quando o fuso horário é conhecido e queremos enfatizar que a data e hora em questão referem-se àquele lugar. Exemplo: "O avião pousou em São Paulo no dia 4 de maio de 2018 às 17:00, horário local".

Ao programar usando datas, é importante saber como a linguagem, API ou *framework* utilizado está tratando os dados e valores relativos à data e hora. As funções ou métodos possuem somente a data ou também levam em conta a hora? O fuso horário é considerado ou descartado (a data e hora é "local" no sentido de não ter um fuso, ou pertence a um específico)? Sempre leia atentamente a documentação para saber como aquela classe, função ou serviço trata a data e hora, para não ter resultados inesperados.

Por exemplo, suponha que em um sistema existe uma informação dizendo que a data e hora de algum evento é 4 de maio de 2018, às 17:00. Quando um usuário acessar o sistema, este deve informar quando será o evento.

Só que este evento vai receber pessoas de diferentes países, e todos precisam saber exatamente quando reservar seus hotéis e passagens aéreas, para que cheguem ao evento no momento certo. Só que a informação que o sistema tem está incompleta. Em cada lugar do mundo, 4 de maio de 2018 às 17:00 ocorre em momentos diferentes. A data e hora do evento está informando um dia e horário local, mas não diz qual é este local.

Um usuário pode ver esta informação e deduzir que se trata do dia e horário do local do evento, por exemplo (uma dedução lógica, você vai pensar – é *óbvio* que é a data e hora da cidade onde o evento ocorrerá). Mas outro usuário pode pensar que o sistema está mostrando a data e hora da sua própria cidade, e acabar comprando passagens aéreas e reservando hotel para os dias errados, podendo inclusive chegar um dia depois e perder o evento.

Se você mostrar uma data e hora local, mas não disser qual é este local, a informação será incompleta e ambígua, pois o usuário não saberá se aquilo se refere ao local do evento ou ao fuso da sua própria cidade. Você não pode esperar que os usuários deduzam, pois sempre há a chance de alguém entender errado.

Um bom exemplo de sistemas que evitam esta ambiguidade são os sites de venda de passagens aéreas, que sempre informam os horários de chegada e saída baseados no local onde o avião estará (o horário da partida está no fuso horário da cidade de origem, e o horário da chegada, no fuso horário da cidade de destino). E para garantir que o usuário não se confunda, muitos também mostram a duração total do voo. Quando se trata de datas, quanto mais claro, melhor.

A melhor forma de remover qualquer ambiguidade é sempre definir o local ao qual uma data e hora se referem. Em outras palavras, informe o fuso horário.

O conceito de fusos horários é um dos mais complicados quando se trata de datas e horas, e por isso é muito fácil usá-los de maneira errada. Vamos entender melhor este conceito, e como usá-lo corretamente, no próximo capítulo.

Capítulo 2
Fusos horários e os timezones

Conforme já vimos, neste exato instante, cada lugar do mundo pode estar em um dia e horário diferentes. Tudo graças aos fusos horários.

Antigamente, não havia uma preocupação tão grande com relação ao horário de outros lugares, e cada cidade adotava seu próprio horário local, muitas vezes com diferença de alguns minutos entre localidades próximas.

Com o aumento das linhas ferroviárias, principalmente nos EUA e Reino Unido, isso se tornou um problema. Como cada cidade tinha seu próprio fuso horário local, as tabelas de horários de saída e chegada dos trens acabavam ficando enormes. Só nos EUA, por exemplo, uma das companhias mantinha uma tabela com horários de mais de 100 cidades, porém, estima-se que existiam mais de 300 fusos diferentes no país.

Como a manutenção destas tabelas de horários tornou-se impraticável, resolveram padronizar os fusos horários da forma que conhecemos hoje. Em vez de cada cidade ter sua própria hora local, uma região inteira – que poderia ser inclusive um país inteiro – passava a seguir o mesmo horário.

2.1 E assim surgiu o GMT

Por volta de 1847, a maioria das companhias ferroviárias inglesas já usavam o "Horário de Londres" (*London Time*), que era definido pelo Observatório Real de Greenwich. Porém, somente em 1880 isso passou a ser adotado pelo restante do país. Com isso, a ilha da

Grã Bretanha adotou o *Greenwich Mean Time* – também conhecido pela sigla **GMT** – em 2 de agosto de 1880.

Quatro anos depois, em outubro de 1884, ocorreu a *International Meridian Conference*, em Washington D.C., EUA. Nesta conferência, decidiu-se pelo uso do Meridiano de Greenwich como a base a partir da qual todos os fusos horários seriam definidos. Um dos principais fatores que influenciaram a decisão é que o Observatório de Greenwich produzia os dados mais confiáveis da época.

A partir daí, cada país passou a adotar um ou mais fusos horários baseados em Greenwich, com diferença de um determinado número de horas para mais ou para menos (nem sempre são horas inteiras, como veremos adiante). O horário oficial de Brasília, por exemplo, passou a ser de 3 horas antes de Greenwich.

Teoricamente, os fusos horários deveriam seguir os meridianos, com variação de uma hora a cada 15 graus de longitude, mas na prática os países acabam usando as suas divisas ou qualquer outro critério, resultando em várias regiões do mundo que, mesmo estando na mesma longitude, possuem fusos diferentes.

2.2 UTC, o novo GMT

Na mesma conferência que definiu o GMT como o padrão a ser adotado – a *International Meridian Conference*, em 1884 – também foi definido o *Universal Time* (**UT**), que é basicamente definido pela velocidade média de rotação da terra. Há várias versões do UT (como o UT0, UT1 e UT2), cada uma com um modo específico de ser calculado, sempre levando em conta observações e cálculos astronômicos. A versão UT1, combinada com o *International Atomic Time* (TAI) – que é medido por relógios atômicos de alta precisão – resulta no padrão **UTC** (*Coordinated Universal Time*, ou "Tempo Universal Coordenado").

Na época, ingleses e franceses discutiam se a sigla deveria ser CUT (pois em inglês o nome é *Coordinated Universal Time*) ou TUC (em francês, *Temps Universel Coordonné*). No fim, decidiu-se por uma sigla que não favorecesse nenhum dos idiomas, e assim surgiu o nome **UTC**.

Como o objetivo deste livro é explicar o UTC do ponto de vista da programação, não vamos entrar nos detalhes específicos de como ele é calculado (caso tenha ficado curioso, veja a descrição da Wikipedia: https://en.wikipedia.org/wiki/Universal_Time/). O que importa para nós é o fato de que em 1972 ele passou a ser o padrão adotado para fusos horários.

Ou seja, a partir de 1972, o UTC substituiu o GMT, portanto todos os horários locais do mundo são definidos como uma diferença em relação a UTC. O horário oficial de Brasília, por exemplo, está 3 horas atrás de UTC. Para descrever tal informação, costuma-se escrever como `UTC-03:00`, `-03:00`, `-0300` ou simplesmente `-03`. **Esta diferença é chamada de *UTC offset*, ou simplesmente *offset*.** Pode-se então dizer que o horário oficial de Brasília tem um offset negativo (-3 horas, ou 3 horas atrás de UTC).

> Não há uma tradução certa para *UTC offset* em português. As opções oferecidas pelo dicionário (deslocamento, compensação) ou mesmo palavras com sentido próximo (como "diferença") na minha opinião não são satisfatórias. Por isso, usarei o termo em inglês.
>
> Os formatos `-03:00`, `-0300` e `-03` são definidos pela norma ISO 8601, que define formatos para representar datas e horas, e será explicada em mais detalhes posteriormente.

Depois que foi substituído pelo UTC, o GMT passou a ser apenas o nome do fuso horário adotado pelo Reino Unido – e mais alguns países – quando não está em horário de verão. Você pode ver todos os países que adotam GMT nesta lista:

https://www.timeanddate.com/time/zones/gmt. Na verdade, estes países estão em `UTC+00:00` (o offset é zero, ou seja, nenhuma diferença com relação a UTC).

É mais comum ver o offset zero sendo escrito como `Z` , e também é chamado de *Zulu Time*, já que a mesma letra é usada para definir a *Zulu Time Zone*, que é um dos *timezones* militares, que não serão cobertos por este livro: https://en.wikipedia.org/wiki/List_of_military_time_zones/.

Apesar disso, você ainda verá em muitos lugares um offset escrito como `GMT-03:00` ou `GMT-3` , ou alguma outra variação. Conceitualmente não está correto, já que o padrão atual é o UTC, mas algumas APIs ainda aceitam estes formatos, provavelmente por questões de retrocompatibilidade.

Outro detalhe importante é que cada país adotou o GMT e/ou UTC em uma data diferente, então cuidado ao verificar datas antigas, especialmente antes da definição de cada um destes padrões. Para datas anteriores à padronização ou adoção dos fusos atuais, é usado o *Local Mean Time* (LMT), que é baseado na longitude do local em questão. Com isso, o resultado são offsets "quebrados", como `UTC -3:06:28` (3 horas, 6 minutos e 28 segundos atrás de UTC).

> Você pode ver mais informações sobre GMT x UTC nestes artigos:
>
> - Diferenças entre eles, sob uma ótica mais prática do que científica: https://codeofmatt.com/2014/06/18/utc-vs-gmt/
> - Detalhes sobre UTC: https://codeofmatt.com/2017/10/05/please-dont-call-it-epoch-time/#afewthingsyoushouldknowaboutunixtimestamps/

Timezone = Fuso horário?

A princípio, o termo em inglês *timezone* poderia ser muito bem traduzido para "fuso horário". Porém, nos próximos tópicos, explicarei o que é este conceito e por que não acho que a tradução em português seja totalmente fiel ao conceito ao qual o termo em inglês se refere.

Um timezone representa uma região específica do planeta, e possui uma lista de todos os offsets que aquela região teve, tem e terá durante toda a sua história – ou pelo menos durante a história em que o horário local, com as regras que temos hoje (24 horas por dia, 60 minutos por hora etc.), é usado.

Conforme já dito anteriormente, antigamente cada cidade tinha seu próprio horário local, e muitas usavam o *Local Mean Time*, baseado na longitude. São Paulo, por exemplo, usava o offset `-3:06:28`, e somente em 1914 passou a adotar o offset atual, que é `-03:00`.

Porém, o offset não é o mesmo ao longo do ano, graças a vários fatores, entre eles o horário de verão.

2.3 Como funciona o horário de verão

Do ponto de vista da população em geral, o funcionamento do horário de verão já é um pouco confuso, mas basicamente consiste em adiantar ou atrasar o relógio em uma hora (na maioria dos casos), sendo que a mudança ocorre em um dia e horário específicos.

Em alguns países do hemisfério norte, o horário de verão começa no último domingo de março, em outros é no primeiro domingo. E o horário de início pode ser às 2 ou 3 da manhã, dependendo da região. Já no hemisfério sul, ele costuma se iniciar entre outubro e novembro, e cada país tem sua própria regra quanto aos dias e horários de início e fim. O motivo de haver tanta variação é que

cada governo decide as regras de seu respectivo país, estado ou região. Não há um padrão universal, cada caso é um caso.

Do ponto de vista técnico, o que ocorre é uma mudança de offset. Vamos ver com mais detalhes como funciona, usando como exemplo o horário de verão em São Paulo.

Em 2017, o horário de verão em São Paulo teve início no dia 15 de outubro. À meia-noite, os relógios foram adiantados em uma hora, ou seja, na prática "pulamos" uma hora. Hoje em dia, computadores, celulares e diversos outros aparelhos já vêm programados para fazer esta mudança automaticamente, então o relógio pula diretamente de 23:59:59 para 01:00.

Ou seja, quando o horário de verão começa em São Paulo, todos os minutos entre meia-noite (00:00) e 00:59 **não existem naquele lugar e dia específicos**. Uma data e hora como "15 de outubro de 2017, às 00:30 em São Paulo", por exemplo, está errada, pois está informando um horário que não existe naquele dia e lugar. Nada impede que um relógio informe este horário, claro, mas neste caso o relógio estará errado, provavelmente porque alguém esqueceu de adiantá-lo em uma hora.

Este "salto" também é chamado de *DST gap*. DST é a sigla para *Daylight Saving Time*, que é como o horário de verão é chamado em inglês, embora também seja comum o uso do termo *Summer Time*. Já a palavra *gap* tem o sentido de "lacuna" ou "quebra de continuidade", já que um intervalo de tempo foi "pulado", do ponto de vista local. Mas de qualquer forma, usarei o termo em inglês mesmo.

Você verá em alguns lugares escrito *Daylight Saving**s** Time*, mas o correto é *Saving* (no singular). E quando não é horário de verão, usa-se os termos *Standard Time* ou *Winter Time*.

Como é possível que uma hora inteira seja pulada, sem que haja uma descontinuidade no tempo? Na verdade, o que acontece é que o offset é mudado, e a data local passa a refletir esta mudança.

Para entender melhor o que isso significa, vamos ver como está a data e hora em São Paulo quando falta 1 minuto para começar o horário de verão. Ou seja, 14 de outubro de 2017, às 23:59. Neste dia e horário, o offset é `-03:00` , ou seja, 3 horas antes de UTC. Portanto, para saber o instante correspondente em UTC, basta somar 3 horas (ou seja, 23:59 no offset `-03:00` equivale a 02:59 do dia seguinte em UTC):

São Paulo	UTC (offset `00:00`)
14 de outubro de 2017, às **23:59**, offset `-03:00`	15 de outubro de 2017, às **02:59**

O que acontece 1 minuto depois em São Paulo? Se não houvesse horário de verão, teríamos 15 de outubro de 2017 à meia-noite, mas este é o instante exato em que começa o horário de verão, e o relógio deve ser adiantado em 1 hora. O horário local, portanto, passa a ser 01:00.

Mas 1 minuto depois em UTC deve resultar em 15 de outubro de 2017, às 03:00. Isso porque o tempo deve ser contínuo, já que UTC é o padrão seguido no mundo todo, não sofre influência do horário de verão e não podem haver "saltos". Por isso, a solução é ajustar o offset de São Paulo para `-02:00` . Como resultado, temos:

Instante	São Paulo	UTC (offset `00:00`)
1 minuto antes do horário de verão em SP	14 de outubro de 2017, às 23:59, offset `-03:00`	15 de outubro de 2017, às 02:59

Instante	São Paulo	UTC (offset `00:00`)
Começa o horário de verão em SP	15 de outubro de 2017, às 01:00, offset `-02:00`	15 de outubro de 2017, às 03:00

Repare que o horário local em São Paulo comportou-se conforme a regra do horário de verão: à meia-noite o relógio foi adiantado em uma hora, direto para 01:00. Já o instante correspondente em UTC é contínuo (das 02:59 para as 03:00), e isso é conseguido mudando-se o offset usado em São Paulo, de `-03:00` para `-02:00` .

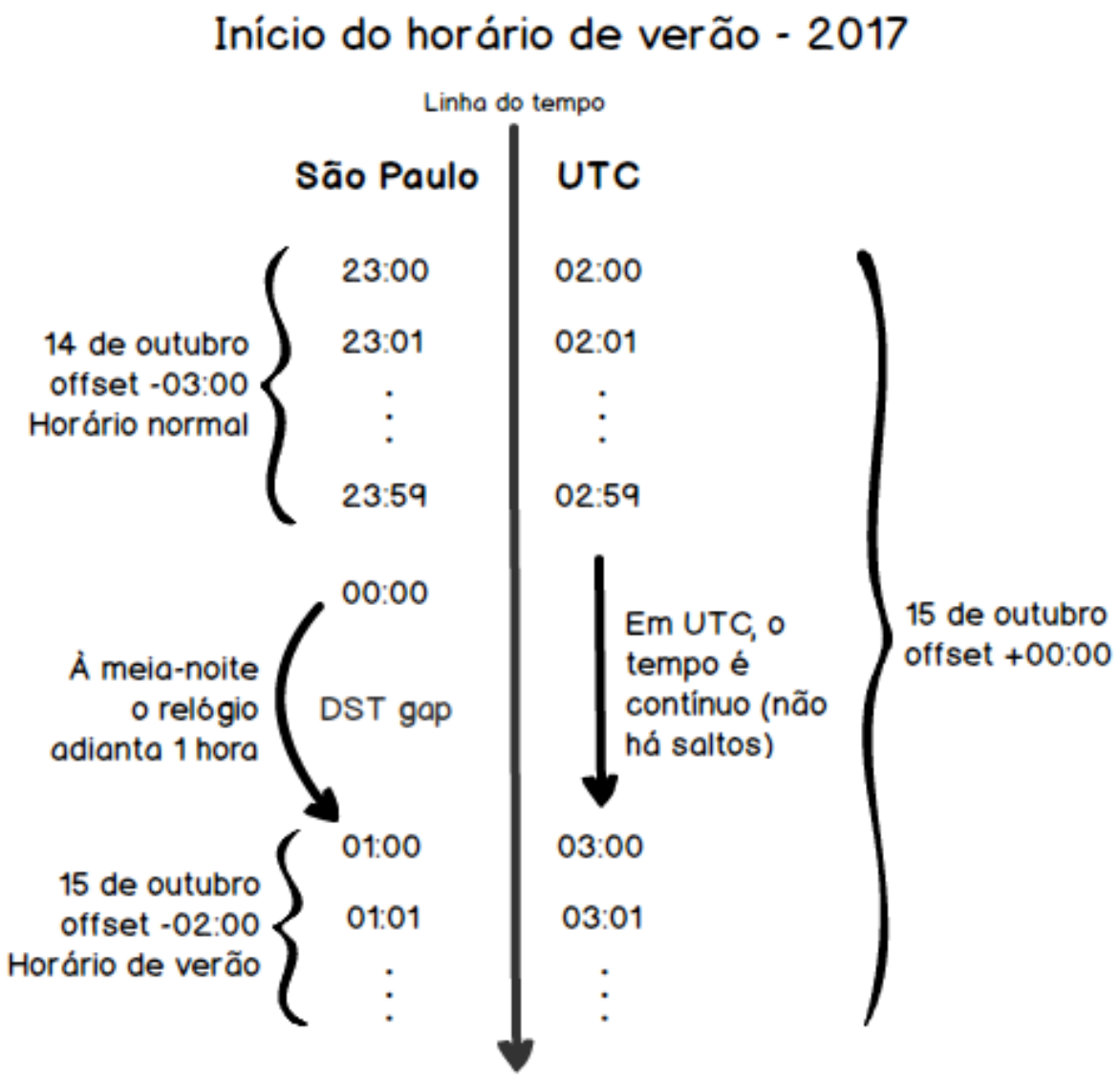

Figura 2.1: DST Gap. Os minutos entre 00:00 e 00:59 são pulados

Outro ponto interessante de notar é que, como foi pulada uma hora, o dia 15 de outubro de 2017, em São Paulo, começou de fato à 01:00. Portanto, nesta cidade – e em todas as cidades que seguem esta mesma regra de horário de verão – este dia teve apenas **23 horas**.

Pois é, nem todos os dias têm 24 horas.

Quando o horário de verão acaba, acontece o oposto: quando chega meia-noite, o relógio é atrasado em 1 hora, de forma que "recuperamos" aquela hora que havia sido "pulada" anteriormente. Do ponto de vista técnico, o offset volta a ter o valor anterior, para que os instantes em UTC permaneçam contínuos.

Em 2018, o horário de verão em São Paulo terminou no dia 18 de fevereiro. À meia-noite, os relógios foram atrasados em 1 hora, voltando para o dia 17 às 23:00. Nos sistemas e aparelhos configurados para se ajustar automaticamente, o relógio muda das 23:59:59 de volta para 23:00.

Ou seja, quando acaba o horário de verão, todos os minutos entre 23:00 e 23:59 **ocorrem duas vezes**: uma no horário de verão e outra no horário "normal". Para que não haja descontinuidade no tempo e os instantes em UTC sejam contínuos, o que ocorre na prática é a volta do offset para o valor que tinha anteriormente:

Instante	São Paulo	UTC (offset `00:00`)
1 minuto antes de acabar horário de verão em SP	17 de fevereiro de 2018, às 23:59, offset `-02:00`	18 de fevereiro de 2018, às 01:59
Acaba o horário de verão em SP	17 de fevereiro de 2018, às 23:00, offset `-03:00`	18 de fevereiro de 2018, às 02:00

Quando um horário local ocorre 2 vezes, uma em cada offset, é chamado de *DST overlap*, sendo que *overlap* tem o sentido de "sobreposição" (mas de qualquer forma, usarei o termo em inglês).

Quando há um overlap e o offset não é informado, é impossível saber se estamos nos referindo ao momento antes ou depois do término do horário de verão.

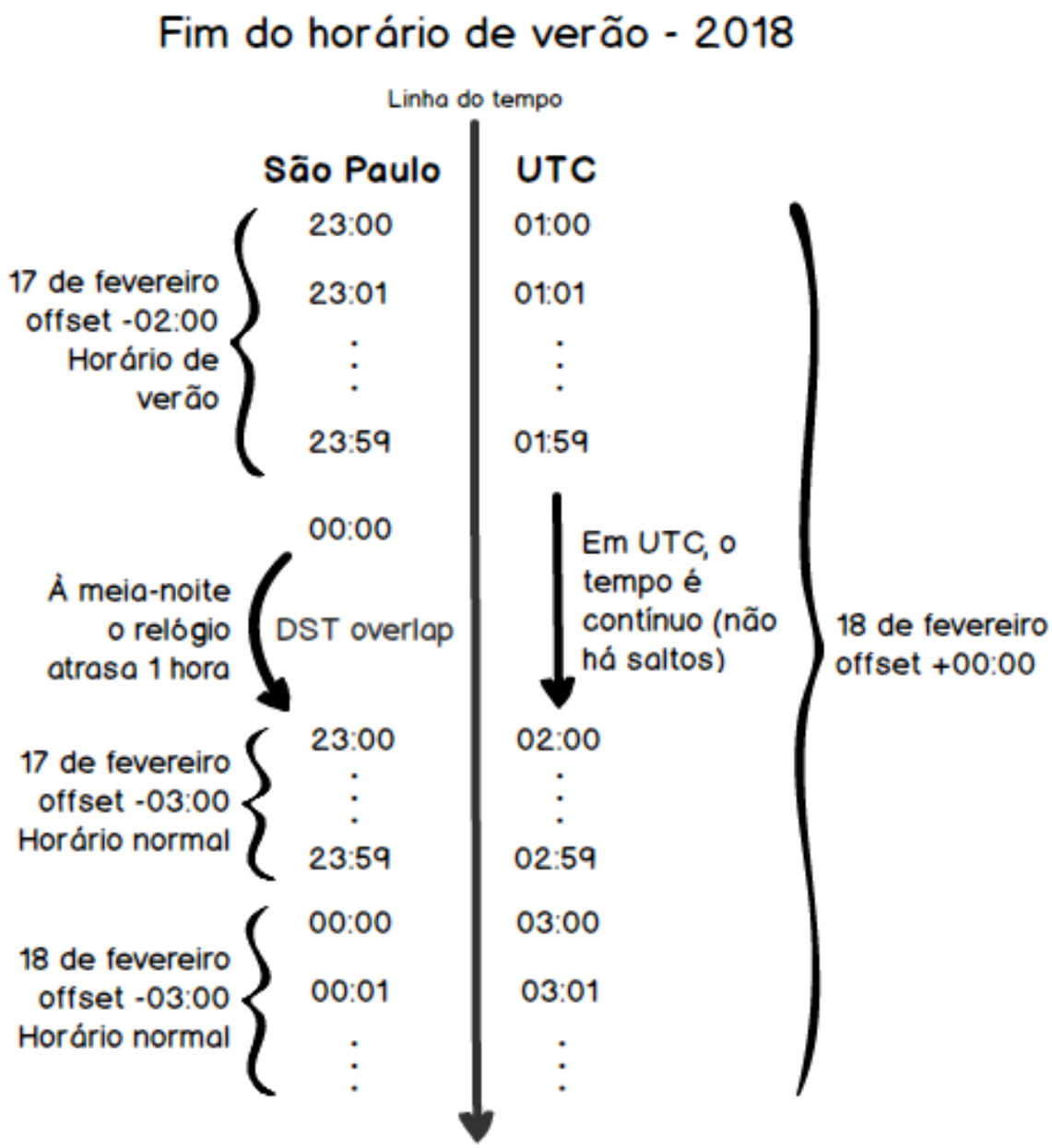

Figura 2.2: DST Overlap. Os minutos entre 23:00 e 23:59 ocorrem duas vezes

Note também que, como tivemos uma hora repetida (todos os minutos entre 23:00 e 23:59 ocorreram duas vezes), o dia 17 de fevereiro de 2018, em São Paulo – e em todos os lugares que seguem esta mesma regra de horário de verão – teve **25 horas**.

Nem sempre é sobre o horário de verão

O horário de verão é o caso mais comum de mudança de offset em uma determinada região, mas nem toda mudança de offset ocorre por causa dele.

Um dos casos mais notáveis é o de Samoa. Em dezembro de 2011, eles mudaram seu offset de `-10:00` para `+14:00` (estavam 10 horas antes de UTC, e passaram a estar 14 horas depois). Isso só foi possível porque o país fica localizado muito próximo à Linha Internacional de Data, que é o meridiano oposto ao de Greenwich.

Partindo de Greenwich, ao viajar para o leste, os offsets vão aumentando, e para o oeste eles diminuem. O ponto onde os offsets mínimo e máximo se encontram é a Linha Internacional de Data. Cruzando-a, você pode pular ou voltar um dia, dependendo do sentido em que está viajando.

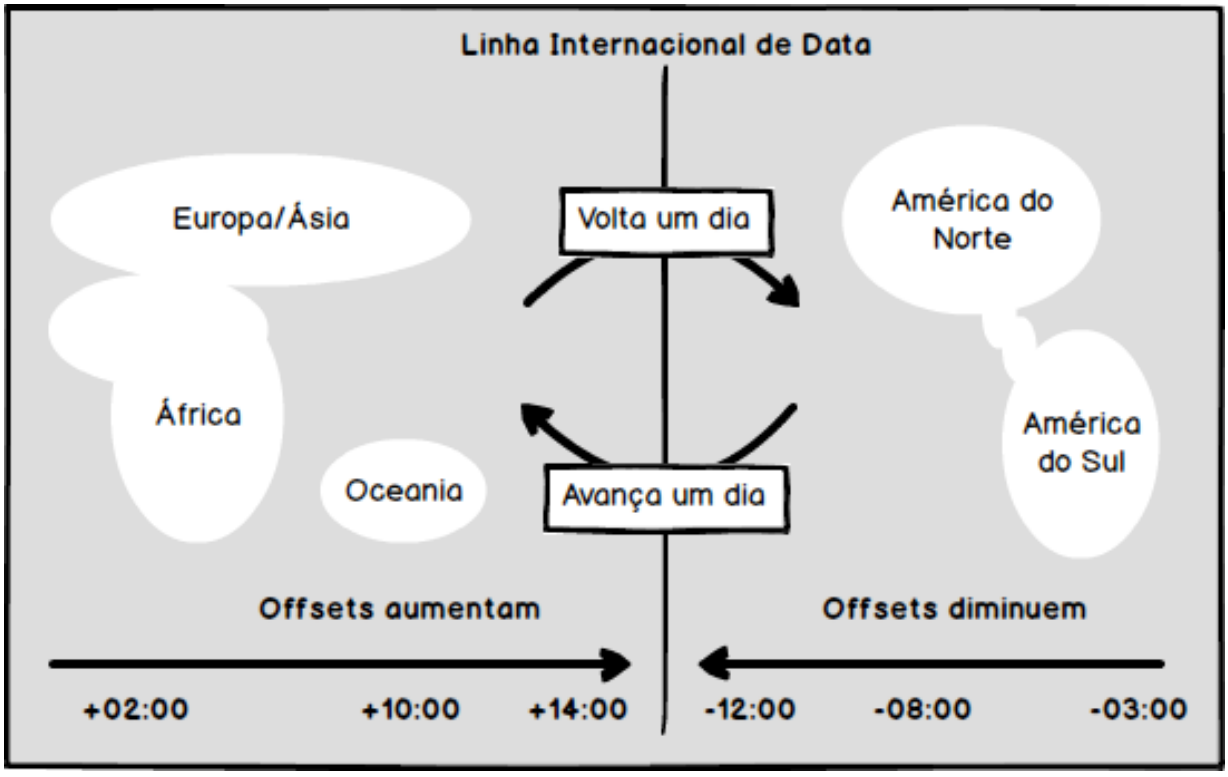

Figura 2.3: Cruzar a Linha Internacional de Data é um jeito de "viajar no tempo"

Na verdade, a linha não é uma reta perfeita, e todos os países próximos ao meridiano (com longitude próxima dos 180 graus)

podem escolher de qual lado ficam. Usando como referência a imagem anterior, Samoa estava do lado direito da linha, no offset `-10:00` . Depois da mudança, passou a estar do lado esquerda da linha, no offset `+14:00` .

Obviamente, o país não mudou de lugar. A Linha Internacional de Data, assim como qualquer meridiano, é uma linha imaginária. E como Samoa está muito próxima dela, é possível escolher de qual lado ficar. É como se uma cidade pertencesse a um país A, mas ficasse próxima à fronteira com um país B, e decidisse mudar de país. A cidade não mudaria de lugar, mas a fronteira – esta linha imaginária que separa os dois países – seria movida, de modo que a cidade passasse a fazer parte do país B.

Esta decisão foi tomada para facilitar o comércio com a Austrália e Nova Zelândia. Antes da mudança, enquanto era sexta-feira de manhã em Samoa, na Austrália já era sábado, e quando era segunda-feira na Austrália, em Samoa ainda era domingo. Como perder dois dias úteis com um importante parceiro comercial não é bom para os negócios, a solução foi mudar para o outro lado da Linha Internacional de Data, de forma a diminuir essa diferença.

A mudança foi programada para ocorrer no dia 30 de dezembro de 2011: à meia-noite, o offset mudaria de `-10:00` para `+14:00` , mas ao contrário do que ocorre no horário de verão, os relógios não seriam adiantados nem atrasados. Apenas o offset seria mudado.

Com isso, os horários locais ficaram assim:

Instante	Samoa	UTC (offset `00:00`)
1 minuto antes da mudança	29 de dezembro de 2011, às 23:59, offset `-10:00`	30 de dezembro de 2011, às 09:59
Horário da mudança	**31** de dezembro de 2011, meia-noite, offset `+14:00`	30 de dezembro de 2011, às 10:00

Repare que novamente os instantes UTC são contínuos, mas a mudança de offsets fez com que localmente **todo o dia 30 de dezembro fosse pulado**. É isso mesmo: graças a esta mudança, em Samoa não existe o dia 30 de dezembro de 2011. É um *gap* de 24 horas, e não está relacionado com o horário de verão.

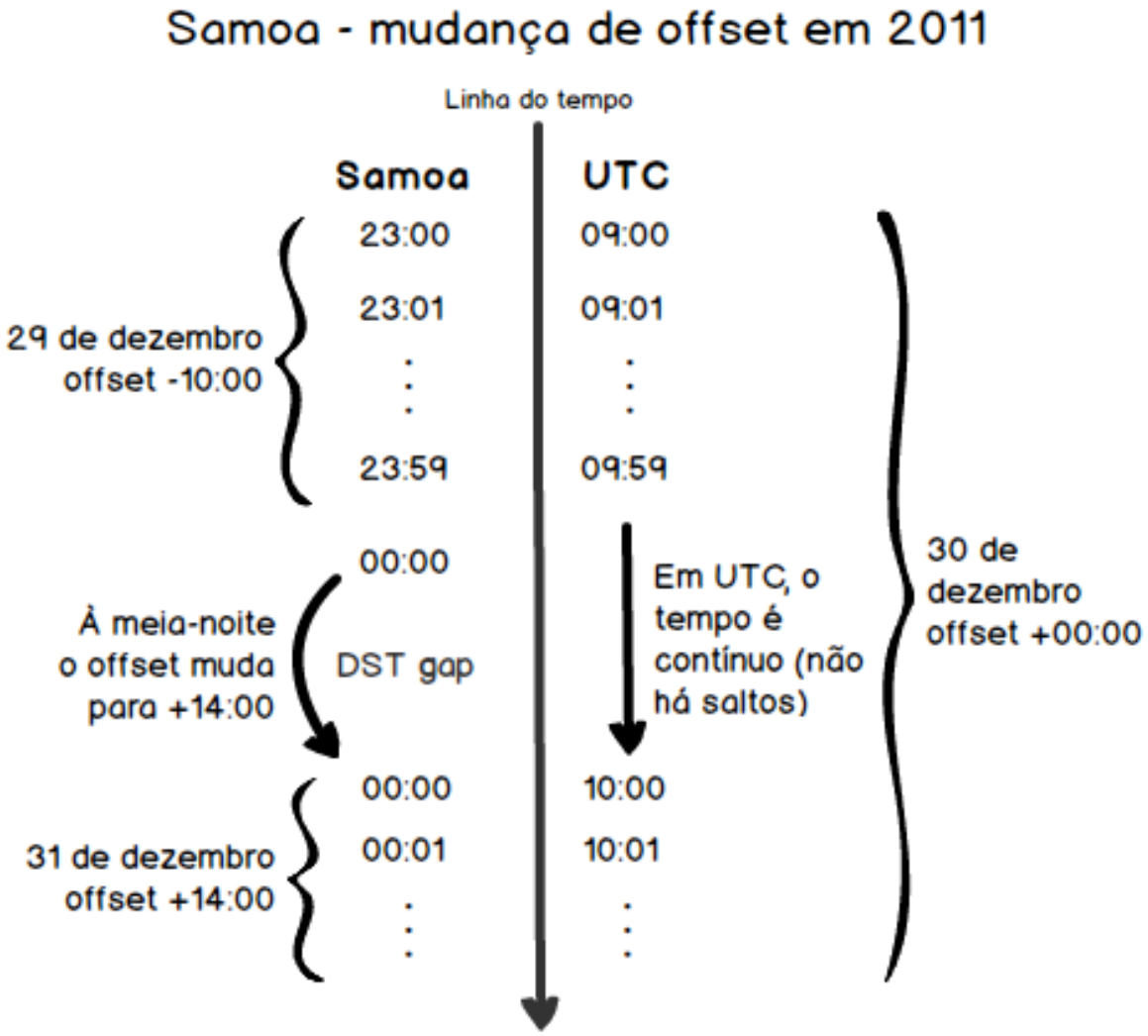

Figura 2.4: A mudança de offset fez com que o dia 30 fosse pulado

Outro caso recente é o da Coreia do Norte, que no dia 5 de maio de 2018 à meia-noite adiantou os relógios em meia-hora (e o offset mudou de `+08:30` para `+09:00`), para alinhar seu horário com a Coreia

do Sul. É um *gap* de meia hora, e também não está relacionado ao horário de verão.

Nem sempre a diferença é de 1 hora

A maioria dos timezones usa um offset de horas inteiras, como `-03:00` ou `+09:00` , mas nem todos são assim. A Índia, por exemplo, atualmente usa `+05:30` (5 horas e 30 minutos depois de UTC), enquanto o Nepal usa `+05:45` (sim, 5 horas e 45 minutos), e há vários outros lugares do mundo com offsets assim, como você pode ver nesta lista (https://www.timeanddate.com/time/time-zones-interesting.html/).

Os *gaps* que ocorrem quando um lugar muda seu offset também podem não ser de 1 hora, como já vimos nos exemplos anteriores. E mesmo o horário de verão em alguns países não consiste em atrasar ou adiantar o relógio em 1 hora. Nas ilhas Lord Howe (Austrália), por exemplo, os relógios são adiantados 30 minutos durante o horário de verão (https://www.timeanddate.com/time/zone/australia/lord-howe-island?year=2010/). E nos anos 30, algumas regiões da Malásia adiantavam apenas 20 minutos (https://www.timeanddate.com/time/zone/malaysia/kuching?year=1935/).

2.4 Nada é garantido, nada é para sempre

Todas as regras de timezones são definidas pelos governos dos respectivos países, estados, províncias, enfim, por qualquer autoridade que possua autonomia administrativa sobre determinada região.

Entre estas regras, a principal é o offset a ser usado pela região. Outra regra importante é se o horário de verão vai ser adotado ou

não. Caso seja, também é definido o respectivo offset, além dos dias e horários em que ele começa e termina. E a cada ano estas datas de início e fim podem ocorrer em um dia diferente, devido a regras como "começa no primeiro domingo de novembro" ou ainda "acaba no terceiro domingo de fevereiro, mas se for Carnaval, é postergado para o domingo seguinte".

Não só os dias de início e fim do horário de verão são diferentes, mas o horário também varia de um lugar para outro. No Brasil, as transições do horário de verão ocorrem à meia-noite, nos EUA às 02:00 e em vários países da Europa o início é às 02:00, mas o fim é às 03:00. Além disso, o verão do hemisfério norte corresponde ao inverno do hemisfério sul, e vice-versa. Por isso, em qualquer época do ano pode ter algum lugar do mundo em horário de verão.

Estas decisões costumam ser muito mais político-administrativas do que baseadas em critérios técnicos, e elas mudam o tempo todo. Já vimos alguns exemplos de países que mudaram seus offsets, e eles não são casos isolados. Muito pelo contrário: a todo momento, em algum lugar do mundo, há alguém discutindo se deveria abolir ou adotar o horário de verão, ou se o offset não deveria ser uma hora a mais, ou meia hora a menos. Um link interessante para acompanhar estas discussões e mudanças é a seção de "Time Zone News" do site *timeanddate* (https://www.timeanddate.com/news/time/).

Uma mudança recente, inclusive, aconteceu no Brasil. Até 2017, o horário de verão começava no terceiro domingo de outubro. Mas a partir de 2018, passou a começar no primeiro domingo de novembro, conforme decreto publicado em dezembro de 2017 (http://www.planalto.gov.br/ccivil_03/_ato2015-2018/2017/decreto/D9242.htm). E será assim "para sempre", ou seja, até que o governo decida mudar de novo.

Como as regras dos timezones mudam o tempo todo, qualquer tentativa de mapeá-las manualmente vai eventualmente falhar. Nunca deixe nada *hardcoded* no seu código, por exemplo, usar sempre um offset específico para cada região, ou tentar adivinhar

quando determinado local está em horário de verão. Estas regras mudam o tempo todo, por mais que achemos que não. Nunca assuma nada, e nunca tente codificar estas regras manualmente. Use uma API dedicada e não reinvente a roda.

Capítulo 3
Nomenclatura dos timezones e formatos de data

No capítulo anterior, vimos que as regras de timezones mudam o tempo todo, e que tentar codificá-las manualmente é inviável. Mas então, onde e como eu posso obter estas informações?

Infelizmente, não há um padrão para isso: não existe nenhuma norma ISO ou algo do tipo. O que temos são bancos de dados com informações de timezones ao redor do mundo, e mantidos por um grupo específico de pessoas ou corporações. Os dois mais notáveis são:

Microsoft Windows Time Zone Database

Já vem junto com a instalação do Windows, e pode ser acessado pela chave de registro `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Time Zones`. Para mais detalhes, consulte o site da Microsoft (https://support.microsoft.com/en-us/help/22803/daylight-saving-time/).

É mantido pela Microsoft e atualizado nos *updates* do Windows. Mas algumas zonas são abrangentes demais, cobrindo áreas com regras diferentes e muitos nomes podem ser ambíguos ou confusos (alguns "bons" exemplos são "Arab Standard Time", "Arabian Standard Time" e "Arabic Standard Time", que se referem a 3 lugares diferentes).

Também tem outro problema: as descrições possuem um offset, mas geralmente é o valor usado durante o *Standard Time* (quando não está em horário de verão), independente da data em que você consulta. Por exemplo, "(UTC-05:00) Eastern Time (US & Canada)".

IANA/Olson Time Zone Database

Também chamado de *ZoneInfo*, *TZDB*, *TZ database*, *IANA database*, *Olson database* ou simplesmente *IANA*, é o que abordaremos no decorrer do livro, e seu site é https://www.iana.org/time-zones/.

> O nome *Olson database* vem do seu criador, Arthur David Olson. Na Wikipedia há um breve histórico de como a IANA se tornou responsável por este banco de dados (https://en.wikipedia.org/wiki/Tz_database#History/).

A IANA é a *Internet Assigned Numbers Authority*, entidade que cuida de várias questões relacionadas à Internet (https://www.iana.org/about), e o TZDB é só mais uma de suas atribuições. O TZDB é o banco de dados de timezones mais abrangente e completo que existe, e é usado por várias plataformas e linguagens (Linux, Mac, Java, PHP, entre outros). Quando a linguagem não tem suporte nativo, provavelmente já existe uma biblioteca que usa os dados da IANA. A lista de softwares que usam o TZDB pode ser encontrada neste link (https://en.wikipedia.org/wiki/Tz_database#Use_in_software_systems/).

Neste banco de dados, os timezones recebem nomes no formato "Continente/Região", por exemplo `America/Sao_Paulo`, `Europe/London` e `Asia/Tokyo`. Cada um desses identificadores representa uma região do globo terrestre, e o banco de dados possui todo o histórico de offsets que esta região teve, tem e terá durante sua história, além de conter o momento exato em que essas mudanças ocorreram, e quais os offsets usados antes e depois de cada mudança. Portanto, as regras de horário de verão (quando começa e termina, qual o offset antes e depois etc.), e qualquer outra mudança no fuso horário, como os exemplos de Samoa e Coreia do Norte que vimos anteriormente, estão neste banco de dados.

Para as mudanças futuras, assume-se que a regra atual é a que será usada. Por exemplo, no timezone `America/New_York`, atualmente, a regra para o horário de verão diz que ele termina no primeiro domingo de novembro, que a mudança ocorre às 02:00 (neste horário, o relógio é atrasado em uma hora), e o offset muda de `-04:00` para `-05:00`. Como esta é a regra atual, assume-se que ela será usada em todas as datas futuras (ou seja, o horário de verão de 2019, 2020, e qualquer outro ano no futuro, todos seguirão esta regra). Mas se houver alguma mudança (o governo local pode decidir mudar o offset, ou abolir o horário de verão, ou mudar a data de início ou fim), a nova regra é introduzida no banco de dados e passa a ser a atual (e futura).

O critério para escolher os nomes é usar a maior cidade de determinada região que possui o mesmo histórico de offsets. Por isso, não há um identificador para `America/Osasco` ou `America/Atibaia`, por exemplo, já que estas cidades sempre seguiram a mesma hora local de São Paulo. Ou seja, o timezone dessas regiões também é `America/Sao_Paulo` (ou, em outras palavras, o timezone `America/Sao_Paulo` também engloba as cidades de Atibaia e Osasco). Se por acaso um dia alguma dessas cidades começar a usar um horário diferente, o histórico de offsets não será mais igual ao de São Paulo e a IANA criará um outro timezone para a região.

Basicamente, é por isso que optei por usar o termo "timezone" em vez de "fuso horário". O termo em português não possui o mesmo significado, pois não leva em conta o histórico de offsets.

Por exemplo, durante parte do ano, São Paulo e Recife possuem a mesma hora local (o mesmo fuso horário). Mas Recife atualmente não adota o horário de verão, então durante estes meses São Paulo está 1 hora adiantado com relação a Recife. Costuma-se dizer que, durante o horário de verão, estas cidades estão em fusos horários diferentes, ou que o fuso de São Paulo mudou por causa do horário de verão.

> Pode ser que você tenha um entendimento diferente quanto ao uso do termo "fuso horário", e essa divergência é mais um motivo para usar o termo em inglês. Optei por esta forma para evitar confusão e até para você se familiarizar com ele, já que muitas linguagens também o usam. Só usei "fuso horário" nas partes iniciais do livro para que o texto não ficasse muito confuso, mas agora que o termo "timezone" foi introduzido, passarei a usá-lo daqui em diante.

Já a abordagem da IANA considera que estas cidades **sempre estão** em dois timezones diferentes (`America/Sao_Paulo` e `America/Recife`), e que durante parte do ano, por acaso, usam o mesmo offset. Mas como os seus históricos são diferentes (um tem horário de verão e muda seu offset, o outro não), então cada um tem seu próprio identificador. Uma diferença sutil, porém muito importante para entender o conceito de timezone.

Se existem dois lugares que atualmente têm o mesmo horário e seguem as mesmas regras (mesmo offset, horário de verão começa e termina no mesmo instante etc.), mas possuem identificadores diferentes no TZDB, significa que em algum momento no passado eles não usavam o mesmo offset. Se o histórico é diferente, nem que seja por apenas um segundo, a IANA considera que é outro timezone.

Um exemplo são `Europe/Berlin` e `Europe/Paris` . Atualmente ambos usam as mesmas regras: o horário de verão começa e termina nos mesmos dias e horários e os offsets usados antes e depois são os mesmos (`+01:00` no horário "normal" e `+02:00` no horário de verão). Mas por que existe um identificador para cada cidade? Porque o histórico não é o mesmo: a França começou a usar o horário de verão em 1976, enquanto a Alemanha só o fez em 1980. Como o histórico não é o mesmo, então são 2 timezones diferentes.

> Para ser mais preciso, a IANA considera que um timezone é uma região onde os horários locais sempre foram os mesmos desde 1970. Mas muitas regiões possuem informações mais antigas e, para fins práticos, podemos considerar que eles possuem "todo o histórico" de uma região.
>
> E de qualquer forma, isso não muda as definições já explicadas: se em algum momento duas regiões possuem alguma diferença nos seus horários locais, então elas são consideradas 2 timezones diferentes.

A IANA lança versões novas do seu banco regularmente. Só em 2018, até maio já haviam sido lançadas 5 versões – média de 1 por mês – o que mostra que essas coisas mudam com mais frequência do que imaginamos. Você pode se cadastrar na lista de e-mails para receber um aviso sempre que sai uma versão nova: https://mm.icann.org/mailman/listinfo/tz-announce/. Esta lista não é muito movimentada, pois só são enviados os anúncios de novas versões. Mas se você quiser receber **muitos** e-mails, pode se cadastrar na lista de discussão: https://mm.icann.org/mailman/listinfo/tz/.

Durante este livro usaremos os identificadores da IANA. Alguns exemplos utilizarão mudanças recentes, como a da Coreia do Norte, já citada anteriormente. A versão que usarei é a `2018e` , mas sempre verifique o site da IANA (https://www.iana.org/time-zones) para saber qual é a mais recente. Versões anteriores podem ser baixadas em https://data.iana.org/time-zones/releases/. E você também pode contribuir, seja escrevendo código, seja informando que seu governo resolveu mudar o horário local da região onde você mora, pois o código está todo no GitHub (https://github.com/eggert/tz/).

3.1 Abreviações de timezone e suas limitações

Em muitos lugares, é comum o uso de abreviações para designar o timezone. Nos EUA, por exemplo, eles costumam usar nomes como EST (*Eastern Standard Time*, usado na costa leste) e PST (*Pacific Standard Time*, usado na costa oeste).

Porém, estas abreviações não são timezones de fato, já que o nome EST não encapsula todo o histórico de offsets daquela região. Na verdade, durante o horário de verão, o nome muda para EDT (*Eastern Daylight Time*). Ou seja, EST e EDT são no máximo "apelidos" para os offsets usados na costa leste dos EUA, quando estão no horário "normal" (`-05:00`) e no horário de verão (`-04:00`), respectivamente.

Também há o costume de usar o nome ET (*Eastern Time*) para representar a junção de EST e EDT. Mas ainda assim isso não é um timezone, pois é um nome abrangente demais, cobrindo até outros países. Além disso, há regiões do *Eastern Time* que adotam o horário de verão (como New York) e outras que não o adotam (como o Panamá). Por cobrir várias regiões com históricos de offsets totalmente distintos, o *Eastern Time* não é um timezone, e sim um nome comum que é adotado em várias regiões com timezones diferentes.

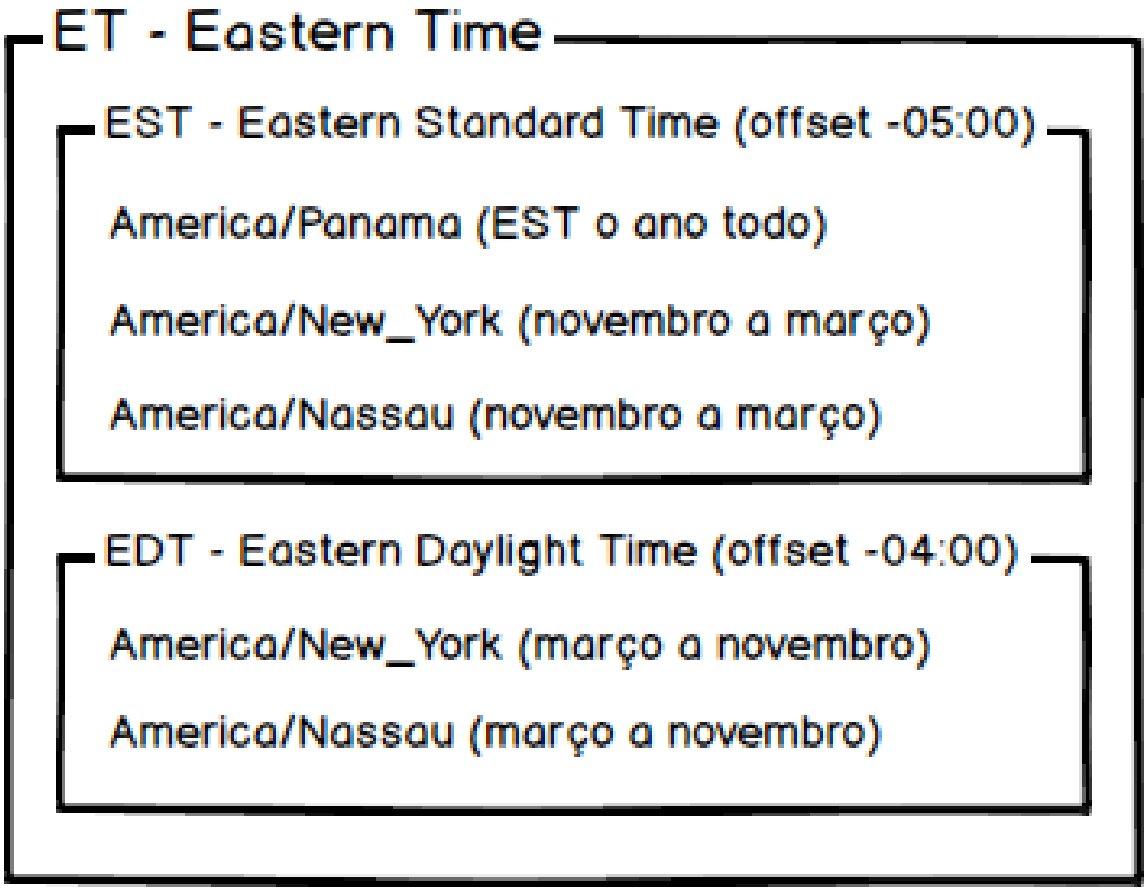

Figura 3.1: Abreviações como ET, EST e EDT não representam um único timezone

Outro problema das abreviações é que muitas são ambíguas. IST, por exemplo, pode ser:

- *India Standard Time*: corresponde ao timezone `Asia/Kolkata` e atualmente usa o offset `+05:30`
- *Israel Standard Time*: corresponde ao timezone `Asia/Jerusalem` e atualmente usa o offset `+02:00` (no horário de verão, o offset muda para `+03:00` e o nome passa a ser IDT, *Israel Daylight Time*)
- *Irish Standard Time*: corresponde ao timezone `Europe/Dublin` e atualmente usa o offset `+01:00` , e é usado quando a Irlanda está em horário de verão (pois é, apesar de ter *Standard* no nome, é usado durante o horário de verão)

Há vários outros casos de abreviações ambíguas, como CST (que é usada em Cuba, EUA e China). Esta resposta do Stack Overflow tem mais alguns exemplos de abreviações problemáticas, e por fim recomenda o uso do TZDB da IANA (https://stackoverflow.com/a/18407231/).

Por serem ambíguas e nem sempre representarem uma única região com um histórico de offsets distinto, as abreviações não são consideradas timezones, de fato. Apesar disso, algumas APIs as aceitam e assumem alguns valores arbitrários para elas. Por exemplo, há APIs que podem receber o valor "IST" e retornar um dos 3 timezones que usam esta abreviação: `Asia/Kolkata` (Índia), `Asia/Jerusalem` (Israel) ou `Europe/Dublin` (Irlanda) – e torça para que ela retorne o que você precisa. Outras APIs permitem que você pelo menos escolha qual timezone usar em caso de ambiguidade.

Portanto, é possível mapear um timezone para uma abreviação (`America/New_York` pode ser EDT ou EST, se estiver em horário de verão ou não, respectivamente), mas o oposto não é possível, a menos que você faça alguma escolha arbitrária (como escolher se IST será mapeado para Índia, Israel ou Irlanda).

3.2 Timezone e Offset não são a mesma coisa

Em muitos lugares, seja em artigos, documentação de API ou nomes de métodos, você vai ver um offset (como `-03:00` ou `+02:00`) sendo chamado de timezone. Mas isso está **errado**, porque eles não são a mesma coisa.

Um offset é simplesmente a diferença em relação a UTC, nada mais que isso. O offset `-03:00` é e sempre vai ser 3 horas antes de UTC, `+05:30` é e sempre vai ser 5 horas e meia depois de UTC, e assim por diante. Um offset é um valor fixo.

Um timezone é o histórico de todos os offsets que uma região teve, tem e terá em sua história. Ele tem um identificador único, como `America/Sao_Paulo` ou `Europe/London` , e a lista de offsets também possui o instante exato em que cada mudança ocorre, e quais os offsets antes e depois da mudança.

Dizer que um offset é igual a um timezone está errado, apesar de muita gente achar que eles são a mesma coisa. É só ver a quantidade de perguntas no Stack Overflow sobre este assunto, com pessoas confundindo os conceitos ou achando que não há diferença entre eles. Há centenas de perguntas como esta (https://stackoverflow.com/q/16467114/), que mostra como esta dúvida é comum entre desenvolvedores.

O problema é que muitas linguagens e APIs acabam caindo neste erro, como a citada neste link (https://stackoverflow.com/q/16069846/): a API retorna um campo chamado `timeZone` , mas na verdade o valor é um offset (`-07:00`). Este tipo de erro, além de causar confusão, contribui para disseminar a noção errada de que timezones e offsets são a mesma coisa.

Sempre leia a documentação das APIs que você usa, e se aparecer algo como `-03:00` ou `+02:00` , e for chamado de timezone, você já saberá que está errado, pois estes são todos offsets. Vale lembrar de que o `Z` também não é um timezone, é apenas uma indicação de que a data e hora está em UTC, e significa que o offset é zero (mas também pode ser escrito como `+00:00`).

Outra dúvida que muitos têm – praticamente toda semana no Stack Overflow surge uma pergunta parecida – é: dado um offset, por exemplo `+02:00` , como saber o timezone?

A resposta pode ser um pouco frustrante, caso você esteja esperando apenas um único timezone. Porque, dado um offset, o máximo que você pode ter é uma lista de timezones. E ainda assim,

esta lista também pode variar conforme a data e hora usadas como referência.

Lembre-se de que um timezone é um histórico de diferentes offsets que cada região teve, tem e terá durante sua história, e por isso não tem um único offset. Para cada data e hora diferente, seja no passado, presente ou futuro, você pode ter um offset diferente: São Paulo, por exemplo, pode ter offset `-03:00` ou `-02:00` , pois dependendo da data e hora, pode estar em horário de verão ou não – e em cada ano ele começa e termina em um dia diferente. Sem uma data e hora de referência, não há como saber o offset utilizado por um timezone.

Voltando à nossa pergunta, dado o offset `+02:00` , como saber o timezone?

Depende.

A resposta depende da data e hora que você usa como referência. Por exemplo, se você usar "4 de maio de 2018, às 17:00 em São Paulo", terá uma lista de 57 timezones que usam o offset `+02:00` neste dia e horário. Mas se eu mudar a data de referência para "31 de outubro de 2018, às 17:00 em São Paulo", a lista tem somente 44 timezones.

> Veja o método `algunsTimezonesComOffset0200()` na classe `exemplos.part1.Main` no GitHub (https://github.com/hkotsubo/java-datetime-book/blob/master/src/main/java/exemplos/part1/Main.java/) para ver um exemplo.

Isso acontece porque os 57 timezones que em maio usam o offset `+02:00` não são necessariamente os mesmos que usam este offset em outubro. Um exemplo é `Africa/Ceuta` , que em maio está em horário de verão (offset `+02:00`), mas em outubro está em horário "normal" (offset `+01:00`). Outro exemplo é `Asia/Amman` , que em maio

também está em horário de verão, mas com offset `+03:00` , e em outubro está em horário normal e com offset `+02:00` . E há outros que atualmente não têm horário de verão e usam o offset `+02:00` o ano todo, como `Africa/Cairo` (mas se você usar como referência alguma data em agosto de 2014, este timezone não vai aparecer na lista, pois neste ano o Cairo teve horário de verão e o offset utilizado era `+03:00`).

Timezones com offset +02:00

	2014-08-10 Agosto de 2014	2018-05-04 Maio de 2018	2018-10-31 Outubro de 2018
Africa/Ceuta	**+02:00** **Horário de verão**	**+02:00** **Horário de verão**	+01:00 Horário normal
Asia/Amman	+03:00 Horário de verão	+03:00 Horário de verão	**+02:00** **Horário normal**
Africa/Cairo	+03:00 Horário de verão	**+02:00** **Horário normal**	**+02:00** **Horário normal**

Figura 3.2: Dependendo da data e hora, a lista de timezones com o mesmo offset pode variar

Conclusão: dado um offset, não é possível ter um único timezone. O máximo que você vai ter é uma lista de timezones, e mesmo assim esta lista pode variar de acordo com a data e hora que você usar como referência. Tendo esta lista, você pode tomar a decisão mais apropriada, dependendo do seu caso de uso: escolher um arbitrariamente, deixar o usuário escolher, usar algum critério qualquer para escolher algum deles etc.

E, dado um timezone, também não é possível ter um único offset, pois precisamos de uma data e hora de referência, para saber qual é o offset usado naquele momento. Na terceira parte do livro veremos como lidar melhor com estes casos, usando a API `java.time` , que possui mecanismos avançados para obtermos todas estas informações de um timezone.

Algumas APIs podem ter uma função que retorna o offset de um timezone, sem receber uma data e hora como referência. É importante ler a documentação para saber o que esta função faz. Pode ser que ela use a data e hora atual, por exemplo. Mas pode ser que ela retorne o offset baseado em algum outro critério – e, se for uma API minimamente decente, ela explicará este critério – e é importante você saber o que ela está fazendo, para poder tomar a decisão correta com relação ao valor retornado.

> Só mais um detalhe: no exemplo anterior, estou usando a versão 2018e do TZBD (https://data.iana.org/time-zones/releases/tzdata2018e.tar.gz/) para verificar os timezones que usam o offset `+02:00` . Se você usar outra versão que tenha alguma informação diferente para algum dos timezones em questão, pode ser que você obtenha resultados diferentes. As regras de timezones mudam toda hora, e nada garante que ficarão assim para sempre. E a IANA não atualiza somente mudanças futuras, pois muitas vezes eles também corrigem informações do passado, então qualquer resultado pode mudar dependendo do quão atualizadas estão as informações de timezone.

Lembre-se: informações de timezone mudam o tempo todo. Nada é garantido, nada é para sempre. Nunca assuma nada.

Timezones não são padronizados

Agora você já sabe o que é um timezone, e nunca mais vai confundi-lo com um offset. E não sei se você está tão espantado quanto eu, ao saber que não há uma padronização universal, nem uma norma ISO, nem nada parecido, para uniformizar todas as informações sobre timezones.

Os bancos de dados da Microsoft e da IANA são apenas um conjunto de informações compiladas e mantidas por pessoas e

corporações, mas estão longe de serem uma norma. O TZDB da IANA é mais difundido simplesmente por ser a fonte mais confiável e completa que temos – e muitos consideram que, na prática, ele é o padrão a ser seguido.

Outra coisa que não é nem um pouco padronizada são as formas como cada país escreve suas datas. Alguns usam `dd/mm/aaaa` (dia/mês/ano), enquanto outros usam `aa/mm/dd` , ou ainda `mm/dd/aaaa` . Há lugares que usam traços ou pontos em vez de barras para separar os campos, ou colocam o nome do mês abreviado, ou usam sufixos depois dos dias (1st, 2nd, 3rd). As variações parecem não ter fim.

Felizmente, apesar de todos estes formatos diferentes, existe uma norma para (tentar) uniformizar esta bagunça. É o que veremos em seguida.

3.3 Datas não têm formato

Repita comigo: **"Datas não têm formato"**. Elas podem **estar** em um formato, mas não possuem nenhum.

Vamos usar como exemplo a data de 4 de maio de 2018. A data em si (dia 4, do mês 5, que em português chama-se maio, do ano de 2018 do calendário gregoriano) é uma ideia, um conceito. Ela representa um ponto específico no nosso calendário, uma forma de nos situarmos no tempo, com valores numéricos atribuídos de acordo com determinadas regras. Mas a data, por si só, não possui um formato.

O que podemos fazer é **representar** esta data em diferentes formatos. Por exemplo:

- 04/05/2018 – dia, mês e ano, muito comum em vários países
- 5/4/18 – mês, dia e ano, o terrível formato americano

- Sexta-feira, 4 de maio de 2018 – em bom português
- May, 4th 2018 – em inglês, com o sufixo "th"
- e muitos outros...

Repare que cada um desses formatos é diferente, mas todos representam a mesma data (a mesma ideia, o mesmo ponto no calendário). O valor é o mesmo, a representação deste valor não. É mais ou menos o que fazemos com números, por exemplo. O número 2 representa uma ideia: o conceito de uma determinada quantidade (duas coisas). Mas este valor pode ser representado de várias maneiras: "02", "2.0", "Dois", "Two" etc. Apesar dos formatos serem diferentes, eles representam a mesma ideia, o mesmo valor. Com datas é a mesma coisa.

Alguns formatos podem ser mais comuns do que outros, mas isso não faz com que as datas **tenham** um formato específico. Repita mais uma vez: **datas não têm formato**. No fim, os formatos são apenas formas de se representar uma data, de expressar um conceito abstrato, de transformar um ponto arbitrário no calendário – com regras igualmente arbitrárias para definir seus valores – em um texto (uma sequência de caracteres que faz sentido e torna possível entendermos que se trata daquele dia, mês e ano).

Isso tudo acaba gerando um problema. Muitos sistemas conversam entre si trocando basicamente texto – seja um HTTP POST, um JSON, um XML, o que for, no fim tudo isso vira texto, inclusive as datas. E combinar um formato para transmitir datas sempre foi – e em muitos sistemas ainda é – um problema, principalmente se duas culturas diferentes estiverem envolvidas: um americano vai ter a tendência de colocar o mês na frente, por exemplo, e achar que 04/05/2018 é 5 de abril, quando na verdade é 4 de maio. Todos temos a tendência de usar um formato que nos é mais familiar, o problema é que há formatos demais no mundo, e nem todos concordam entre si.

3.4 O formato ISO 8601

Felizmente, ao contrário do que acontece com os timezones, existe um padrão que define um formato específico e não ambíguo para intercâmbio de informações relacionadas a data e hora. Este formato é definido pela norma ISO 8601 (https://en.wikipedia.org/wiki/ISO_8601/).

> O documento oficial da ISO 8601 é pago, mas o artigo da Wikipedia tem informações suficientes para termos um bom entendimento da norma, e atende à maioria dos casos de uso.

A data que estamos usando (4 de maio de 2018) ficaria assim de acordo com a ISO 8601: `2018-05-04` (ano com 4 dígitos, hífen, mês com 2 dígitos, hífen, dia com 2 dígitos). É um formato que evita qualquer ambiguidade (como a que pode ocorrer com 04/05/2018, por exemplo), pois existe uma ordem bem definida que não depende de fatores culturais.

Os campos sempre estão nesta ordem, com esta quantidade de dígitos. Quando o valor do campo tiver menos dígitos (dia e mês menor que 10, ou ano menor que 1000), ele é preenchido com zeros à esquerda – no nosso exemplo, o mês 5 foi escrito como `05` e o dia 4, como `04` . Se o ano fosse 900, seria escrito como `0900` .

E as horas?

Neste formato, eu também posso representar apenas o horário, como `17:00` . As horas sempre têm valores de 00 (meia-noite) até 23 (11 da noite) – não há os famosos "AM" e "PM" dos americanos. Os campos das horas, minutos e segundos sempre são escritos com dois dígitos, e, se o valor for menor que 10, é completado com um zero à esquerda.

Note que no exemplo anterior eu só coloquei as horas e minutos (`17:00`), pois os segundos podem ser omitidos caso o valor seja zero. Também são permitidas as frações de segundo, sem um limite de casas decimais. Exemplo: `17:30:10.188499274` representa 17 horas (ou 5 da tarde), 30 minutos, 10 segundos e 188499274 nanossegundos.

Juntando data e hora

Quando houver uma data e uma hora, elas devem ser separadas pela letra `T` , sempre maiúscula. Por exemplo, 4 de maio de 2018 às 5 da tarde ficaria `2018-05-04T17:00` – a data (`2018-05-04`) fica separada das horas (`17:00`) pela letra `T` .

Este formato, com os separadores `-` e `:` , é chamado de formato estendido. Também é possível usar o formato básico, sem os separadores, então esta mesma data e hora seria escrita como `20180504T1700` – note que ainda assim é preciso ter o `T` separando a data e a hora. Eu particularmente acho um pouco confuso e difícil de ler, e prefiro o formato estendido.

Anteriormente eu disse que as horas podem ter valores de 00 a 23, mas também é possível ter o valor 24, que representa "o fim do dia". Na prática, é o mesmo que "meia-noite do dia seguinte". Ou seja, `2018-05-04T24:00` (fim do dia 4 de maio de 2018) é equivalente a `2018-05-05T00:00` (início – ou meia-noite – do dia 5 de maio de 2018).

Na minha opinião, é um caso especial que não acrescenta muito, já que o valor `00:00` já seria suficiente para designar a meia-noite de maneira não ambígua. Mas de qualquer forma, é interessante saber que é um valor válido de acordo com a norma.

Não se esqueça do offset

Por último, também podemos acrescentar o offset, sempre no final, após a data e hora. Um offset de 3 horas atrás de UTC, como já vimos anteriormente, pode ser representado como `-03` , `-0300` ou `-03:00` . Para representar o UTC, usa-se a letra `Z` , ou então `+00:00` –

mas o `Z` é o mais comum. Então, 4 de maio de 2018, às 17h em São Paulo, seria representado como `2018-05-04T17:00-03:00` (neste dia, o offset é `-03:00` , pois em maio São Paulo não está em horário de verão).

Esta mesma data em UTC equivale a `2018-05-04T20:00Z` : como São Paulo está com o offset `-03:00` (3 horas antes de UTC), basta eu somar 3 horas para obter o valor correspondente em UTC. Conforme já vimos, esta mesma data poderia ser escrita como `2018-05-04T20:00+00:00` , mas para representar o UTC, o mais comum é usar o `Z` .

> Um detalhe é que, para o offset zero, somente o sinal de positivo deve ser usado (`+00:00` , `+0000` ou `+00`). Portanto, escrevê-lo como `-00:00` não é permitido pela norma ISO 8601. Apesar disso, existe a RFC 3339 que define o conceito de "offset desconhecido", que é representado por `-00:00` , o que na minha opinião é uma definição bem confusa (https://tools.ietf.org/html/rfc3339#section-4.2/).

Quando você precisar enviar uma data em forma de texto para outro sistema, prefira o formato ISO 8601, que não é ambíguo e muitas APIs já reconhecem por padrão. A partir de agora passarei a usar este formato nos exemplos, para que você se habitue a ele.

Apesar de ser muito usado e recomendado, o ISO 8601 é apenas mais um dos muitos formatos possíveis para representar uma data. Mas isso não muda o fato de que – repita novamente – **datas não têm formato**.

Claro que datas têm um formato, estou vendo um agora

Talvez você esteja achando estranho que datas não tenham um formato – e elas, de fato, não têm – afinal de contas, *"Quando eu consulto a data no banco de dados, ela aparece no formato*

dd/mm/yyyy", ou "*Eu imprimi a data, e apareceu* `Fri May 04 17:00:00 BRT 2018` *, isso não é um formato?"*.

Apenas para esclarecer, ou reforçar o que foi dito anteriormente, a data em si só possui valores. Geralmente são valores numéricos que equivalem ao ponto específico do calendário que aquela data representa. Mas quando você imprime a data, mostra na tela, salva em um arquivo ou vê seu valor em um *debugger*, aí ela tem que ser apresentada de alguma maneira: algum formato é escolhido, seja pela API, pelo banco de dados, ou o que quer que você esteja usando para imprimi-la.

Mas isso não quer dizer que a data tem aquele formato. Só porque você fez um `SELECT` no banco e a data apareceu como `04/05/2018`, isso não significa que ela está necessariamente salva naquele formato, principalmente se o campo for de um tipo relacionado a data (como `DATE`, `TIMESTAMP`, entre outros). Cada banco de dados vai salvar a data de alguma maneira que não importa para nós, pois é um detalhe interno de sua implementação. Mas quando esta informação é exibida na tela, por exemplo, o banco vai escolher algum formato para mostrar a data, e isso não quer dizer que a data em si possui aquele formato. Na verdade, ela não possui nenhum.

O formato não é uma característica intrínseca da data, não faz parte dela. Já os valores, como o dia, mês e ano, estes sim são parte da data, pois são eles que definem o que ela, de fato, representa (um ponto específico no calendário). O formato é apenas uma maneira de representar a data, de forma que seus valores façam sentido para quem a vê.

Por mais que você acredite que datas têm formato, acostume-se com o fato de que elas não têm.

3.5 Parsing e formatação

Hoje em dia, praticamente toda linguagem de programação possui um ou mais tipos de data. Procure na API da sua linguagem favorita e provavelmente você vai encontrar algum tipo chamado `Date`, `DateTime`, `Timestamp` ou algo parecido. Se não tiver um tipo nativo, com certeza já deve existir alguma biblioteca.

Toda linguagem também possui um outro tipo para armazenar texto (uma sequência de caracteres), provavelmente uma `String` ou um array de `char` (ou `VARCHAR`, como é mais comum nos bancos de dados).

Quando eu digo que uma data não tem formato, também estou me referindo aos tipos de data das linguagens. Normalmente, eles são implementados de forma que só contenham valores (geralmente os números que correspondem ao dia, mês, ano, hora, minuto e segundo, ou o valor numérico do timestamp), mas que não dependam de nenhum formato específico. São só os valores em si.

Quando estes tipos de data são impressos, eles são transformados em texto, e aí sim será escolhido um formato, para que os valores da data sejam convertidos para uma `String`. O processo de transformar um tipo de data em uma `String` é chamado de **formatação**.

Por exemplo, a data de 4 de maio de 2018 pode ser formatada para várias `Strings` diferentes, cada uma com um formato específico:

- `"04/05/2018"`
- `"May, 4th 2018"`
- `"2018-05-04"`
- etc.

Já o oposto, quando eu tenho uma `String` que representa uma data, em um determinado formato, e quero transformá-la em um tipo de data, é chamado de *parsing*.

> As traduções que encontrei para *parsing* não são satisfatórias (análise, interpretação), então usarei o termo em inglês. Porém, não conjugarei o verbo, para evitar expressões como *parsear* ou *parseado*. Nestes casos, usarei "fazer o parsing", "foi feito o parsing" e variações.

Os conceitos de formatação e parsing são importantes, pois muitas APIs possuem métodos com nomes como `format` e `parse` , que fazem exatamente isso: `format` transforma uma data em `String` , e `parse` faz o processo inverso. Mesmo que os métodos não tenham estes nomes, o conceito é o mesmo: se uma data é transformada em texto, é uma formatação, se o texto é transformado em data, é um parsing.

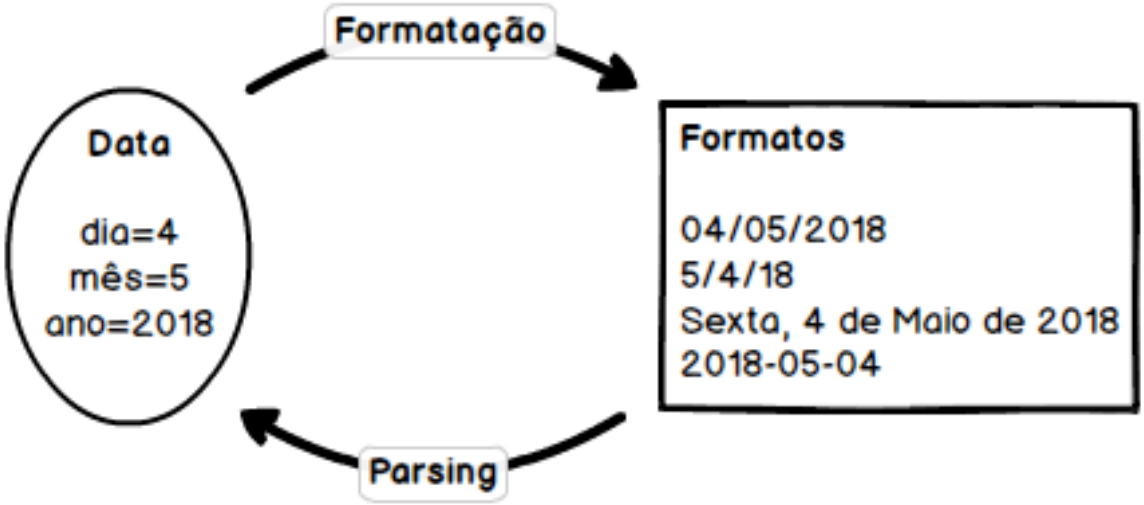

Figura 3.3: Formatação e parsing, não confunda!

Praticamente todos os dias no Stack Overflow surge pelo menos uma pergunta sobre isso, e na maioria das vezes o erro ocorre porque a pessoa está tentando fazer um `format` quando deveria estar fazendo um `parse` , ou vice-versa.

Também é comum ver casos em que a pessoa faz um `parse` depois de um `format` , ou seja, transforma uma data em `String` para depois

transformar em data novamente, o que geralmente é inútil e redundante, pois o resultado final quase sempre é a mesma data passada no início. Claro que há casos e casos, e é importante entender os conceitos para saber quando isso é necessário ou não.

Na segunda e terceira parte do livro, teremos muitos exemplos com código, e os conceitos de formatação e parsing ficarão ainda mais claros.

Existe uma outra forma, também muito usada, para representar uma data e hora. Não é exatamente um formato, apenas uma outra maneira de referir-se ao mesmo valor. É aquele número gigante (`1525464000000`) que vimos no começo do livro. No próximo capítulo veremos o que esse número significa.

Capítulo 4
O Unix Epoch e os timestamps

Antes de explicar o que é aquele número gigante (`1525464000000`), precisamos saber o que é o *Unix Epoch*.

Comecemos pela palavra *epoch*, que neste caso não significa "época". É claro que esta é uma tradução válida, mas *epoch* também tem outros significados em inglês, e um em particular é o que nos interessa:

- *An instant of time or a date selected as a point of reference*

Em tradução livre: "Um instante ou uma data escolhida como ponto de referência" (https://www.merriam-webster.com/dictionary/epoch/).

Ou seja, *epoch* é um ponto específico na linha do tempo (uma data e hora com valor bem definido), que é usado como o ponto de referência a partir da qual todos os outros valores se baseiam. Em computação, existem várias datas diferentes usadas como ponto de referência (vários *epochs*), cada um sendo usado em uma situação distinta (https://en.wikipedia.org/wiki/Epoch_(reference_date)#Notable_epoch_dates_in_computing/).

Um destes é o Unix Epoch, que é definido como `1970-01-01T00:00Z` – 1 de janeiro de 1970, à meia-noite, em UTC. Este instante específico é definido com o valor **zero**, e qualquer outra data e hora é definida como uma quantidade de tempo decorrida a partir do Unix Epoch.

Nos sistemas Unix, esta quantidade era medida em segundos, mas hoje em dia muitos sistemas usam a quantidade de milissegundos (também conhecidos como "milésimos de segundo"). O nosso número gigante (`1525464000000`), por exemplo, está em milissegundos. Ou seja, ele representa o instante que ocorre 1525464000000 milissegundos depois do Unix Epoch, e é

equivalente a `2018-05-04T20:00Z` (4 de maio de 2018, às 20:00 em UTC).

Mais do que isso: este número representa um único instante, um **ponto na linha do tempo**. Imagine que o tempo é uma linha contínua, e cada ponto representa um único instante (um único valor para o timestamp):

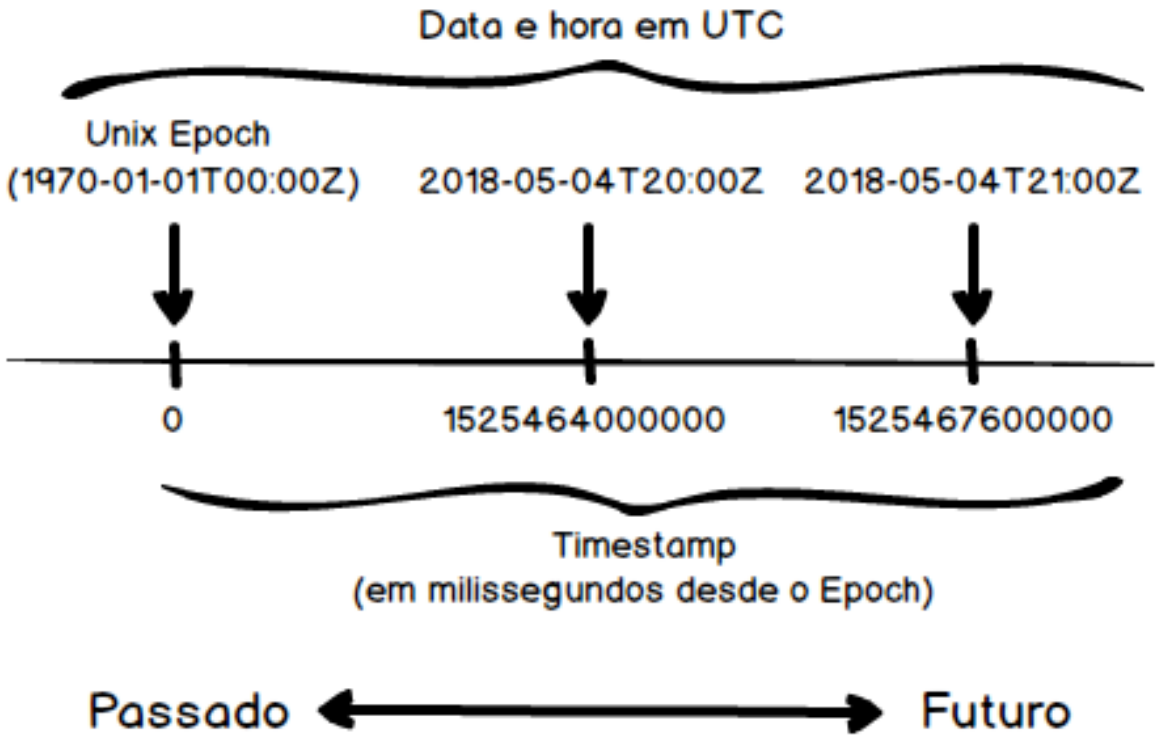

Figura 4.1: Linha do Tempo

Cada timestamp representa um único ponto na linha do tempo. Quando caminhamos para a direita, os valores dos timestamps aumentam, representando datas cada vez mais no futuro. Ao caminhar para a esquerda, os valores diminuem, representando datas cada vez mais no passado.

Um detalhe deste número (`1525464000000`) é que ele tem o mesmo valor no mundo todo, não importando o timezone em que você está. Qualquer computador, em qualquer parte do planeta, que verificasse

a data/hora atual naquele exato momento receberia `1525464000000` como resultado. O que os computadores geralmente fazem, ao mostrar este valor, é convertê-lo para uma data e hora em um timezone específico.

Este é um ponto-chave deste número: ele não depende de timezones, nem de suas regras malucas que mudam o tempo todo. Seu valor não muda, é o mesmo para todos. Só que este mesmo valor pode corresponder a uma data e hora diferente, de acordo como o timezone utilizado. Vejamos qual é a data, hora e offset correspondente ao valor `1525464000000` , em diferentes timezones:

Timezone	Data/hora e offset para `1525464000000`
America/Sao_Paulo	2018-05-04T17:00-03:00
Europe/Berlin	2018-05-04T22:00+02:00
Asia/Tokyo	2018-05-05T05:00+09:00
Pacific/Honolulu	2018-05-04T10:00-10:00

Apesar de ter uma data, hora e offset diferentes em cada timezone, todas as datas da tabela anterior equivalem, em UTC, a `2018-05-04T20:00Z` . Ou seja, todas aconteceram no mesmo instante: **elas são exatamente o mesmo ponto na linha do tempo**. Todas podem ser representadas pelo mesmo número gigante `1525464000000` . E este número representa o instante que ocorre 1525464000000 milésimos de segundo depois do Unix Epoch.

> O número gigante tem vários nomes: *timestamp*, *Unix timestamp*, *Unix Time*, entre outros – só não chame de *Epoch Time* (https://codeofmatt.com/2017/10/05/please-dont-call-it-epoch-time/).
>
> Neste livro, usarei o termo *timestamp*, por ser mais curto, não ter uma tradução satisfatória em português, e também por não existir um nome oficial padronizado (https://stackoverflow.com/a/45469753/).

Um ponto importante ao lidar com timestamps é saber qual unidade está sendo usada. Nos exemplos anteriores, usamos um valor em milissegundos, mas muitas APIs trabalham com este valor em segundos. É importante você ler a documentação da linguagem, banco ou framework que estiver utilizando, para saber se aquele valor está em segundos, milissegundos, dias ou o que for.

Também é importante saber qual o *epoch* utilizado. Apesar de o Unix Epoch ser muito comum, não é o único que existe, e se você usar a referência errada, pode acabar com um valor incorreto para suas datas.

> Um exemplo de API que usa um *epoch* e unidades diferentes é o .NET: a classe `DateTime` usa o número de intervalos de 100 nanossegundos decorridos desde 1 de janeiro do ano 1, à meia-noite em UTC (ou `0001-01-01T00:00Z`). Cada intervalo de 100 nanossegundos é chamado de *Tick* (https://msdn.microsoft.com/pt-br/library/system.datetime.ticks(v=vs.110).aspx/).

O timestamp também pode ser negativo, indicando que se refere a uma data anterior ao Unix Epoch. Por exemplo, o timestamp `-100000000000` (menos 1 trilhão) indica um instante ocorrido 1 trilhão de milissegundos antes do Unix Epoch, e corresponde, em UTC, a

`1938-04-24T22:13:20Z` (e em cada timezone, esse mesmo valor vai corresponder a uma data, hora e offset diferentes).

Os timestamps baseados no Unix Epoch são muito usados, e várias linguagens, APIs e frameworks possuem suporte a eles, sendo capazes de convertê-los de/para datas e horas. Consulte a documentação para saber se eles são suportados, qual o *epoch* usado e em qual unidade deve estar o valor do timestamp (segundos, milissegundos, dias etc.).

4.1 Tenho uma data, como calcular o timestamp?

Depende.

Suponha que eu tenho uma data (`2018-05-04` – 4 de maio de 2018) e quero calcular o timestamp correspondente (a quantidade de milissegundos deste o Unix Epoch). Como eu faço?

Lembre-se de que o timestamp representa um único instante, um ponto específico na linha do tempo. Se eu tenho apenas o dia, mês e ano, não é o suficiente para calcular o timestamp, pois um dia representa um período de várias horas. Ou seja, vários instantes, vários pontos na linha do tempo.

Eu também devo definir um horário, caso contrário não será possível calcular um único timestamp. Dependendo do valor que você escolher, vai obter um timestamp diferente. Vamos definir que o horário é 17 horas. Então agora eu quero saber o timestamp correspondente a `2018-05-04T17:00` (4 de maio de 2018, às 17:00).

Ainda não é possível. Como vimos nos capítulos anteriores, em cada parte do mundo, a data e hora de "4 de maio de 2018, às 17:00" ocorreu em um instante diferente. Para que este dia e horário sejam mapeados para um único ponto na linha do tempo, precisamos saber de qual timezone estamos falando. Mais que isso,

precisamos saber o offset exato – lembre-se dos casos de *DST overlap* (que vimos no capítulo sobre timezones), quando uma hora local existe duas vezes, uma em cada offset. Em casos de *overlap*, precisamos saber de qual offset estamos falando.

Somente tendo a data, hora e offset, podemos calcular o timestamp. E, dependendo do offset, você vai obter um timestamp diferente. Se eu escolher o timezone `America/Sao_Paulo` , por exemplo, a data fica `2018-05-04T17:00-03:00` (em maio de 2018, São Paulo não está em horário de verão, e o offset é `-03:00`). Com a informação completa, é possível mapear este valor para um único ponto na linha do tempo, e com isso podemos calcular o valor do timestamp (cada linguagem/API vai ter seu próprio jeito de fazer isso). No caso deste exemplo, o valor (em milissegundos) é `1525464000000` .

Algumas APIs podem calcular o timestamp com informações incompletas, por exemplo, tendo só a data, ou tendo a data e hora mas não o offset. Mas na verdade estas APIs estão usando algum valor padrão para os campos que faltam (como usar o timezone que estiver configurado no servidor, usar uma data ou horário específicos etc.), e se for uma API minimamente decente, estas regras estarão devidamente documentadas.

4.2 Devo usar sempre timestamp/UTC?

O valor do Unix timestamp é diretamente associado ao UTC. Afinal, o *epoch* ao qual ele se refere é definido em termos de UTC (`1970-01-01T00:00Z` - 1 de janeiro de 1970, à meia-noite, em UTC). Então, nada mais natural do que achar que o timestamp está em UTC (ou que ele é um valor em UTC, ou algo do tipo).

Tecnicamente falando, o timestamp é apenas um valor numérico que indica a quantidade de tempo decorrida a partir do Unix Epoch. Esse mesmo valor pode ser convertido para uma data e hora

específica, e esta data e hora pode estar em UTC ou em qualquer outro timezone.

A vantagem do timestamp é que ele representa um instante único, não ambíguo. Seu valor (o número gigante) não muda de acordo com o timezone. Não há casos estranhos como o *DST overlap*, no qual um horário local pode existir duas vezes no mesmo dia, por exemplo. Também não há saltos, como acontece quando há um *DST gap*: o valor sempre aumenta de 1 em 1. E por ser um valor numérico, é muito fácil para um computador manipulá-lo, fazer comparações, ordenar etc.

O UTC também não sofre interferências de horário de verão e não é afetado pelas regras malucas dos timezones. Não há *gaps* nem *overlaps*, como ocorre com os timezones. E como cada instante em UTC pode ser mapeado para apenas um único valor de timestamp, é comum que muitos pensem que eles são exatamente a mesma coisa.

Talvez eu esteja sendo preciosista demais nas definições, mas é importante saber que um valor de timestamp representa uma (e somente uma) data e hora específica em UTC, só que ele também pode ser convertido para qualquer timezone (resultando em uma data e hora diferente). O mesmo acontece com o UTC: ele só pode ser mapeado para um único valor de timestamp, mas também pode ser convertido para qualquer timezone.

Por isso, os conceitos de timestamp e UTC são frequentemente confundidos como sendo uma coisa só, mesmo não sendo. UTC define valores de data e hora no offset zero, sem *gaps* e *overlaps*, sem horário de verão, nada disso. E todos os timezones definem seus respectivos dias e horários locais baseados na diferença para o UTC (que chamamos de "offset"). Já o timestamp é um valor numérico que representa uma quantidade de tempo decorrida a partir do Unix Epoch. É apenas um número, sem qualquer noção de dia, hora ou offset.

Mas talvez isso não importe muito, pois dizer que "o timestamp sempre está em UTC" não é algo que vai "quebrar" seu sistema, e provavelmente a maioria vai entender o que você quer dizer.

De qualquer forma, existe uma recomendação geral — que tem se tornado um "mantra" em muitos casos — que é: **"Sempre grave suas datas em UTC"** (ou "Sempre grave o timestamp").

"Sempre" é uma palavra muito forte. E como já vimos em vários exemplos deste livro, quando se trata de datas e horas, para muitas perguntas a resposta é "Depende". E com este caso, não é diferente.

UTC é muito bom para eventos passados

Para gravar o instante exato em que algo aconteceu, o melhor é usar um timestamp (ou a respectiva data e hora em UTC).

Se você usar uma data e hora e não informar o timezone ou offset, acabará com um valor impreciso. Se eu gravar somente `2018-05-04T17:00` , não saberei o momento exato ao qual esta data se refere, já que em cada parte do mundo "4 de maio de 2018 às 17:00" ocorreu em um instante diferente. Se não houver nenhuma indicação quanto ao lugar ao qual esta data se refere (seja o timezone, o nome da cidade, o que for), não teremos como saber com exatidão qual o timestamp correspondente.

Pode ser que esta informação (somente a data e hora, não importa o local) seja suficiente para seus casos de uso e não há nenhum problema quanto a isso — cada caso é um caso. Mas se você precisar saber o instante exato, é necessário ter a informação completa (data, hora e timezone/offset). E uma das maneiras recomendadas é guardar o timestamp, ou a respectiva data e hora em UTC.

Caso você precise exibir esta informação em termos de data/hora local para os usuários, basta convertê-la para o timezone correspondente. Mas internamente você trabalha com UTC e/ou

timestamp: operações como ordenar os eventos por data, ou selecionar os que ocorreram há mais de X horas, podem ser feitas com os timestamps, sem se preocupar com todas as regras complicadas dos timezones.

UTC não é tão bom assim para eventos futuros

Suponha que estamos em 2016. O seu sistema permite que os usuários agendem eventos futuros, e um usuário agenda um evento para 31 de outubro de 2018, às 10h, em São Paulo.

Você ouviu falar que o melhor é sempre usar UTC para guardar datas de maneira não ambígua e evitar qualquer problema com os timezones. Então você pega a data e hora inserida pelo usuário (`2018-10-31T10:00`) e verifica qual é o timezone de São Paulo (`America/Sao_Paulo`). Em 2016, as regras deste timezone diziam que o horário de verão começa no terceiro domingo de outubro. Ou seja, em 31 de outubro de 2018, São Paulo já estaria em horário de verão, e por isso o offset usado seria `-02:00` . Então você junta a data/hora com o offset (`2018-10-31T10:00-02:00`), converte para UTC (`2018-10-31T12:00Z`) e grava este valor no banco.

Até aqui, tudo certo. Quando o usuário for consultar este evento, o sistema lê do banco de dados o valor em UTC (`2018-10-31T12:00Z`) e converte para o timezone correspondente — e o nome do timezone pode estar gravado nas preferências do usuário, ou então no próprio evento, por exemplo. O importante é que o valor em UTC representa um instante único e não ambíguo, que pode ser convertido para qualquer timezone sem problema.

Mas aí acontece algo que achamos que nunca acontecerá conosco, mas que no capítulo sobre timezones foi dito que sim, acontece o tempo todo: o governo muda as regras do timezone. No caso, a regra do Horário Brasileiro de Verão foi mudada por um decreto publicado em dezembro de 2017 (http://www.planalto.gov.br/ccivil_03/_ato2015-2018/2017/decreto/D9242.htm/). Este decreto diz que, a partir de

2018, o horário de verão teria início no primeiro domingo de novembro.

Ciente disso, você verifica se a IANA já lançou uma versão com esta atualização. Conforme já falamos anteriormente, você pode se inscrever na lista de e-mails *tz-announce* e ser avisado quando há uma nova versão (https://mm.icann.org/mailman/listinfo/tz-announce/). A nova regra do horário de verão brasileiro foi implementada na versão `2018c` (https://mm.icann.org/pipermail/tz-announce/2018-January/000048.html/). O recomendado é que você use a versão mais recente.

Feitas as verificações no site da IANA, você atualiza as informações de timezone no seu sistema — posteriormente veremos como fazer isso em Java — e agora o timezone `America/Sao_Paulo` já possui a nova regra. Só que agora temos um problema.

Ao consultar a data do evento, o valor em UTC ainda será o mesmo (`2018-10-31T12:00Z`). Mas agora o timezone `America/Sao_Paulo` possui a nova regra, que diz que o horário de verão começa no primeiro domingo de novembro. Ou seja, em 31 de outubro de 2018, o horário de verão ainda não terá começado, e o offset usado neste timezone ainda é `-03:00` . E ao converter o valor em UTC para o offset `-03:00` , o resultado é `2018-10-31T09:00-03:00` (9 da manhã, uma hora antes do horário informado pelo usuário).

Para corrigir os dados já existentes, não basta sair somando uma hora em todos os eventos. Somente aqueles cuja data e hora estão entre o terceiro domingo de outubro (dia 21) e o primeiro domingo de novembro (dia 4) serão afetados. Mesmo assim, você deverá considerar apenas os cadastros que foram feitos antes de você ter atualizado as regras do timezone, pois os eventos cadastrados depois que você atualizou estas regras já estarão com o offset correto.

Para evitar que esta situação ocorra novamente — e vai acontecer, pois não temos como garantir que o governo nunca mais mudará as

regras — o melhor é gravar dois campos separados: um contendo somente a data e hora (`2018-10-31T10:00`) e outro contendo o timezone (`America/Sao_Paulo`). Quando o usuário consultar a data e hora do evento, basta mostrá-las do jeito que estão. Você também pode mostrar o timezone, para deixar claro que aquela data e hora se referem àquele local.

Depois que o evento ocorrer, você pode, por exemplo, gravar um outro campo, contendo a data e hora convertida para UTC (o instante exato em que o evento efetivamente ocorreu). Como o evento já aconteceu, não há mais problema em gravar a data e hora em UTC, pois esta conversão foi feita usando as regras que o timezone possuía naquele momento. Mesmo que a regra mude depois, só afetará eventos futuros.

> Em alguns updates, a IANA corrige informações do passado, principalmente em datas muito antigas sobre as quais é difícil obter dados confiáveis quanto ao offset usado na época. Ou seja, mesmo dados no passado podem mudar ("nada é garantido, nada é para sempre"). Mas de qualquer forma, a abordagem que acabamos de ver funciona muito bem para a maioria dos casos — melhor do que se você seguir a regra de "**SEMPRE** usar UTC".

4.3 Nem sempre você precisa do timestamp

Há alguns casos em que não é necessário guardar o timestamp (ou a data e hora em UTC). Por exemplo, muitos cadastros de usuários possuem a data de nascimento. A menos que seja um sistema de uma maternidade, ou de um cartório (só para ficar em dois exemplos), na maioria dos casos você só quer saber o dia, mês e ano, e nada mais. Não importa o horário, nem o timezone.

Muitos sistemas acabam pegando a data (dia, mês e ano) e setando algum valor arbitrário para as horas (como meia-noite, ou qualquer outro valor) e usando o timezone padrão (seja do servidor, ou o que estiver configurado na linguagem, API, framework ou banco de dados).

Isso pode funcionar na maioria dos casos, desde que se tomem os devidos cuidados para que todos os servidores estejam configurados no mesmo timezone. Mas e se um deles mudar esta configuração (seja por engano, ou por qualquer outro motivo), isso pode causar problemas.

Exemplo: em um sistema o usuário escolhe o dia, mês e ano referentes à sua data de nascimento. Vamos supor que o usuário digitou:

```
dia=10
mes=2
ano=1980
```

Então o sistema pega esses valores e constrói a data. Mas no banco de dados este campo foi definido como um timestamp (cada banco implementa de um jeito, mas vamos supor que é um campo que precisa da informação completa — data, hora e offset).

Como já vimos, apenas a data não é suficiente para termos um único valor de timestamp. Mas tudo bem, o sistema completa as informações que faltam usando valores predefinidos. O horário é setado para meia-noite, e é usado o timezone padrão configurado na aplicação (vamos supor que é `America/Sao_Paulo`). Assim, o valor final data é:

```
// Em 1980, São Paulo não teve horário de verão, por isso o offset é
-03:00
1980-02-10T00:00-03:00

Em UTC: 1980-02-10T03:00:00Z
Timestamp: 318999600000
```

E este é o valor gravado no banco. Sempre que um usuário vai consultar seu cadastro, o timestamp é convertido para o timezone do servidor (`America/Sao_Paulo`), garantindo que a data correta será exibida.

Alguns dias depois, alguém muda o timezone do servidor para `America/Los_Angeles` . Não importa se a mudança foi feita por engano, ou se era algo que realmente deveria ser feito porque outra equipe decidiu, o que importa é que pode acontecer (comigo já aconteceu algumas vezes).

Quando o valor do timestamp for convertido para o timezone `America/Los_Angeles` , a data exibida será `1980-02-09T19:00-08:00` — dia 9, um dia antes da data que o usuário cadastrou.

Neste caso, uma solução melhor seria gravar apenas a data (dia, mês e ano), e assim não depender do timezone. Muitos bancos de dados possuem um tipo específico para conter apenas a data, e cada linguagem também vai implementar isso de uma maneira diferente. Sempre leia a documentação, e escolha o tipo mais adequado para cada caso.

Para que mais serve o timestamp?

Muitos usam o valor do timestamp para calcular a diferença entre duas datas. Ao subtrair os respectivos timestamps, obtemos a quantidade de milissegundos que há entre elas.

Em muitos casos, é uma abordagem válida, mas há várias outras situações em que isso não funciona tão bem. É que a aritmética de datas não é tão óbvia assim, e possui vários casos estranhos e contraintuitivos. E é o que veremos em mais detalhes no próximo capítulo.

Capítulo 5
Durações e a bizarra aritmética de datas

Dadas as 2 frases a seguir:

1. A reunião será às **duas horas** da tarde.
2. A duração deste filme é de **duas horas**.

Repare que ambas têm as palavras "duas horas", porém com significados diferentes.

Na primeira frase, "duas horas" significa um horário: um ponto específico do dia. A reunião não será de manhã, nem no fim da tarde, nem de noite. Será exatamente às duas horas da tarde (14:00).

Já na segunda frase, "duas horas" significa uma duração: uma quantidade de tempo. Não quer dizer que o filme começa em determinado horário. Na verdade, não diz nada sobre horários em si, não diz que horas o filme começa ou termina, e nem se ele vai de fato começar. Só diz quanto tempo ele demora (duas horas de duração).

Este é o conceito de duração: uma quantidade de tempo. Ela não está atrelada a nenhuma data ou hora específica. É simplesmente uma quantidade, sem qualquer relação com calendários, timezones, ou relógios.

5.1 Formato ISO 8601 para durações

Durações não são datas, tanto que a ISO 8601 define um formato diferente para cada um destes conceitos. O formato de data nós já vimos anteriormente, já o de duração é `PnYnMnDTnHnMnS` ou `PnW` .

A letra `P` maiúscula significa *Period* (período, outro nome para "quantidade de tempo") e deve estar sempre no começo. É ela que indica que estamos falando de uma duração, e não de uma data.

A seguir, o `n` pode ser qualquer valor numérico, indicando a quantidade de tempo correspondente à unidade representada pela letra subsequente. No caso, as letras `Y`, `M`, `W` e `D` significam, respectivamente, anos, meses, semanas e dias (são as iniciais destas unidades de tempo em inglês: *years*, *months*, *weeks* e *days*). Por exemplo, uma duração de "1 ano, 5 meses e 13 dias" é escrita como `P1Y5M13D`.

A ISO 8601 define que uma duração pode ter apenas o campo de semanas (`W`) **ou** os demais campos (anos, meses, dias etc.). Durações que misturam `W` com os outros campos, como `P2W3D` (2 semanas e 3 dias), não são válidas de acordo com a norma.

Apesar disso, muitas APIs permitem durações com todos os campos misturados, e a forma como isto é tratado varia de acordo com cada implementação. Por exemplo, uma duração como `P1W1D`, pode ser interpretada como "uma semana e um dia", mas algumas APIs convertem automaticamente para "8 dias" (`P8D`). Sempre leia a documentação para saber o que acontece.

Após os campos de data (anos, meses, semanas e dias), podemos ter os campos de horas, que são a hora, o minuto e o segundo, representados respectivamente por `H`, `M` e `S` (também do inglês: *hours*, *minutes* e *seconds*). Repare que o `M` pode ser usado duas vezes, tanto para meses quanto para minutos. Para resolver esta ambiguidade, usa-se a letra `T` maiúscula como separador: antes do `T` estão os campos de data, e depois ficam os campos de horas.

Portanto, `P1M` é uma duração de um mês, enquanto `PT1M` é uma duração de 1 minuto. Os campos que têm valor zero podem ser omitidos, e quando não houver nenhum campo de data, mesmo assim eu devo colocar o `P` no começo, e usar o `T` para indicar que os próximos campos são relacionados a horas. O `T` não serve somente

para resolver a ambiguidade entre meses e minutos, ele também indica que todos os campos depois dele são relacionados a horas. Ou seja, um período de 2 horas deve ser escrito como `PT2H` , já que "horas" não é um campo de data, e deve estar sempre depois do `T` .

Mais alguns exemplos de duração:

- `P3M4DT20H17S` : 3 meses, 4 dias, 20 horas e 17 segundos
- `P2Y5M` : 2 anos e 5 meses
- `PT3M4.73S` : 3 minutos e 4,73 segundos (as frações podem ser separadas por `.` ou `,`)
- `P1.5D` : 1 dia e meio – qualquer campo pode ser fracionado, desde que seja o último. Ou seja, valores como `P1.5DT2H` não são permitidos, já que o último campo é `H` , então o campo `D` não pode ser fracionado neste caso.
- `PT36H` : 36 horas

O último exemplo (`PT36H`) é interessante por um motivo: quando estamos representando datas e horas, cada campo tem seus valores máximos e mínimos. As horas, por exemplo, podem ter valores de 00 a 23, enquanto os meses só podem ser de 01 a 12 etc. Mas para representar durações, estes limites não se aplicam. Por isso eu posso ter durações como `P15MT200H` (15 meses e 200 horas), e tais valores são perfeitamente válidos. Já para representar uma data e hora, eu não posso ter o mês com valor `15` e as horas com valor `200` .

Pelo menos uma das unidades deve estar presente, então valores como `P` ou `PT` não são durações válidas. Para representar uma duração com valor zero, pode-se usar `P0D` (zero dias) ou `PT0S` (zero segundos), por exemplo. Como as unidades que não aparecem têm automaticamente o valor zero, então entende-se que essas durações correspondem a zero. Lembre-se de que ter uma unidade é obrigatório, então eu não poderia escrever `P0` , nem `PT0` e muito menos `P0T` .

Além disso, não há um formato oficial para uma duração com valor zero. As opções anteriores são igualmente válidas, mas eu também poderia usar `PT0H0M0S` , `P0DT0H` , ou qualquer outra variação. O formato escolhido vai depender mais do software que você está usando, pois nem todos suportam todas as variações de formatos, conforme discutido neste link (https://stackoverflow.com/a/29824127/).

Durações com valores negativos

Os valores de uma duração também podem ser negativos. Mas neste caso, devemos prestar muita atenção, porque durações negativas não são especificadas pela ISO 8601 (https://github.com/moment/moment/issues/2408#issuecomment-110180010/).

Porém, várias APIs estendem a ISO 8601, permitindo valores negativos. A classe `java.time.Duration` , por exemplo, entende `PT-1H3M` como "-1 hora e +3 minutos", o que equivale a "-57 minutos". Pode-se interpretar uma duração negativa como algo que "volta no tempo", ou para calcular algo que ocorreu no passado ("Isto aconteceu há 57 minutos atrás"), mas isso depende de cada implementação.

Como cada API implementa de um jeito, é importante ler a documentação para saber como os valores são interpretados e como isso afeta o funcionamento da duração no seu código. Só para citar alguns exemplos, algumas aceitam que a duração inteira seja negativa (`-PT2H3M15S`), já outras permitem que cada campo possa ser negativo ou positivo (`PT-2H3M-15S`), e há as que aceitam ambos os casos. Sempre leia a documentação.

Não confunda duração com data

O que pode nos confundir, conforme já vimos, é que tanto datas quanto durações usam as mesmas palavras: anos, meses, dias, horas, minutos, segundos, frações de segundo etc. São conceitos

diferentes, e é importante saber distingui-los, e não tratar um como se fosse o outro.

Mas embora sejam conceitos diferentes, ambos estão relacionados, e podem ser combinados de diversas maneiras.

Uma data pode ser somada a uma duração, resultando em outra data. Exemplo:

```
2018-05-04 + P2D = 2018-05-06
// 4 de maio de 2018 somada a uma duração de 2 dias resulta em 6 de maio de 2018
```

A diferença entre duas datas é uma duração. Exemplo:

```
2018-05-06 - 2018-05-04 = P2D
// Entre os dias 4 e 6 de maio de 2018, há uma diferença (ou uma duração) de 2 dias
```

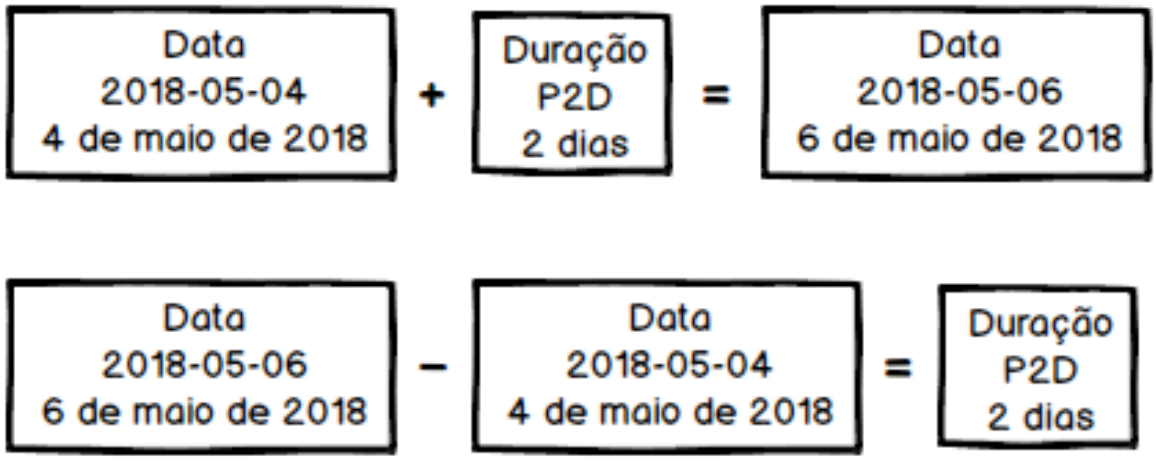

Figura 5.1: Data e duração, dois conceitos diferentes, porém relacionados

Até aqui tudo parece muito simples. Mas lembre-se dos capítulos iniciais, nos quais eu disse que datas são difíceis, mais até do que deveriam. Somar ou subtrair datas e durações parece simples, mas existem muitos casos estranhos e contraintuitivos. Prepare-se para entrar no bizarro mundo da aritmética de datas.

5.2 Somar um dia é o mesmo que somar 24 horas?

Depende.

Na grande maioria dos casos, sim. Mas lembre-se do horário de verão, que vimos no capítulo sobre timezones. Antes de relembrar o que acontece no horário de verão, vamos ver como uma duração é somada a uma data.

A ideia básica é bem parecida com os números. Por exemplo, para somar os números `19` e `23` , vamos seguir o algoritmo que todos aprendemos na escola. **Eu sei** que você já sabe somar, mas a ideia é destrinchar o passo a passo em detalhes (por mais óbvios que sejam), para que possamos usar o mesmo raciocínio quando formos somar datas:

- primeiro eu somo as unidades (no caso, `9` e `3`). O resultado é `12` ;
- Como `12` é maior do que o valor máximo que a unidade suporta (`9`), eu deixo o `2` nas unidades, pego o "excedente" (`1`) e levo-o para a campo seguinte (as dezenas) – o famoso "vai um";
- somando `1` (o excedente), com outro `1` (a dezena do `19`) com `2` (a dezena do `23`), temos o valor `4` no campo das dezenas;
- o resultado final é `42` .

Com datas, é mais ou menos assim. Claro que é um pouco mais complicado, mas a ideia geral é a mesma. Se eu somar um dia, eu aumento o valor do campo correspondente (o dia do mês). Se o resultado exceder o valor máximo permitido (gerou um dia maior do que o mês suporta), eu ajusto o dia e jogo o "excedente" para o campo seguinte (mês).

Usando o exemplo que vimos anteriormente, para somar 2 dias (`P2D`) à data de 4 de maio de 2018 (`2018-05-04`):

- pego o dia do mês (`4`) e somo a quantidade de dias correspondente à duração (`2`) – o resultado é `6` ;
- verifico se este valor ultrapassa o máximo permitido. Como `6` é um dia válido para o mês de maio, não é necessário ajustar;
- o resultado é 6 de maio de 2018 (`2018-05-06`).

Mas e se eu somar 2 dias à data de 30 de maio de 2018 (`2018-05-30`)? Seguindo o mesmo algoritmo:

- pego o dia do mês (`30`), somo a quantidade de dias correspondente à duração (`2`) - o resultado é `32` ;
- porém, `32` ultrapassa o valor máximo permitido, pois maio tem 31 dias. Então eu faço um ajuste, da seguinte forma: `32` ultrapassa o valor máximo em `1` (`32 - 31 = 1`), então o dia deve ser `1` ;
- como eu ultrapassei o último dia de maio em 1 dia, o mês deve ser ajustado para o próximo valor (de `05` para `06`);
- o resultado é `2018-06-01` (1 de junho de 2018).

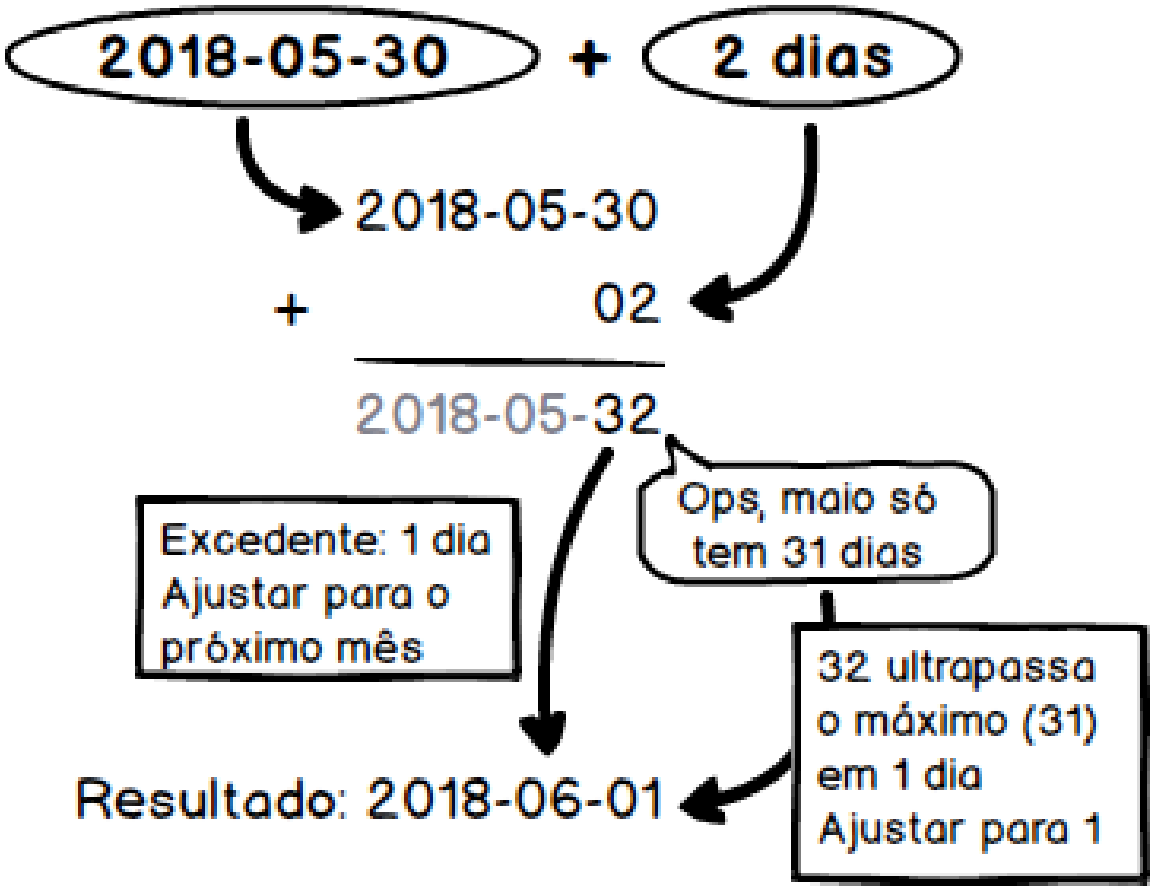

Figura 5.2: Somar dias pode resultar em uma data no mês seguinte

Quando eu somo horas, é a mesma coisa: eu somo a quantidade no campo horas, verifico se ultrapassou o valor máximo e faço os ajustes nos demais campos (dia, mês etc.) conforme necessário.

Por exemplo, se eu somar 24 horas à data/hora de `2018-05-04T17:00` (4 de maio de 2018, às 17:00):

- pego as horas (`17`) e somo `24` , o resultado é `41` ;
- `41` ultrapassa o valor máximo para horas, então faço o ajuste: o dia só pode ter 24 horas, então subtraio `24` de `41` e o resultado é `17` , que passa a ser o valor das horas. O excedente (`41 - 17 = 24`) é passado para o próximo campo (dias);
- o excedente de `24` horas equivale a `1` dia, e este valor é somado ao respectivo campo (`04` se torna `05`). Como `05` é um valor

válido para o dia, nenhum outro ajuste é necessário na data;
- o resultado é `2018-05-05T17:00` (5 de maio de 2018 às 17:00 – o mesmo horário do dia seguinte).

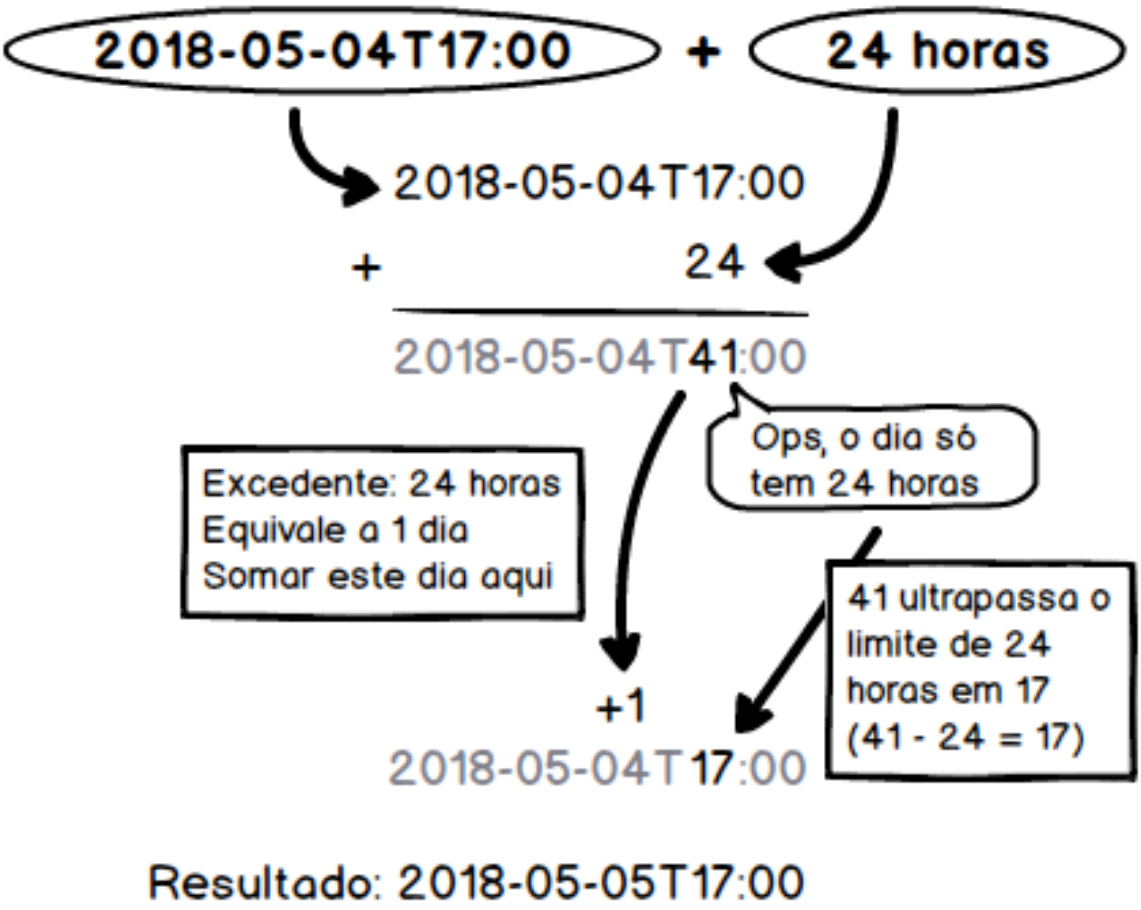

Figura 5.3: Somar 24 horas é o mesmo que somar 1 dia, na maioria das vezes

Note que, neste caso, se eu somar 1 dia a `2018-05-04T17:00` , o resultado também será `2018-05-05T17:00` . Na grande maioria dos casos, somar 24 horas é o mesmo que somar 1 dia. Mas se levarmos em conta os timezones...

Timezones e os dias que não têm 24 horas

Lembra do capítulo sobre timezones, no qual falei sobre o horário de verão e outros casos em que ocorrem *gaps* e *overlaps*, e por isso

nem todos os dias tem 24 horas? Se você não se lembra ou não entendeu muito bem, sugiro revisá-lo antes de prosseguir.

Vamos usar como exemplo o horário de verão em São Paulo, que em 2017 começou no dia 15 de outubro. Vamos usar um horário do dia anterior: `2017-10-14T10:00-03:00` (14 de outubro de 2017 às 10:00, um dia antes de começar o horário de verão). Lembre-se de que o offset usado em São Paulo neste dia é `-03:00` .

Se eu somar um dia, o respectivo campo (o dia do mês, com valor `14`) vai ser incrementado para `15` , e nenhum outro ajuste será necessário no mês, ano, horas, minutos e segundos. Então o resultado vai ser o dia seguinte (`2017-10-15`) e com o mesmo horário (`10:00`). Porém, lembre-se de que durante o horário de verão, o offset muda para `-02:00` (se você não lembra por que o offset muda, releia o capítulo sobre timezones). Ou seja, somando um dia, o resultado final é `2017-10-15T10:00-02:00` .

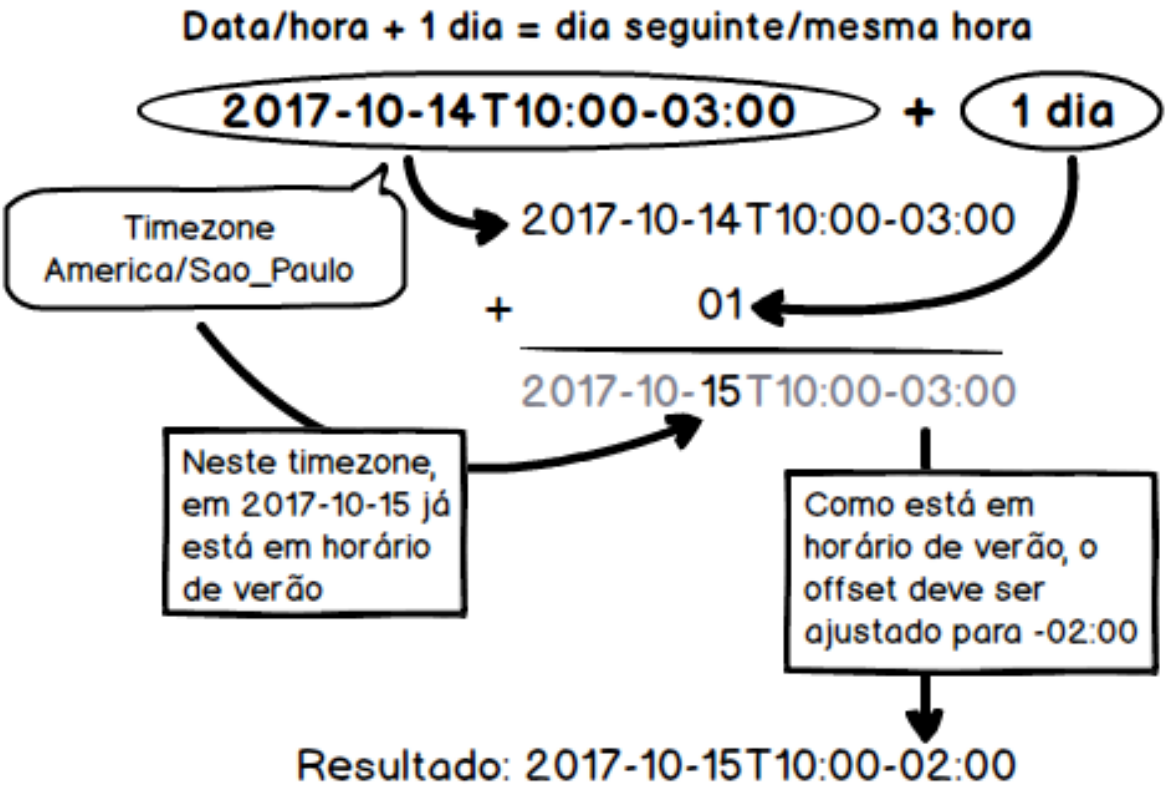

Figura 5.4: Somar 1 dia resulta no dia seguinte, no mesmo horário. O ajuste de offset é necessário devido ao horário de verão

Mas se eu somar 24 horas, o resultado é diferente. Para somar 24 horas a `2017-10-14T10:00-03:00`, devo adicionar `24` ao campo de horas (`10`) e o resultado é `34`. Porém, o dia só pode ter 24 horas, então eu desconto essas horas a mais de `34`, resultando em `10` (`34 - 24 = 10`), que deve ser o valor final para a hora. O excedente de 24 horas é convertido em um dia e somado ao campo dia (`14`), resultando em `15`. Por isso, o resultado seria `2017-10-15T10:00-03:00`.

Mas temos que lembrar que neste instante o horário de verão já está sendo usado no timezone `America/Sao_Paulo`, então o offset deve ser ajustado para `-02:00`. Então, convertendo `2017-10-15T10:00-03:00` para o offset `-02:00`, temos o resultado final, que é `2017-10-15T11:00-02:00` (dia 15 de outubro de 2017, às **11:00**).

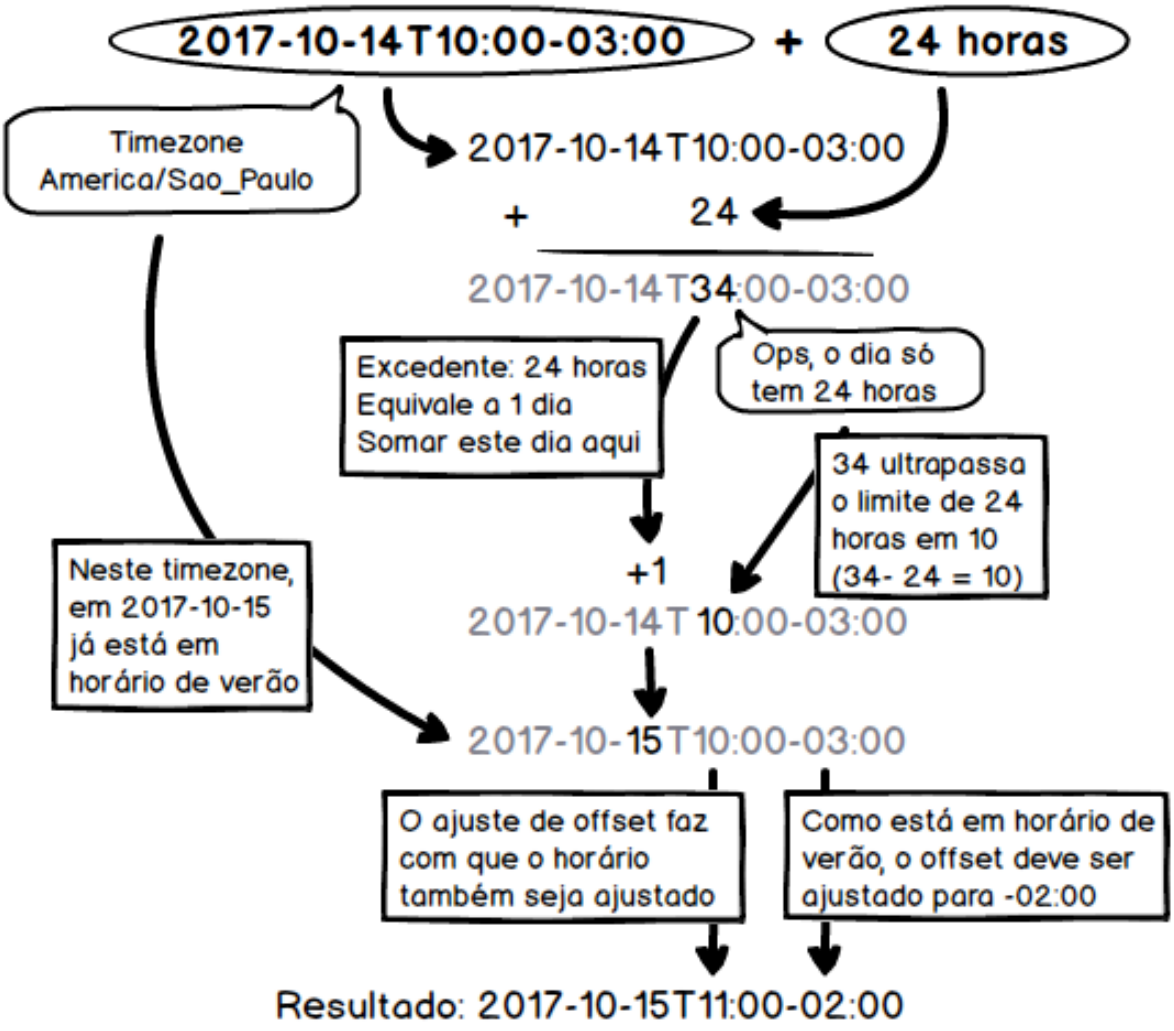

Figura 5.5: Somar 24 horas nem sempre é o mesmo que somar 1 dia

A tabela a seguir resume o que acabamos de ver:

Data	São Paulo	UTC
Data inicial	2017-10-14T10:00-03:00	2017-10-14T13:00Z
Somar 1 dia	2017-10-15T10:00-02:00	2017-10-15T12:00Z
Somar 24 horas	2017-10-15T11:00-02:00	2017-10-15T13:00Z

Esta diferença acontece porque, quando o horário de verão começou, o relógio foi adiantado uma hora em São Paulo, e o offset mudou de `-03:00` para `-02:00` . Como uma hora é "pulada", isso acaba causando estes resultados estranhos.

Na tabela anterior, veja que ao somar 1 dia o resultado é um instante cuja diferença para a data inicial é de 23 horas (compare os valores em UTC) – em outras palavras, este é um caso em que 1 dia é equivalente a 23 horas.

O resultado pode parecer contraintuitivo, pois desde sempre temos esta noção de que o dia tem 24 horas e que esta é uma regra imutável. Mas temos que levar em conta a semântica da operação "somar 1 dia": estou aumentando o valor do campo "dia", do ponto de vista daquele timezone. Ao somar 1 dia, o resultado deve ser o dia seguinte (15 de outubro de 2017), no mesmo horário local (10:00) observado naquele timezone. Como entre um dia e outro ocorreu a mudança do horário de verão e o offset mudou, ocorre esta distorção.

Já para somar 24 horas, imagine que um cronômetro, em São Paulo, seja iniciado no dia 14 de outubro de 2017 às 10:00. Quando a data e horário local for 15 de outubro de 2017 às 11:00, o cronômetro estará marcando 24 horas. Isso porque à meia-noite o relógio foi adiantado em uma hora, mas o cronômetro não teve uma hora adicionada à sua contagem – ele continuou contando o tempo decorrido, sem saltos. Por isso, somar 24 horas resulta em um horário diferente do dia seguinte.

Quando acaba o horário de verão, também acontece algo similar. Ainda no timezone `America/Sao_Paulo` , vamos usar a data `2018-02-17T10:00-02:00` (17 de fevereiro de 2018, às 10:00). Lembrando que neste dia São Paulo ainda estava em horário de verão, por isso o offset é `-02:00` .

Ao somar 1 dia, o resultado deverá ser o dia seguinte (`2018-02-18`), no mesmo horário local (`10:00`). Porém, neste timezone, o horário

de verão já terminou (no dia 18 à meia-noite os relógios foram atrasados em uma hora), e o offset usado já é `-03:00` . Por isso, o resultado é `2018-02-18T10:00-03:00` .

Mas se somarmos 24 horas, o resultado é diferente. A data `2018-02-17T10:00-02:00` mais 24 horas resulta em `2018-02-18T10:00-02:00` . Porém, como o offset usado no timezone `America/Sao_Paulo` neste instante é `-03:00` , devemos converter o resultado para este offset. Assim, o resultado final é `2018-02-18T09:00-03:00` (18 de fevereiro de 2018, às **09:00**).

Usando a mesma analogia do cronômetro: imagine que ele foi iniciado em São Paulo, no dia 17 de fevereiro às 10:00. Quando chega meia-noite, o horário de verão acaba e os relógios são atrasados em uma hora, de volta para as 23:00. Ou seja, todos os minutos entre 23:00 e 23:59 acontecem duas vezes. Só que o cronômetro não tem uma hora retirada da sua contagem. Na verdade, ele conta todo o período das 23:00 às 23:59 duas vezes. Por isso, quando são 09:00 do dia 18, ele já estará marcando 24 horas.

A tabela a seguir mostra os valores obtidos em cada caso:

Data	São Paulo	UTC
Data inicial	2018-02-17T10:00-02:00	2018-02-17T12:00Z
Somar 1 dia	2018-02-18T10:00-03:00	2018-02-18T13:00Z
Somar 24 horas	2018-02-18T09:00-03:00	2018-02-18T12:00Z

Repare que ao somar 1 dia, o resultado é um instante cuja diferença para a data inicial é de 25 horas (compare os valores em UTC). Ou seja, neste caso, 1 dia é equivalente a 25 horas.

Situações como essa fazem com que uma duração de 1 dia (`P1D`) nem sempre seja equivalente a uma duração de 24 horas (`PT24H`). Provavelmente é por isso que a norma ISO 8601 permite que os valores dos campos de uma duração não tenham limites (como é feito para as datas e horas).

Se a maior duração permitida em horas fosse 24 (`PT24H`), o único jeito de representar 36 horas seria `P1DT12H` (1 dia e 12 horas). Mas como a duração de 1 dia nem sempre é igual a 24 horas, então `P1DT12H` nem sempre será igual a 36 horas. O único jeito de representarmos esta quantidade de horas, sem sofrer interferência dos *gaps* e *overlaps*, é permitindo que se use o valor 36 no campo "horas" (`PT36H`).

Lembre-se de que não é só o horário de verão que faz isso

Relembrando novamente o capítulo sobre timezones, vimos que *gaps* e *overlaps* não ocorrem somente por causa do horário de verão, e nem sempre são intervalos de 1 hora (como o caso da Coreia do Norte, que pulou meia hora, ou ainda o caso de Samoa, que pulou um dia inteiro). Sempre que tiver um timezone envolvido e você que tiver que somar uma duração a uma data, ou calcular a diferença entre duas datas, algumas destas situações podem acontecer.

Para casos assim, você deve decidir, por exemplo, se quer que o resultado seja o mesmo horário no dia seguinte, ou 24 horas depois, mesmo que o horário local seja diferente. Lembre-se do caso em que há um *gap* e alguns minutos podem não existir naquele dia, então não haverá como mostrar "o mesmo horário" nesses casos (se você não lembra desta situação, releia o capítulo de timezones).

Repare também como estas contas são complicadas, pois temos que considerar todas as regras do timezone em questão, se houve mudança de offset etc. Somar datas nunca é algo simples. Você pode até fazer um código simples que "funciona" na maioria dos

casos, mas usar uma API dedicada é sempre melhor do que tentar fazer estas contas manualmente.

Mas lembre-se de que esta é apenas uma forma de se abordar o problema de somar dias e horas a uma data. Cada linguagem ou API pode implementar estas operações de maneira diferente (por exemplo, considerar 1 dia sempre igual a 24 horas), então sempre leia a documentação para não ser pego de surpresa.

> Como veremos em capítulos posteriores, a API `java.time` usa a abordagem que explicamos anteriormente (pode trazer resultados diferentes quando somamos 1 dia ou 24 horas).

5.3 Um mês tem quantos dias?

Depende.

Um mês pode ter 28, 29, 30 ou 31 dias. Por isso, somar uma duração de 1 mês pode resultar em uma data que está 28, 29, 30 ou 31 dias depois.

Exemplo: se eu somar 1 mês à data de `2018-01-01` (1 de janeiro de 2018), o resultado é `2018-02-01` (1 de fevereiro de 2018). E se eu calcular a diferença entre estas datas, em dias, terei como resultado `31` . Agora, se eu somar 1 mês à data de `2018-02-01` , o resultado é `2018-03-01` (1 de março de 2018). Porém, a diferença entre 1 de fevereiro e 1 de março, em dias, é de `28` (isso porque o ano é 2018, pois se fosse em um ano bissexto, a diferença seria de 29 dias).

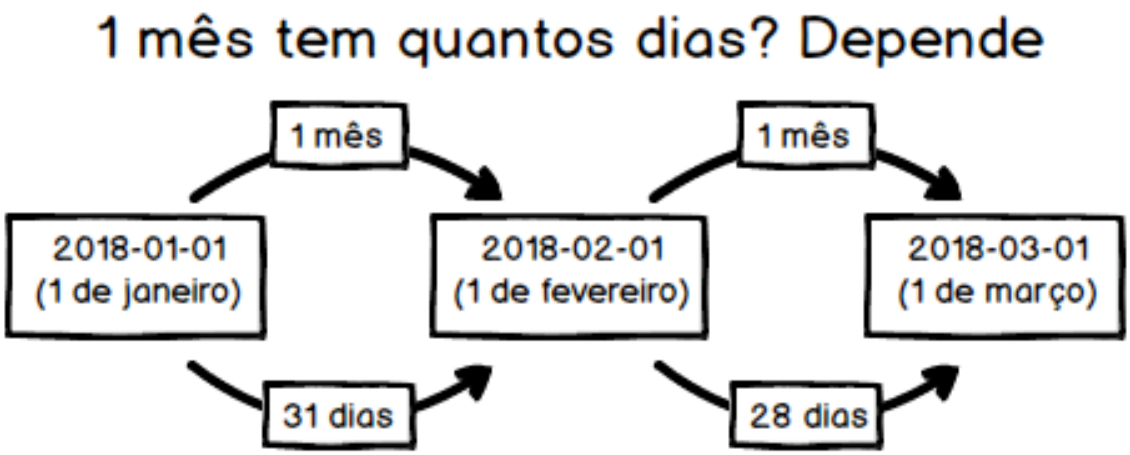

Figura 5.6: Cada mês tem uma quantidade diferente de dias

Somar 1 mês é igual a somar uma quantidade de dias que depende das datas envolvidas na operação. Claro que no dia a dia, em alguns contextos, existe a convenção de que um mês é igual a 30 dias, mas essa regra não se aplica na aritmética de datas.

Novamente, temos que pensar na semântica da operação. "Somar 1 mês" significa que o campo "mês" vai ter seu valor aumentado em 1, e os demais campos devem ser ajustados se necessário: se eu somar 1 mês à data de `2018-12-01` (1 de dezembro de 2018), por exemplo, terei que ajustar o ano para 2019, pois o resultado será `2019-01-01` (1 de janeiro de 2019). É a mesma regra já vista anteriormente para dias e horas (somar um valor ao campo em questão, verificar se ultrapassa os valores permitidos, ajustar os demais campos caso necessário).

E se a data for `2018-01-31` (31 de janeiro de 2018), o que acontece ao somar 1 mês? Relembrando nosso algoritmo:

- somar `1` no campo mês (`01`), o resultado é `02` (fevereiro);
- mas o dia é `31` , e fevereiro não tem 31 dias;
- então devemos ajustar alguns campos, mas quais?

Qual o ajuste que deve ser feito? Poderíamos fazer algo similar ao que foi feito ao somar dias: o dia 31 de fevereiro não existe, mas 31

são 3 dias depois do último dia de fevereiro (28), então a data seria ajustada para 3 de março.

Mas pensando novamente na semântica da operação "somar 1 mês": se estou somando 1 mês a uma data em janeiro, porque o resultado deveria ser em março (2 meses depois)? Eu deveria ter uma data em fevereiro, seria o mais "lógico". Então o correto seria manter o mês `02` e ajustar o dia. E já que a data inicial corresponde ao último dia de janeiro, somar 1 mês deveria resultar no último dia de fevereiro. Por isso, a data é ajustada para `2018-02-28` (e se fosse em um ano bissexto, o dia seria ajustado para `29`).

Então esta é a resposta correta? Depende. Esta é uma das formas de se abordar o problema de adicionar meses a uma data. Até mesmo o meu comentário sobre o fato de o resultado ser em março (2 meses depois) pode ser questionado, pois não existe uma definição formal da operação "somar meses a uma data". Pode ser que a API que você usa implemente uma regra diferente, e por isso você deve sempre ler a documentação para saber como o cálculo está sendo feito.

O problema de somar unidades que não têm tamanho fixo

Estes problemas que surgem na aritmética de datas acontecem porque nem todas as unidades têm sempre o mesmo tamanho.

Com os números isso não acontece, porque 10 unidades formam uma dezena, 10 dezenas formam uma centena e assim por diante, e isso nunca muda. Não existe uma centena que pode ter 10 ou 11 dezenas, dependendo dos números envolvidos. O tamanho é sempre fixo.

Já com datas, as coisas não funcionam assim, pois 1 mês pode ter 28, 29, 30 ou 31 dias e 1 ano pode ter 365 ou 366 dias. Quando somamos meses ou anos, a quantidade total de dias adicionados varia de acordo com a data inicial.

Por isso, ao calcular a diferença entre duas datas em termos de meses e anos, deve-se tomar alguns cuidados. Alguns algoritmos simplesmente calculam a diferença em segundos, por exemplo, e depois convertem para horas (dividindo por 3600), e depois para dias (dividindo por 24), e finalmente para anos, dividindo por 365. Por fim, o resto desta última divisão, dividido por 30, dá a quantidade de meses.

Nada errado com este algoritmo, contanto que você saiba que ele retorna apenas valores aproximados de anos e meses, pois está assumindo que todos os anos têm 365 dias (ignorando anos bissextos) e todos os meses têm 30 dias. Este algoritmo também não leva em conta as mudanças que ocorrem por causa dos timezones, e que já vimos que podem causar diferenças sutis, dependendo das mudanças de offset envolvidas. Cabe a você avaliar se isso é uma aproximação aceitável para o seu caso de uso.

Um exemplo bem comum com anos é o cálculo da idade, baseado na data de nascimento. Primeiro, o caso mais fácil: se a data de nascimento é `2000-01-01` (1 de janeiro de 2000), em `2006-12-31` (31 de dezembro de 2006) a idade será 6 anos, pois ainda falta um dia para o sétimo aniversário – mesmo que falte um dia, e tecnicamente a idade seja `6,9999...` anos, o valor é arredondado para baixo (supondo que você só queira um valor inteiro para a idade). Somente em `2007-01-01` (1 de janeiro de 2007) a idade será 7 anos.

Mas e se a data de nascimento for `2000-02-29` (29 de fevereiro de 2000)? Em `2007-02-28` (28 de fevereiro de 2007), a idade é 6 ou 7 anos? Algumas APIs podem considerar que, como o dia 29 não chegou, então ainda não completou aquele ano, e por isso a idade é 6 anos (somente a partir de `2007-03-01` a idade seria 7 anos). Outras podem implementar de maneira diferente: a data de nascimento é no último dia de fevereiro, e como 2007 não é ano bissexto, `2007-02-28` é o último dia do mês, e por isso o sétimo ano já foi completado.

Qual abordagem é a correta? Ambas. Também não há uma regra oficial para definir esta diferença, e cada API pode implementar de um jeito diferente. Sempre leia a documentação (e teste seu código).

Como meses e anos têm tamanhos variados, outras medidas de tempo baseadas neles também não terão tamanho definido. Um bimestre, trimestre ou semestre, por exemplo, têm quantos dias? Depende dos meses envolvidos. Se eu considerar o semestre de 1 de janeiro a 30 de junho, ele tem 181 dias (ou 182 em anos bissextos), já o semestre de 1 de julho a 31 de dezembro tem 184 dias. Mas se o período de 6 meses for de 1 de abril a 30 de setembro, então serão 183 dias.

O mesmo vale para os períodos baseados em anos. Uma década tem quantos dias? Não podemos simplesmente multiplicar 365 por 10, pois temos que levar em conta a quantidade de anos bissextos contidos na década em questão.

Por exemplo, entre `2000-01-01` e `2010-01-01` , há 3653 dias. Isso porque este período passa por 3 anos bissextos (2000, 2004 e 2008). Já entre `1897-01-01` e `1907-01-01` há 3651 dias, pois neste período somente o ano 1904 é bissexto – 1900 não é, porque anos que são divisíveis por 100, só são bissextos se também forem divisíveis por 400 (https://pt.wikipedia.org/wiki/Ano_bissexto#Calend%C3%A1rio_Gregoriano).

Se você usar uma abordagem simplista (como subtrair os anos e multiplicar por 365), não vai obter o resultado exato. Quando o assunto é datas, não reinvente a roda, use uma API dedicada, que faz todos os cálculos complicados para você.

A aritmética de datas é complicada por causa das regras que regem nosso calendário. Isso gera muitos resultados inesperados e contraintuitivos. Estar ciente de todos estes detalhes ajuda a

entender melhor quando a diferença entre duas datas não der o resultado que você espera.

Capítulo 6
E vamos ao código?

Chegamos ao final da primeira parte do livro. Até aqui, vimos os principais conceitos envolvendo datas e horas, do ponto de vista da programação. Sabendo lidar com UTC, timezones, durações e aritmética de datas, você já conseguirá atacar a maioria dos problemas do dia a dia.

Nos próximos capítulos veremos vários exemplos em Java, explicando em detalhes suas APIs e mostrando como elas implementam os conceitos vistos até então.

Algumas coisas que vimos nesta primeira parte serão aprofundadas, seja em mais detalhes, ou em casos de uso diferentes, com situações não cobertas anteriormente. Por exemplo, veremos o código para saber quais os offsets usados por um timezone, para fazer *parsing* e formatação de datas, entre outros.

Mesmo que você não se interesse pelo código (ou por Java), a explicação destes conceitos pode ajudá-lo, pois pode ser que eles também existam na sua linguagem favorita. Por exemplo, a ideia geral de formatação e *parsing* (e os erros mais comuns ao tentar aplicá-la) costuma ser bem parecida em todas as linguagens. Mesmo que o código Java em si não lhe interesse, conhecer o mecanismo correto pode ajudá-lo ao usar APIs de outras linguagens.

APIs legadas do Java

Nos próximos capítulos, veremos como o Java implementa os conceitos de data e hora que acabamos de conhecer.

Esta segunda parte do livro é dedicada à API legada (`java.util.Date` , `java.util.Calendar` e demais classes relacionadas). Vamos ver seu funcionamento básico, além de apontar seus problemas e limitações, que levaram à criação de outra API (o pacote `java.time` , que foi introduzido no Java 8 e será explicado em detalhes na terceira parte).

A partir de agora, passarei a chamar estas classes de `Date` e `Calendar` . Só usarei o nome completo quando necessário — por exemplo, para diferenciar `java.util.Date` de `java.sql.Date` , ou em qualquer outra situação em que o nome possa ficar ambíguo ou confuso.

E se você achou estranho que eu esteja chamando `Date` e `Calendar` de "API legada", saiba que esta definição é da Oracle e não minha: em seu tutorial *Date Time*, é assim que estas classes são chamadas (https://docs.oracle.com/javase/tutorial/datetime/iso/legacy.html/) e em vários pontos do livro também usarei esta nomenclatura.

Alguns podem até questionar a necessidade de ainda precisarmos saber usar `Date` e `Calendar` , pois com o surgimento do `java.time` , ninguém mais deveria usar a API legada. Concordo em partes. Por um lado, qualquer código novo, de fato, deveria usar as APIs mais novas, sempre que possível. O `java.time` é muito superior, por ter uma maior facilidade de uso, corretude, mais funcionalidades, e várias outras vantagens que veremos na terceira parte do livro. Por outro lado, ainda há muito código legado usando `Date` e `Calendar` . Muitas bibliotecas ainda não possuem suporte às classes do `java.time` , ou só suportam nas suas versões mais recentes (e sabemos que nem sempre é fácil mudar a versão de nossas dependências), e há muitos sistemas por aí usando versões anteriores ao JDK 8.

> É possível ter boa parte da funcionalidade do `java.time` no JDK 6 e 7, através do ThreeTen Backport (http://www.threeten.org/threetenbp/).

Por isso, nem sempre é possível ter um código que só use a nova API. Em muitos casos, você terá que usar uma abordagem mista, e ir migrando o código aos poucos. Em alguns pontos, você poderia usar o `java.time` para realizar as lógicas de negócio e demais operações com datas (já que ela é melhor para isso) e converter de/para `Date` ou `Calendar` quando for usar o código ou as bibliotecas que ainda trabalham com a API legada.

Porém, como o `java.time` possui conceitos e implementações diferentes de `Date` e `Calendar` , nem sempre é trivial migrar o código de um para o outro. Só com um bom entendimento de como cada classe funciona podemos saber exatamente como migrar nosso código de forma a termos os mesmos resultados. Por isso, os próximos capítulos são dedicados à API legada.

Configurações e outros detalhes

Vários exemplos usam código que retorna a data e hora atual. Como esta é uma informação que muda o tempo todo, cada vez que você rodasse estes códigos, obteria um resultado diferente.

Para evitar este problema, em todos estes exemplos a data e hora atual corresponde a `2018-05-04T17:00-03:00` (4 de maio de 2018, às 17:00 em São Paulo — horário oficial de Brasília). É um valor arbitrário, sem nenhum significado especial, escolhido apenas para que os exemplos tenham um mesmo valor e não fiquem confusos. Quando o código retornar um valor diferente, isso será explicado no respectivo exemplo.

Além disso, na JVM usada para rodar estes exemplos, o timezone padrão é `America/Sao_Paulo` e o locale padrão é `pt_BR` . Ao longo dos capítulos, será explicado o que estas configurações significam e

como elas afetam o funcionamento do código. Para ter as mesmas configurações, você pode usar a classe `exemplos.setup.Setup` (disponível no GitHub: https://github.com/hkotsubo/java-datetime-book/blob/master/src/main/java/exemplos/setup/Setup.java/), ou configurar sua JVM com estes parâmetros:

```
-Duser.country=BR -Duser.language=pt -Duser.timezone=America/Sao_Paulo
```

E vamos ao código!

Capítulo 7
A primeira API de data do Java

Desde a versão `1.0` do Java, existe a classe `java.util.Date` , feita para trabalhar com datas. No JDK `1.1` , foi criada a classe `java.util.Calendar` , para complementar e/ou corrigir algumas funcionalidades de `Date` . Estas são duas das classes mais problemáticas e mal compreendidas da linguagem, e ao longo deste e dos próximos capítulos entenderemos os motivos.

O objetivo deste capítulo não é simplesmente criticar estas classes. A internet já faz um bom trabalho quanto a isso, em artigos como este (https://hackernoon.com/going-on-a-date-with-java-9bdac2c950b3/), e é claro, no Stack Overflow, nesta resposta (https://stackoverflow.com/a/1969651/) e também nesta outra (https://stackoverflow.com/a/1571329/) – e em muitas outras, basta procurar.

Na verdade, a ideia principal é entender como estas classes funcionam, para que você chegue às suas próprias conclusões quanto à qualidade, funcionalidade e facilidade de uso. E para que também possa comparar estas caraterísticas com a API `java.time` , na terceira parte do livro. Mas não vou deixar de dar minha opinião em alguns pontos.

Para começar, o que é a classe `Date` ? Apesar do nome, ela **não é uma data**.

"Como assim?" – você pensa.

"Mas eu uso `Date` para trabalhar com datas!!"

Calma.

7.1 java.util.Date não é uma data

A classe `Date` não é uma data, simplesmente porque ela não representa exatamente um único dia, mês e ano específicos. Na verdade, ela representa um ponto na linha do tempo (um Unix timestamp, aquele número gigante dos capítulos anteriores). E como já sabemos, o timestamp pode representar uma data e hora diferente, dependendo do timezone utilizado. Por exemplo, se eu criar um objeto `Date` contendo a data atual:

```
// Date contendo o instante atual
Date agora = new Date();
```

Internamente, `new Date()` chama o método `System.currentTimeMillis()` , que por sua vez retorna um `long` contendo o número de milissegundos desde o Unix Epoch (ou seja, o valor do timestamp). Este número é a única informação que o `Date` possui. Qualquer outro valor (dia, mês, ano, hora, minuto, segundo) é calculado de acordo com o timezone padrão que está configurado na JVM (mais adiante explicarei o que é esta configuração).

Se você imprimir a data usando `System.out.println(agora)` , a saída será:

```
Fri May 04 17:00:00 BRT 2018
```

Esta é a saída obtida quando a data e hora atual é 4 de maio de 2018, às 17:00 em São Paulo, e o timezone padrão da JVM é `America/Sao_Paulo` .

Internamente, `println()` chama o método `toString()` do objeto passado. E o método `toString()` da classe `Date` retorna a data nesse formato. Repare que o dia da semana e o mês estão em inglês (`Fri` `May`), independente da linguagem configurada na JVM. Note também que foi usada a abreviação do timezone ("BRT", que significa *Brasília Time*, e é a abreviação usada pelo timezone `America/Sao_Paulo` quando não está em horário de verão).

Lembra do capítulo no qual falamos que datas não têm formato? Pois bem, a classe `Date` não tem um formato. Ela só tem um valor: o timestamp (o número gigante), que no caso é guardado em um campo do tipo `long` . Quando o método `toString()` é chamado, ele usa o timezone padrão da JVM (que no meu caso é `America/Sao_Paulo`) para calcular os valores da data e hora, e em seguida os imprime em um formato específico (`Fri May 04 17:00:00 BRT 2018`). Mas nem o timezone, nem o formato, fazem parte da classe `Date` .

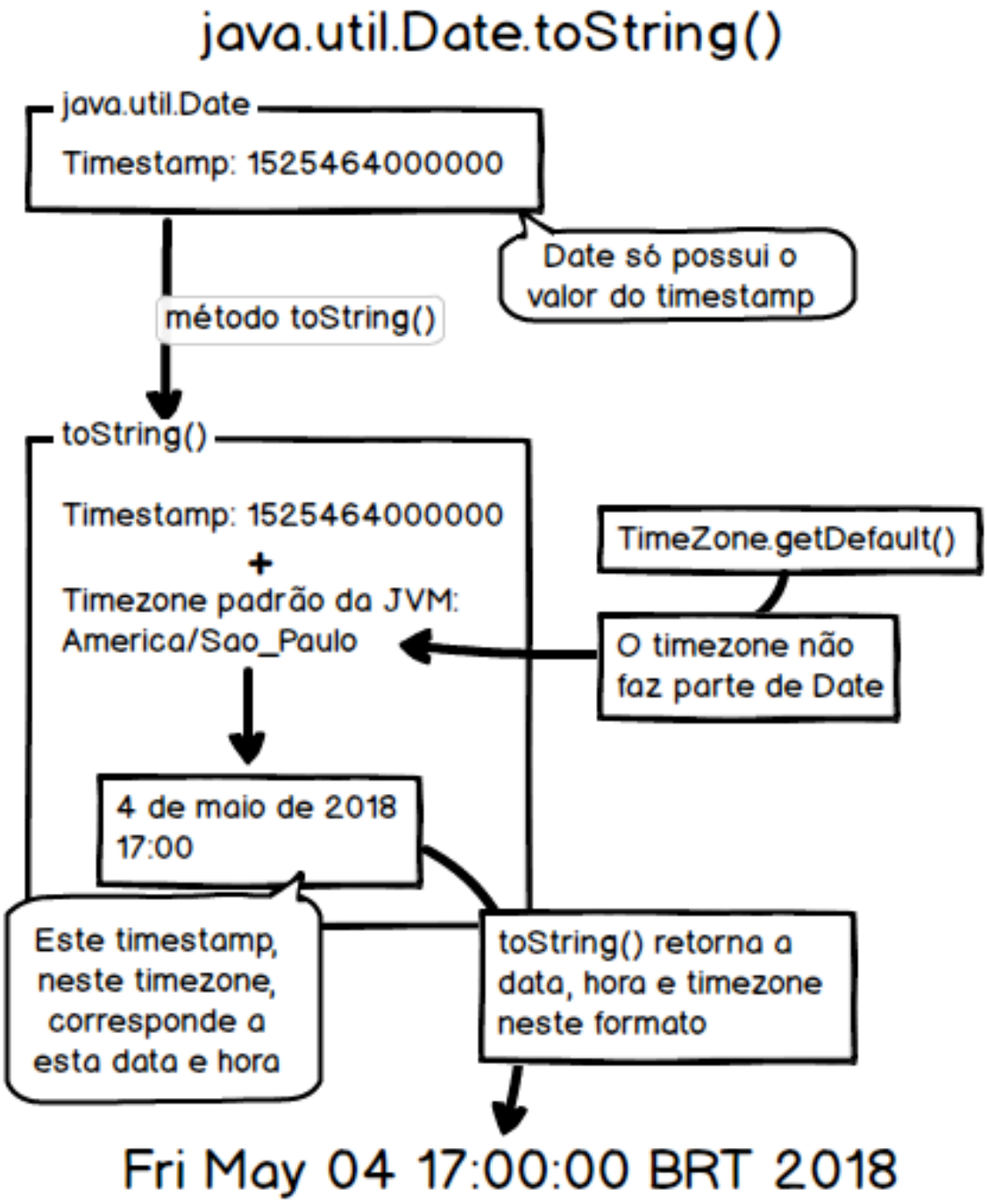

Figura 7.1: Date.toString() usa o timezone padrão da JVM para calcular os valores de data e hora

O retorno depende do timezone configurado na JVM

Para entender melhor como isso funciona, vamos mudar o timezone padrão, usando o método `setDefault()` da classe `java.util.TimeZone` , e ver como a saída muda de acordo com esta configuração. A classe `TimeZone` representa – adivinhe – um timezone, e possui o método estático `getTimeZone()` , que recebe como parâmetro uma `String` com um identificador da IANA, como `America/Sao_Paulo` ou `Europe/Berlin` .

E para mostrar que o valor do `Date` continua o mesmo, independente do timezone padrão, vamos usar o método `getTime()` , que retorna o timestamp. Eu também uso `TimeZone.getDefault()` , que retorna o timezone padrão que está setado no momento, e uso o método `getID()` , que retorna o seu nome (o identificador da IANA):

```
TimeZone.setDefault(TimeZone.getTimeZone("Europe/Berlin"));
System.out.println(agora.getTime() + "=" + agora + " - " +
TimeZone.getDefault().getID());
TimeZone.setDefault(TimeZone.getTimeZone("Asia/Tokyo"));
System.out.println(agora.getTime() + "=" + agora + " - " +
TimeZone.getDefault().getID());
TimeZone.setDefault(TimeZone.getTimeZone("UTC"));
System.out.println(agora.getTime() + "=" + agora + " - " +
TimeZone.getDefault().getID());
```

Primeiro, mudamos o timezone padrão para `Europe/Berlin` (usado na Alemanha), depois `Asia/Tokyo` (Japão), e depois para `UTC` . Com isso, a saída deste código é:

```
1525464000000=Fri May 04 22:00:00 CEST 2018 - Europe/Berlin
1525464000000=Sat May 05 05:00:00 JST 2018 - Asia/Tokyo
1525464000000=Fri May 04 20:00:00 UTC 2018 - UTC
```

Na primeira linha, temos o timestamp convertido para o timezone `Europe/Berlin` , e note que horário mudou para 22:00 e a abreviação do timezone é CEST (*Central European Summer Time*, o horário de verão de boa parte da Europa Central).

Já na segunda linha, temos a data e hora correspondente no timezone `Asia/Tokyo` , e veja que no Japão já é dia 5 de maio, às

05:00, e a abreviação usada é JST (*Japan Standard Time*). E por fim, na terceira linha, temos a data e hora em UTC.

Mas veja que nas 3 linhas o timestamp é o mesmo. O valor do objeto `Date` não muda, mas o método `toString()` converte este valor para o timezone padrão que estiver setado no momento em que ele é chamado, resultando em uma data e hora diferente.

Outro ponto de atenção é que o formato usado por `toString()` não mostra o offset, apenas a abreviação do timezone. E como já vimos no capítulo sobre timezones, usar abreviações não é a melhor opção, já que muitas são ambíguas e/ou abrangentes demais, não sendo possível mapeá-las para um único timezone.

> Lembre-se de que UTC é um padrão, e não um timezone de fato. Ele é a base a partir da qual todos os offsets são definidos, não é afetado pelo horário de verão, nem possui qualquer tipo de *gap* ou *overlap*. Apesar disso, o método `TimeZone.getTimeZone()` aceita `"UTC"` como parâmetro, e o resultado é um "timezone" que possui offset zero, sem nenhuma mudança (como horário de verão, por exemplo). Ou seja, no fim, ele se comporta como se fosse o próprio UTC.
>
> Esta é uma prática comum (tratar UTC como se fosse um timezone), pois simplifica o modelo de classes — caso contrário, seria necessário ter uma classe específica para representar UTC, por exemplo. Há uma discussão interessante sobre isso nesta pergunta do Stack Overflow (https://stackoverflow.com/q/44756430/).

Este é um dos problemas desta API: o retorno do método `toString()` nos engana. Ao mostrar a data dessa maneira (`Fri May 04 17:00:00 BRT 2018`), a API passa a impressão de que o objeto `Date` possui exatamente aquela data e hora, naquele timezone. Por isso, é muito comum perguntar *"Como eu converto um `Date` para outro timezone?"*. E a resposta é: você não converte. Porque um `Date` não

possui um timezone. O que você pode fazer é transformar o valor do timestamp em uma `String` que representa a data e hora em um timezone específico, em um determinado formato. Mas o `Date` em si não terá seu valor alterado.

Cuidados ao mudar o timezone padrão

O timezone padrão pode ser mudado a qualquer momento, por qualquer ponto do seu código. Quando você usa `TimeZone.setDefault()` , todas as aplicações que rodam na mesma JVM serão afetadas e o valor retornado por `TimeZone.getDefault()` mudará para todas elas.

O timezone padrão também pode mudar caso a configuração do sistema operacional seja alterada e a JVM seja reiniciada, e também pode ser alterado via configuração, através da propriedade `user.timezone` , conforme explicado neste link (https://stackoverflow.com/a/45797132/).

De qualquer forma, esta é uma configuração sobre a qual você não tem muito controle, então tenha muito cuidado ao usar `TimeZone.setDefault()` , pois você pode estar afetando outras partes do seu sistema, ou até mesmo outras aplicações, sem nem saber.

Na verdade, não é um "pecado mortal" usar ou alterar o timezone padrão, desde que você esteja ciente de que ele está sendo usado e saiba quais os efeitos que isso pode causar. Minha crítica é mais para o fato de a saída do método `toString()` depender desta configuração e nem sequer documentar este fato — não há qualquer menção ao timezone padrão (*JVM default timezone*) na documentação do método (https://docs.oracle.com/javase/8/docs/api/java/util/Date.html#toString--/).

`Date` possui métodos para obter campos específicos, como `getMonth()` para obter o valor numérico do mês, ou `getHours()` para obter as horas. Todos estes métodos também usam o timezone padrão para calcular o valor do respectivo campo. Porém, todos esses *getters*

estão *deprecated* desde o JDK `1.1` , sendo recomendado o uso de `Calendar.get()` , que explicarei posteriormente.

Além disso, alguns nomes de métodos não foram muito bem pensados, como `getDate()` , que retorna o dia do mês, e `getDay()` , que retorna o dia da semana, e o já citado `getTime()` , que retorna o valor do timestamp. Uma opção melhor seria usar nomes mais claros, como `getDayOfMonth()` , `getDayOfWeek()` e `getTimestamp()` (ou `getMillisSinceEpoch()` , ou qualquer outro nome menos genérico que `getTime()`). Nomes ruins por si só não tornam uma API ruim, mas também não ajudam a melhorá-la.

7.2 Construindo uma data específica

Mesmo que `Date` não seja exatamente uma data, um caso de uso bem comum é criar uma instância desta classe que represente uma data específica, em vez da data atual. Suponha que eu quero criar um `Date` para 10 de janeiro de 2018. Eu consulto a documentação e vejo que tem um construtor que recebe 3 parâmetros: ano, mês e dia (https://docs.oracle.com/javase/8/docs/api/java/util/Date.html#Date-int-int-int-/).

Eu até reparo que este construtor está *deprecated* desde o JDK `1.1` , mas resolvo usar mesmo assim, e já saio escrevendo o seguinte código:

```
// tentando criar 10 de janeiro de 2018
Date janeiro = new Date(2018, 1, 10);
System.out.println(janeiro);
```

Parece tudo certo, não? Usei o ano 2018, o mês 1 e o dia 10. Mas ao rodar o código, a saída é:

```
Sun Feb 10 00:00:00 BRST 3918
```

A data gerada foi **10 de fevereiro de 3918**! Eu deveria ter lido o restante da documentação, pois ela explica as regras usadas por este construtor:

- O ano é indexado em 1900, ou seja, o valor que você passar como parâmetro é somado a 1900, e o resultado (no caso, 1900 + 2018 = 3918) será o ano utilizado para construir o `Date` .
- O mês é indexado em zero, ou seja, janeiro é zero, fevereiro é 1 etc. A propósito, o método `getMonth()` retorna o valor do mês usando esta mesma regra.

Ou seja, temos que subtrair `1900` do ano e `1` do mês para que funcione:

```
// ano = 2018 - 1900, mês = zero
Date janeiro = new Date(118, 0, 10);
System.out.println(janeiro);
```

Agora sim, tenho a data correta (10 de janeiro de 2018):

```
Wed Jan 10 00:00:00 BRST 2018
```

Há alguns detalhes interessantes a se notar no resultado. O horário foi automaticamente setado para meia-noite, e mais uma vez foi usado o timezone padrão ("BRST" é a sigla para *Brasília Summer Time*, outro nome para o horário de verão brasileiro; mas o timezone continua sendo `America/Sao_Paulo`). Esta é uma consequência do fato de que `Date` representa um instante (um timestamp, um ponto específico na linha do tempo):

- Se eu tiver apenas o dia, mês e ano, não é o suficiente para ter um único valor de timestamp, pois quando é 10 de janeiro em São Paulo, pode ser dia 9 ou 11 em outras partes do mundo, dependendo do horário.
- Se eu tiver o dia, mês, ano e o horário, também não é o suficiente, pois 10 de janeiro de 2018 à meia-noite pode ocorrer em instantes diferentes, dependendo do timezone.

Para ter um único valor de timestamp — e ele é necessário para construir o `Date` — eu preciso da data, das horas e do offset (que é calculado usando-se as regras do timezone). Como no construtor eu só passei a data, ele usa valores predefinidos para os campos que estão faltando – no caso, meia-noite no timezone padrão da JVM.

7.3 Calendar, uma tentativa de melhorar Date

A classe `Date` possui construtores e vários outros métodos que estão *deprecated* desde o JDK `1.1` , pois foi nesta versão que surgiu a classe `java.util.Calendar` , como uma tentativa de melhorar e corrigir vários pontos problemáticos de `Date` .

Por exemplo, na classe `Calendar` existem constantes que representam os meses, para tentar amenizar o problema de janeiro ser o mês zero (o que causava — e ainda causa — muitos erros). Então, o exemplo anterior ficaria assim:

```
Date janeiro = new Date(118, Calendar.JANUARY, 10);
```

`Calendar.JANUARY` é uma constante `int` com valor zero, então na prática o problema continua, pois métodos como `getMonth()` continuarão retornando zero para janeiro, 1 para fevereiro etc. O uso das constantes apenas deixa o código um pouco menos confuso. Mas se você tiver um sistema no qual o usuário digita o valor do mês, por exemplo, vai precisar lembrar de subtrair 1 de qualquer jeito.

Como muitos construtores de `Date` passaram a ser *deprecated*, `Calendar` passou a ser usado para criar datas específicas. Primeiro, temos que criar uma instância, usando o método estático `getInstance()` . Depois, usamos o método `set()` para setar cada campo para o valor que queremos. Os campos são indicados através de constantes, conforme exemplificado no código a seguir:

```
// cria um Calendar
Calendar cal = Calendar.getInstance();
// muda o ano para 2018, o mês para janeiro e o dia do mês para 10
cal.set(Calendar.YEAR, 2018);
cal.set(Calendar.MONTH, Calendar.JANUARY);
cal.set(Calendar.DAY_OF_MONTH, 10);
System.out.println(cal);
```

As constantes `Calendar.YEAR` , `Calendar.MONTH` e `Calendar.DAY_OF_MONTH` são valores inteiros (do tipo `int`), e representam, respectivamente, os campos ano, mês e dia. É através delas que o método `set()` sabe o que deve ser mudado no `Calendar` .

Ao imprimir o `Calendar` , o resultado pode assustar um pouco:

```
java.util.GregorianCalendar[time=?,areFieldsSet=false,areAllFieldsSet=true
,lenient=true,zone=sun.util.calendar.ZoneInfo[id="America/Sao_Paulo",offse
t=-10800000,dstSavings=3600000,useDaylight=true,transitions=129,lastRule=j
ava.util.SimpleTimeZone[id=America/Sao_Paulo,offset=-10800000,dstSavings=3
600000,useDaylight=true,startYear=0,startMode=3,startMonth=10,startDay=1,s
tartDayOfWeek=1,startTime=0,startTimeMode=0,endMode=3,endMonth=1,endDay=15
,endDayOfWeek=1,endTime=0,endTimeMode=0]],firstDayOfWeek=1,minimalDaysInFi
rstWeek=1,ERA=1,YEAR=2018,MONTH=0,WEEK_OF_YEAR=18,WEEK_OF_MONTH=1,DAY_OF_M
ONTH=10,DAY_OF_YEAR=124,DAY_OF_WEEK=6,DAY_OF_WEEK_IN_MONTH=1,AM_PM=1,HOUR=
5,HOUR_OF_DAY=17,MINUTE=0,SECOND=0,MILLISECOND=0,ZONE_OFFSET=-10800000,DST
_OFFSET=0]
```

No meio deste texto enorme, podemos ver que há uma referência ao timezone `America/Sao_Paulo` . Ao contrário de `Date` , a classe `Calendar` possui um timezone, e `getInstance()` cria uma instância que usa o timezone padrão da JVM. Nas últimas linhas, podemos ver também o valor dos campos (`YEAR=2018` , `MONTH=0` e `DAY_OF_MONTH=10` , entre outros), e repare que o valor do mês é zero (janeiro).

A partir do `Calendar` , você pode obter um `Date` , usando o método `getTime()` — um nome que pode confundir um pouco, já que `Calendar.getTime()` retorna um `Date` , enquanto `Date.getTime()` retorna um `long` com o valor do timestamp.

Além disso, `Calendar` possui o método `getTimeInMillis()` — que é um nome um pouco melhor — que também retorna um `long` com o valor do timestamp. O código a seguir mostra o uso destes métodos:

```
Calendar cal =  ... // Calendar criado no exemplo anterior
// obter um java.util.Date a partir do Calendar
Date janeiro = cal.getTime();
// obter o valor do timestamp
long timestamp = janeiro.getTime();
// outra maneira de obter o timestamp, sem precisar criar o Date
long timestamp = cal.getTimeInMillis();
// também funciona, mas fica meio confuso (prefira usar getTimeInMillis())
long timestamp = cal.getTime().getTime();
```

Ao imprimir o `Date` retornado por `cal.getTime()` , teremos a saída:

```
Wed Jan 10 17:00:00 BRST 2018
```

A data foi setada corretamente para 10 de janeiro de 2018, mas o horário ficou como 17:00. Isso acontece porque `getInstance()` cria um `Calendar` usando a data e hora atual, no timezone padrão da JVM. E como eu só mudei os valores do dia, mês e ano, o horário se manteve.

Se eu quiser setar o horário para meia-noite, tenho que mudar a hora, minuto, segundo e milissegundos para zero, usando as respectivas constantes para estes campos:

```
// mudar horário para meia-noite
cal.set(Calendar.HOUR_OF_DAY, 0);
cal.set(Calendar.MINUTE, 0);
cal.set(Calendar.SECOND, 0);
cal.set(Calendar.MILLISECOND, 0);
```

Após isso, o `Date` retornado por `cal.getTime()` será 10 de janeiro de 2018, à meia-noite:

```
Wed Jan 10 00:00:00 BRST 2018
```

ONDE ESTÃO OS MILISSEGUNDOS?

O método `toString()` da classe `Date` não imprime os milissegundos. Mas o timestamp contém o número de milissegundos desde o Unix Epoch, e tanto `Date` quanto `Calendar` trabalham com esta precisão. Então existe um valor para este campo, ele só não é mostrado pelo método `toString()` . Uma decisão de implementação questionável, pois além de mostrar coisas que o `Date` não tem (como o timezone), não mostra tudo o que tem.

Também é possível obter o mesmo resultado setando vários campos ao mesmo tempo, pois o método `set()` pode receber de uma só vez os valores do ano, mês, dia. E há outra opção que também recebe a hora, minuto e segundo. Já os milissegundos devem ser setados separadamente:

```
// muda somente o ano, mês e dia (o horário não é mudado)
cal.set(2018, Calendar.JANUARY, 10);

// muda os campos para 10 de janeiro de 2018, meia-noite
cal.set(2018, Calendar.JANUARY, 10, 0, 0, 0);
// não se esqueça de mudar os milissegundos também
cal.set(Calendar.MILLISECOND, 0);
```

Cuidado com as horas

Um detalhe importante nos exemplos anteriores é que para mudar as horas eu usei o campo `Calendar.HOUR_OF_DAY` , que representa as horas do dia, com valores de 0 a 23.

Mas também existe o campo `Calendar.HOUR` , que representa a hora de um período (manhã ou tarde), e em inglês é chamado de *hour-of-am-pm* (algo como "hora AM/PM"). Este campo pode ter valores de de 0 a 11, e para que faça sentido, deve ser usado em conjunto com o campo `Calendar.AM_PM` (do contrário, o valor será ambíguo). Por exemplo, uma maneira de setar as horas para 17:00 é:

```
Calendar cal = Calendar.getInstance();
// mudar o horário para 17:00 (5 PM)
cal.set(Calendar.AM_PM, Calendar.PM);
cal.set(Calendar.HOUR, 5);
```

Para mudar o horário para 05:00 (5 da manhã), basta setar o valor do campo `Calendar.AM_PM` para `Calendar.AM` . Se você não setar o `AM_PM` , será usado o valor atual do `Calendar` . Por exemplo, se o horário do `Calendar` for 10:00, significa que o campo `AM_PM` possui o valor `AM` , e setar o campo `HOUR` para `5` resultará em "5 da manhã".

Este é mais um caso em que os nomes podem confundir um pouco. Os campos que correspondem aos minutos, segundos e milissegundos se chamam, respectivamente, `MINUTE` , `SECOND` e `MILLISECOND` , então é natural pensar que as horas do dia seriam representadas pelo campo `HOUR` . Mas, na verdade, `HOUR` se refere à hora de um período (AM ou PM), enquanto as horas do dia (0 a 23) são representadas por `HOUR_OF_DAY` . Talvez, se o campo `HOUR` se chamasse `HOUR_OF_AM_PM` , seu funcionamento ficasse mais claro.

Obter campos específicos de um Calendar

Assim como é possível mudar o valor de um campo usando `set()` , você pode obter o valor deste mesmo campo usando o método `get()` . Exemplos:

```
// valor numérico do dia
int dia = cal.get(Calendar.DAY_OF_MONTH);
// valor numérico do mês - janeiro é 0, fevereiro é 1 etc.
int mes = cal.get(Calendar.MONTH);
```

Vale lembrar que `get(Calendar.MONTH)` retorna o mês indexado em zero (ou seja, janeiro é zero, fevereiro é 1, e assim por diante).

Leia a documentação de cada campo, para evitar confusões como o caso de `HOUR` e `HOUR_OF_DAY` , já que os nomes nem sempre são tão claros quanto ao seu funcionamento. Outro exemplo é o campo `Calendar.DATE` , que, apesar do nome, não é uma data (dia, mês e

ano), mas sim um sinônimo para o campo `Calendar.DAY_OF_MONTH` , ou seja, retorna somente o dia do mês.

Não confunda campos com valores

Um erro comum ao usar `get()` e `set()` é confundir os parâmetros a serem usados. Todas as constantes de `Calendar` são valores numéricos (do tipo `int`), mas elas possuem significados diferentes:

- Algumas constantes representam **campos** de data ou hora, como `Calendar.MONTH` (mês) e `Calendar.DAY_OF_WEEK` (dia da semana).
- Outras representam **valores** de algum campo específico, como `Calendar.JANUARY` (janeiro, um dos valores válidos para `Calendar.MONTH`) e `Calendar.SUNDAY` (domingo, um dos valores válidos para `Calendar.DAY_OF_WEEK`).

Como todas as constantes são valores numéricos (`int`), o seu uso incorreto não é detectado em tempo de compilação. Além disso, constantes que correspondem a campos podem ter os mesmos valores numéricos de constantes que representam valores. Por exemplo, o campo `Calendar.MONTH` possui o valor `2` , que é o mesmo valor numérico de `Calendar.MARCH` e `Calendar.MONDAY` .

O código a seguir mostra um exemplo de uso incorreto: o objetivo é mudar o mês de um `Calendar` para outubro e, em seguida, pegar o valor do dia da semana. O código compila e roda normalmente, mas o comportamento não será exatamente o que se espera:

```
// data/hora atual: 2018-05-04T17:00-03:00 (America/Sao_Paulo)
Calendar cal = Calendar.getInstance();
// mudar o mês para outubro: parâmetros na ordem ERRADA (estão invertidos)
cal.set(Calendar.OCTOBER, Calendar.MONTH);
// imprime a data, para ver se fizemos tudo certo (spoiler: não fizemos)
System.out.println(cal.getTime());
// obter o dia da semana: usar um valor (FRIDAY) ao invés de um campo
(DAY_OF_WEEK)
System.out.println(cal.get(Calendar.FRIDAY));
```

Como todas as constantes são do tipo `int` , o código vai compilar e rodar normalmente. Porém, o valor da data e dia da semana não serão os esperados. Neste código, eu crio um `Calendar` com a data atual (4 de maio de 2018) e, em seguida, eu tento mudar o mês para outubro, portanto o resultado deveria ser 4 de outubro de 2018 (uma quinta-feira). Mas não foi bem isso que aconteceu.

A chamada do método `set()` está com os parâmetros invertidos e por isso o mês não será mudado para outubro. O primeiro parâmetro deve ser uma constante correspondente a um campo (por exemplo, `Calendar.MONTH`), mas eu passei `Calendar.OCTOBER` , cujo valor é `9` . O método `set()` procura, dentre as constantes que representam um campo, qual tem o valor igual a `9` , e nesse caso é `Calendar.AM_PM` .

O segundo parâmetro corresponde ao novo valor do campo, mas em vez disso eu passei uma constante que corresponde a um campo: `Calendar.MONTH` , cujo valor é `2` . Ou seja, o que o código faz, na verdade, é setar o valor `2` no campo `Calendar.AM_PM` . Neste campo, os valores definidos são `0` (AM) e `1` (PM). Como `2` ultrapassa o valor máximo do campo, ele é ajustado para o próximo período (AM) do dia seguinte. E como o horário do `Calendar` atual é 17:00 (ou 5 PM), o ajuste resulta em 05:00 (ou 5 AM) do dia seguinte (5 de maio). Se esse algoritmo faz sentido ou não, isso é outra história, mas é isso que acontece.

Por isso, o `Calendar` é setado para 5 de maio de 2018, às 05:00. Depois, o método `get()` é chamado, passando-se a constante equivalente à sexta-feira. Só que `Calendar.FRIDAY` tem o valor `6` , e `get()` procura, dentre as constantes que representam um campo, qual tem valor igual a `6` . No caso, esta constante é `Calendar.DAY_OF_YEAR` ("dia do ano"), e por isso o `get()` retorna `125` (pois 5 de maio de 2018 é o centésimo vigésimo quinto dia do ano).

Então, a saída do código é uma data completamente diferente do que queríamos, assim como o dia da semana, que deveria ser `5` (o valor da constante `Calendar.THURSDAY`):

```
Sat May 05 05:00:00 BRT 2018
125
```

Dependendo dos campos e valores utilizados erroneamente, os métodos `set()` e `get()` podem lançar uma exceção. Mas há vários casos como este, em que nenhuma exceção é lançada, e você só vai perceber que há algum problema quando alguém reclamar que o seu sistema está mostrando datas erradas.

7.4 Usando timezones com Calendar

Conforme já mencionado, `Calendar` possui um timezone. Quando você chama o método `getInstance()` , o `Calendar` criado usa o timezone padrão da JVM, e os respectivos campos (dia, mês, hora, minuto etc.) refletem o valor da data e hora atual neste timezone.

Se você mudar o timezone, usando o método `setTimeZone()` , os valores dos campos são recalculados. No próximo exemplo, temos um `Calendar` que usa o timezone padrão (`America/Sao_Paulo`), no qual a hora atual é 17:00. Ao mudar o timezone para `Europe/Berlin` , a hora passa a ser 22:00 (o horário é convertido para outro timezone):

```
// data/hora atual no timezone padrão: 4 de maio de 2018, 17:00,
America/Sao_Paulo
Calendar cal = Calendar.getInstance();
// imprimir hora e timestamp
System.out.println(cal.get(Calendar.HOUR_OF_DAY)); // 17
System.out.println(cal.getTimeInMillis()); // 1525464000000
// mudar timezone: horário muda, mas o timestamp continua o mesmo
cal.setTimeZone(TimeZone.getTimeZone("Europe/Berlin"));
System.out.println(cal.get(Calendar.HOUR_OF_DAY)); // 22
System.out.println(cal.getTimeInMillis()); // 1525464000000
```

O valor do timestamp permanece o mesmo, já que o `Calendar` continua se referindo ao mesmo instante — somente a data e a hora local referente ao timezone utilizado foram atualizadas. Por isso, o

`Date` retornado por `cal.getTime()` será o mesmo antes e depois de mudar o timezone. Lembre-se de que `Date` não possui timezone, apenas o valor do timestamp, e este valor não é alterado se mudarmos o timezone.

Criar Calendar em um timezone específico

Para criar um `Calendar` em um timezone específico, você não precisa fazer todos estes passos (criar um `Calendar` e depois usar `setTimeZone()`). Você pode passar o timezone diretamente para o `getInstance()` , assim:

```
// cria um Calendar com a data e hora atual no timezone Europe/Berlin
Calendar calBerlin =
Calendar.getInstance(TimeZone.getTimeZone("Europe/Berlin"));
```

Com isso, os campos do `Calendar` (dia, mês, ano, hora, minuto etc.) corresponderão à data e hora atual no timezone `Europe/Berlin` . Um detalhe importante é que, se você obtiver o `Date` correspondente (usando `calBerlin.getTime()`) e imprimi-lo com `System.out.println()` , a saída estará no timezone padrão da JVM, independente do timezone utilizado no `Calendar` . Isso acontece porque `Date` só possui o valor do timestamp, e o seu método `toString()` sempre usa o timezone padrão da JVM.

TimeZone.getTimeZone() não valida o nome

Um problema da classe `TimeZone` é que, se você passar um nome inválido para `getTimeZone()` , ele retorna uma instância correspondente a UTC. Na verdade, é criado um timezone chamado "GMT", cujo offset é zero e não tem horário de verão nem nenhuma outra mudança de offset — ou seja, na prática, é criada uma instância de `TimeZone` que corresponde a UTC.

Isso é um problema, porque um simples erro de digitação pode passar despercebido. Por exemplo, no código a seguir, o timezone `America/Sao_Paulo` foi digitado errado ("P**ua**lo" em vez de "P**au**lo"):

```
// tentativa de criar America/Sao_Paulo, mas tem um erro de digitação
System.out.println(TimeZone.getTimeZone("America/Sao_Pualo"));
                                                        ^^
```

A saída deste código será:

```
sun.util.calendar.ZoneInfo[id="GMT",offset=0,dstSavings=0,useDaylight=fals
e,transitions=0,lastRule=null]
```

Repare que o `id` é "GMT", o offset é zero (`offset=0`) e não há mudanças de offset (`transitions=0`) nem horário de verão (`useDaylight=false`). Se corrigirmos o erro de digitação:

```
// erro de digitação corrigido
System.out.println(TimeZone.getTimeZone("America/Sao_Paulo"));
```

Agora sim, a saída mostra o timezone que queremos (parte da saída foi omitida):

```
sun.util.calendar.ZoneInfo[id="America/Sao_Paulo",offset=-10800000,dstSavi
ngs=3600000,useDaylight=true,transitions=129 ...]]
```

Note que agora o `id` está certo ("America/Sao_Paulo"), com o offset e horário de verão corretamente configurados (`useDaylight=true`).

Se o método `getTimeZone()` lançasse uma exceção quando o nome fosse inválido, o erro seria percebido rapidamente. Mas como ele retorna uma instância "válida" (porém com o valor incorreto), você só perceberia o erro quando começasse a ver que as datas e horas do seu sistema estão em UTC. Dependendo do que você estiver fazendo com as datas, este é um erro difícil de perceber e debugar.

Uma maneira de evitar esta situação é verificar se o nome que você está usando é um dos timezones válidos disponíveis na JVM. A lista de todos os timezones que a JVM reconhece é retornada pelo método `TimeZone.getAvailableIDs()` . Outra maneira é verificar se o método `getID()` retorna o mesmo nome que foi passado para o método `getTimeZone()` .

TimeZone também aceita offsets

O método `getTimeZone()` não aceita somente os identificadores da IANA. Ele também aceita offsets, e o resultado é uma instância de `TimeZone` que possui apenas um único offset. Só há um porém: os offsets devem ter o prefixo "GMT" antes do seu valor, conforme mostra o próximo exemplo.

```
// cria um "timezone" que corresponde ao offset +03:00
TimeZone offset = TimeZone.getTimeZone("GMT+03:00");
```

Nos capítulos anteriores, vimos que timezone e offset, embora relacionados, são dois conceitos diferentes. Um offset é simplesmente a diferença com relação a UTC, e um timezone é uma lista com o histórico de todos os offsets que uma região teve, tem e terá.

Apesar disso, a classe `TimeZone` implementa os dois conceitos. Se o método `getTimeZone()` recebe um identificador da IANA, retorna uma instância que possui o histórico de offsets correspondente.

E, se este método recebe um offset, a instância retornada possui apenas o valor de offset que foi passado. Para ficar menos confuso, podemos pensar que o método retorna um "timezone" que possui um histórico com apenas um offset.

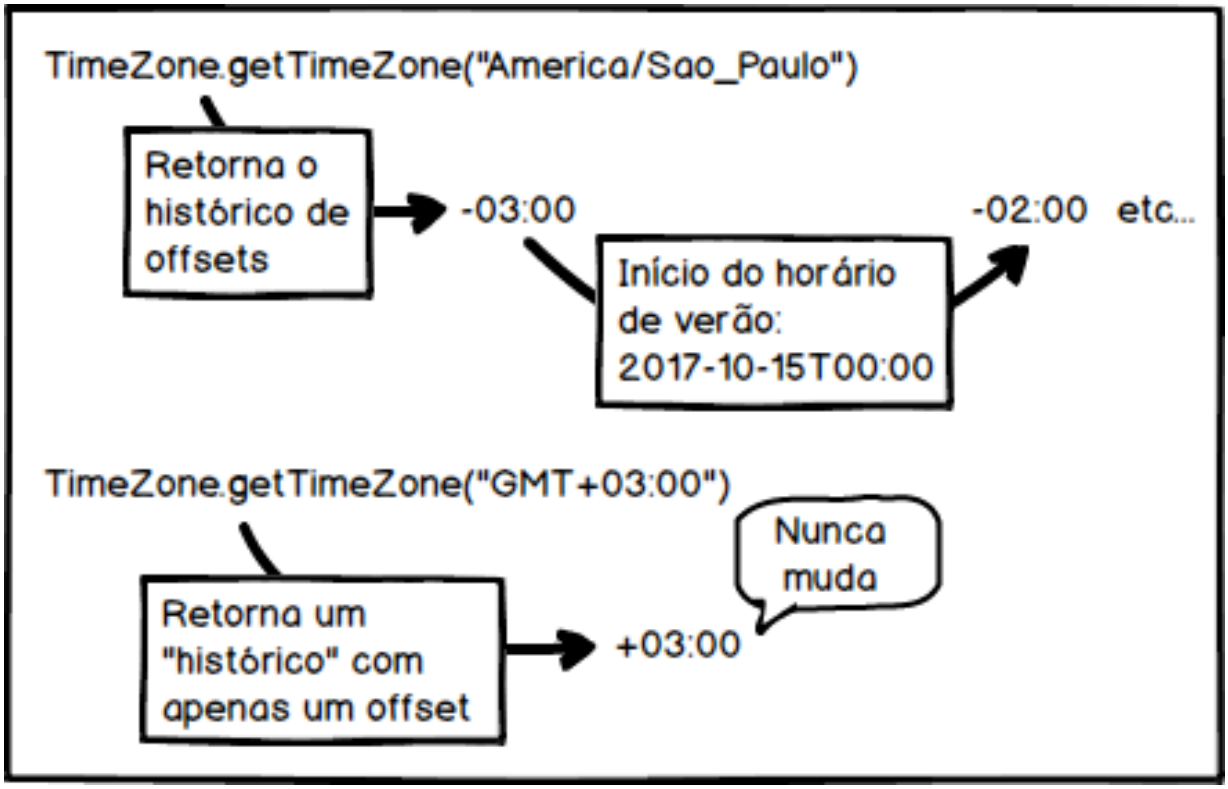

Figura 7.2: TimeZone.getTimeZone() pode receber um timezone ou um offset

Um ponto de atenção é que o método `getTimeZone()` só aceita offsets com o prefixo "GMT". Se eu chamar sem o prefixo (`getTimeZone("+03:00")`) não dará erro, mas a instância retornada corresponderá a UTC, pois o offset sem o prefixo é considerado um parâmetro inválido, similar ao que acontece no exemplo do erro de digitação.

7.5 Calendar aceita qualquer valor

Ao mudar um campo de um `Calendar` , podemos usar valores que ultrapassam o máximo permitido. Por exemplo, eu posso setar o dia do mês para `33` (apesar de nenhum mês poder ter mais que 31 dias):

```
// 2018-05-04T17:00-03:00 (America/Sao_Paulo)
Calendar cal = Calendar.getInstance();
```

```
// mudar o dia do mês para 33
cal.set(Calendar.DAY_OF_MONTH, 33);
System.out.println(cal.getTime());
```

A saída será 2 de junho:

```
Fri Jun 02 17:00:00 BRT 2018
```

`33` ultrapassa o máximo permitido no mês em dois dias (pois maio só tem 31 dias), então a data é ajustada para o dia 2 do mês seguinte (junho). Este comportamento que permite qualquer valor e *"deixa que eu me viro"* é chamado de **leniente**. Esta palavra é sinônimo de "brando" ou "tolerante", e tem exatamente este sentido: permitir (ou tolerar) valores inicialmente inválidos, sem restringi-los, e tentar ajustá-los de alguma maneira. É também o nome usado pela API (em inglês, *"lenient"*) e é o termo que usarei a partir de agora.

Por padrão, `Calendar` é leniente e tenta fazer todos os ajustes possíveis. O comportamento é similar ao que é feito na aritmética de datas: é calculado o excedente do campo, para em seguida ajustar os demais campos. Por exemplo, ao setar as horas para `25` , o excedente é `1` (pois em um dia só são permitidas 24 horas) e, por isso, o `Calendar` é ajustado para 01:00 do dia seguinte.

Mas nem sempre queremos que isso aconteça. Se você quer garantir que somente valores válidos (dentro dos limites de cada campo) sejam aceitos e nenhum ajuste seja feito, basta desligar o modo leniente, usando o método `setLenient()` :

```
// 2018-05-04T17:00-03:00 (America/Sao_Paulo)
Calendar cal = Calendar.getInstance();
// desligar o modo leniente, não permitir valores inválidos
cal.setLenient(false);
// mudar o dia do mês para 33
cal.set(Calendar.DAY_OF_MONTH, 33);
```

Com o modo leniente desligado, o código anterior lança uma exceção, indicando que o campo recebeu um valor inválido:

```
java.lang.IllegalArgumentException: DAY_OF_MONTH
```

É possível verificar se o `Calendar` está em modo leniente ou não, usando o método `isLenient()` , que retorna `true` quando está leniente, e `false` quando não está.

Com isso, cobrimos o funcionamento básico de `Date` e `Calendar` . Nos próximos capítulos, veremos como esta API implementa os conceitos de formatação e parsing.

Capítulo 8
Formatação usando SimpleDateFormat e Locale

Datas não têm formato. Elas só têm valores: um `Date` possui o valor numérico do timestamp (quantidade de milissegundos decorridos deste o Unix Epoch: `1970-01-01T00:00Z`); e um `Calendar` possui um timestamp, um timezone e os respectivos campos: dia, mês, ano, hora, minuto, segundo etc.

Estes valores podem ser convertidos para um formato específico, mas `Date` e `Calendar` , por si só, não possuem nenhum formato. Quem possui um formato e é responsável por converter as datas de/para este formato é a classe `java.text.SimpleDateFormat` .

Já falamos sobre os conceitos de formatação e parsing. Apenas para relembrar, a ideia básica é:

- Se eu tenho um tipo de data/hora (como `Date` ou `Calendar`) e quero mostrar esta informação em um formato específico, eu tenho que convertê-lo para uma `String` . Este processo é chamado **formatação**.
- Se eu tenho uma `String` que representa uma data/hora e quero transformá-la em um objeto que representa esta data/hora, eu converto a `String` para o tipo correspondente (como `Date` ou `Calendar`). Este processo é chamado **parsing**.

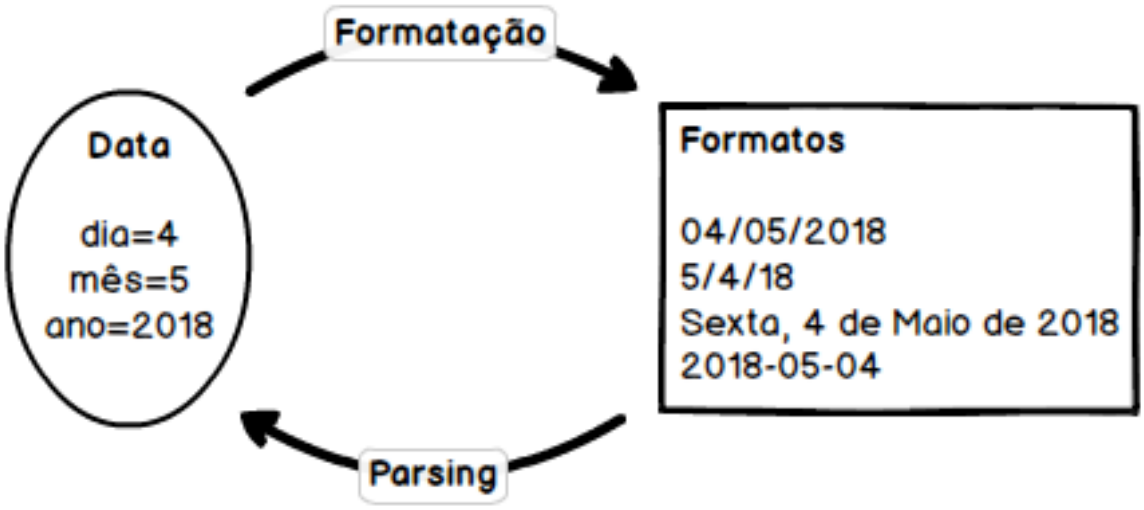

Figura 8.1: Formatação e parsing, não confunda!

A classe `SimpleDateFormat` serve tanto para formatar quanto para fazer o parsing. Neste capítulo veremos exemplos de formatação.

8.1 Formatação: converter data para texto

Para formatar (converter a data para um formato específico), você precisa dizer qual é o formato a ser utilizado. Por exemplo, se eu quiser exibir um `Date` no formato "dia/mês/ano", eu crio um `SimpleDateFormat` com este formato, e chamo o método `format()` :

```
// usar o formato dia/mês/ano
SimpleDateFormat formatter = new SimpleDateFormat("dd/MM/yyyy");
// formatando a data atual
Date dataAtual = new Date();
String dataFormatada = formatter.format(dataAtual);
System.out.println(dataFormatada);
```

Como a data/hora atual em nossos exemplos é `2018-05-04T17:00-03:00` (4 de maio de 2018, às 17:00 no timezone `America/Sao_Paulo`), a saída será:

`04/05/2018`

Um dos pontos mais importantes sobre formatação e parsing de datas é o parâmetro usado no construtor de `SimpleDateFormat` (no exemplo anterior, `dd/MM/yyyy`). Este parâmetro é chamado de *pattern*.

> Em português, a palavra "padrão" é uma tradução válida para *pattern*, *default* e *standard*. Como você já deve ter percebido, estou usando-a no lugar de *default* (como em "timezone padrão da JVM" para traduzir *"JVM default timezone"*) e, no lugar de *standard*, estou usando "norma" (como em "norma ISO 8601"). Já para me referir a *pattern*, usarei o termo em inglês, para evitar confusão com os outros termos.

A `String` que usamos como pattern (`dd/MM/yyyy`) é o que define o formato a ser usado. No caso, ele significa:

- `dd` : dia do mês com 2 dígitos (por isso, o dia 4 é impresso como `04`)
- `/` : é o próprio caractere `/`
- `MM` : mês com 2 dígitos (por isso, o mês 5 é impresso como `05`)
- `/` : o caractere `/` novamente
- `yyyy` : ano com 4 dígitos

E de onde vêm estas letras? Da documentação (https://docs.oracle.com/javase/8/docs/api/java/text/SimpleDateFormat.html/). Sugiro que você entre neste link e veja que há diferença entre usar `M` maiúsculo e `m` minúsculo (um erro bem comum). Cada uma representa um campo diferente: o `M` maiúsculo é usado para o mês e `m` minúsculo é usado para os minutos. **Letras maiúsculas e minúsculas fazem diferença no pattern**.

Um dos erros mais comuns é usar a letra errada e perder tempo tentando descobrir porque a `String` resultante não mostra a data correta. Um que acontece bastante é usar `dd/mm/yyyy` , que na

verdade significa "dia/minutos/ano", pois foi usado o `m` minúsculo ao invés do maiúsculo.

Todas estas letras são usadas pela classe `SimpleDateFormat`, mas nada garante que outras classes e outras linguagens usem as mesmas letras para os mesmos campos. Como veremos mais adiante, na API `java.time` nem todas as letras significam as mesmas coisas. Até mesmo em outras versões do JDK podem existir diferenças: um exemplo é a letra `X`, usada para offsets, que só foi introduzida no JDK 7. Nunca assuma que o mesmo pattern vai funcionar da mesma maneira em outra API, e principalmente em outra linguagem. Sempre leia a documentação.

> Se você seguir a tag `simpledateformat` no Stack Overflow (https://stackoverflow.com/questions/tagged/simpledateformat/), verá que praticamente toda semana surge alguma pergunta relacionada, e a resposta geralmente consiste em apenas trocar `D` por `d`, ou `m` por `M`, ou algo do tipo.

A quantidade de letras também pode mudar o resultado final. Como eu usei `dd`, significa que o dia vai ser escrito com 2 dígitos, e por isso o dia 4 se torna `04`. Mas se eu usar o pattern `d/MM/yyyy`, a saída será `4/05/2018`.

Mas eu não tinha falado que Date não é uma data?

`Date` não representa uma única data (um dia, mês e ano específicos) e, sim, um ponto na linha do tempo (um valor numérico para o timestamp, a quantidade de milissegundos decorridas desde o Unix Epoch). E um mesmo valor de timestamp corresponde a uma data e hora diferentes, dependendo do timezone em que você está.

Então, como o `SimpleDateFormat` consegue retornar um dia, mês e ano específicos, se o `Date` não representa uma única data? Simples: ele usa o timezone padrão da JVM — que nos nossos exemplos, é `America/Sao_Paulo`. Neste timezone, o `Date` (na verdade, o valor do

timestamp que ele contém) corresponde a 4 de maio de 2018, e essa é a data utilizada para gerar a `String` .

Se eu mudar o timezone padrão, a saída gerada poderá ser diferente. Por exemplo, se mudarmos para `Asia/Tokyo` e, em seguida, criarmos o `SimpleDateFormat` , ele usará este timezone para obter os valores da data:

```
// mudar o timezone padrão
TimeZone.setDefault(TimeZone.getTimeZone("Asia/Tokyo"));
// usar o formato dia/mês/ano
SimpleDateFormat formatter = new SimpleDateFormat("dd/MM/yyyy");
// formatando a data atual (2018-05-04T17:00-03:00)
System.out.println(formatter.format(new Date()));
```

Agora a saída é:

```
05/05/2018
```

Afinal, no mesmo instante em que é `2018-05-04T17:00-03:00` em São Paulo (4 de maio de 2018, às 17:00), em Tóquio já é `2018-05-05T05:00+09:00` (**5** de maio de 2018, às 05:00).

No código anterior o `SimpleDateFormat` está usando o timezone `Asia/Tokyo` , pois este era o timezone padrão da JVM no momento em que ele foi criado. Um detalhe importante é que, uma vez criado o `SimpleDateFormat` , este valor é mantido, mesmo que eu mude o timezone padrão depois.

Usar um timezone específico

`SimpleDateFormat` sempre usa o timezone padrão da JVM. E se eu quiser usar outro timezone, como faço? Simples, basta usar o método `setTimeZone()` :

```
// usar o formato dia/mês/ano
SimpleDateFormat formatter = new SimpleDateFormat("dd/MM/yyyy");
// usar um timezone específico
formatter.setTimeZone(TimeZone.getTimeZone("Asia/Tokyo"));
```

```
// formatando a data atual
System.out.println(formatter.format(new Date()));
```

A saída será:

```
05/05/2018
```

O `SimpleDateFormat` foi configurado para usar o timezone `Asia/Tokyo` . Ou seja, o valor do timestamp que está na classe `Date` é convertido para a data e hora correspondente neste timezone. A seguir, esses valores são convertidos em uma `String` , no formato definido pelo pattern. Desta forma, não importa qual é o timezone padrão da JVM, o `SimpleDateFormat` usará o que estiver setado nele.

Mas janeiro não era o mês zero?

Este é outro ponto confuso da API. Quando eu uso `Calendar.get()` ou `Calendar.set()` com o campo `Calendar.MONTH` , os valores usados são indexados em zero. Ou seja, janeiro é zero, fevereiro é 1 etc. Tanto que as constantes para os meses possuem exatamente estes valores (`Calendar.JANUARY` é zero, `Calendar.FEBRUARY` é 1, e assim por diante).

Mas quando `SimpleDateFormat` vai converter estas datas para uma `String` , são usados os valores normais, indexados em 1. Por isso, ao formatar, janeiro vira 1, fevereiro vira 2 etc.

Além disso, tem mais um detalhe meio chato. Se eu passar um `Calendar` para o método `format()` , ele lança uma `IllegalArgumentException` . Quando você quiser formatar um `Calendar` , deve usar o método `getTime()` , que retorna o `Date` correspondente:

```
Calendar cal = Calendar.getInstance();
SimpleDateFormat formatter = new SimpleDateFormat("dd/MM/yyyy");
// getTime() retorna o java.util.Date
System.out.println(formatter.format(cal.getTime()));
```

8.2 Usar nomes em vez de números

É muito comum querer converter uma data para formatos que tenham o nome do mês ou do dia da semana, e isso também é possível com `SimpleDateFormat`. No caso do mês, basta usar 3 ou 4 letras `M`, e para o dia da semana, a letra `E`. Exemplo:

```
SimpleDateFormat formatter = new SimpleDateFormat("EEE dd/MMM/yyyy");
System.out.println(formatter.format(new Date()));
```

A saída será:

```
Sex 04/mai/2018
```

Mas se eu usar 4 letras para o mês e dia da semana (`EEEE dd/MMMM/yyyy`), a saída será `Sexta-feira 04/Maio/2018`.

Isso porque, segundo a documentação (https://docs.oracle.com/javase/8/docs/api/java/text/SimpleDateFormat.html/), se o pattern tem 3 letras ou mais, o mês e o dia da semana são interpretados como texto. E 3 letras fazem com que a forma abreviada seja usada (`Mai`), enquanto 4 letras ou mais usam a forma completa (`Maio`).

Os nomes estão em português mas, dependendo da configuração da JVM que você estiver usando, pode ser que você obtenha a saída em outro idioma. Para entender como isso funciona, vamos fazer uma pausa no assunto formatação e falar rapidamente sobre localização.

Localização, mas não é a do GPS

De forma bem resumida, em computação, **Internacionalização e Localização** é a adaptação de um software para vários idiomas, levando em conta diferenças culturais e regionais.

Além da tradução e do uso de palavras e expressões específicas de cada variante (como o inglês britânico e americano, por exemplo),

também são considerados outros aspectos culturais e regionais, como formatos de números (o número "mil cento e vinte e três e meio" é escrito como `1.123,5` no Brasil e `1,123.5` nos EUA), formatos de datas (4 de maio de 2018: no Brasil é `04/05/18` e nos EUA, `5/4/18`), entre outros.

Em Java, os aspectos relativos à localização são controlados pela classe `java.util.Locale` .

> Não encontrei uma tradução satisfatória para *locale*. Em português, seria "local", mas esta palavra não passa toda a ideia que um `Locale` representa (aspectos culturais, regionais e políticos de uma região geográfica específica). Por isso, usarei o termo em inglês.

Assim como a JVM possui um timezone padrão, ela também possui um locale padrão (*JVM default locale*), que pode ser acessado pelo método estático `Locale.getDefault()` . A minha JVM está configurada para "português do Brasil", e se eu imprimir o locale padrão (`System.out.println(Locale.getDefault())`) a saída será `pt_BR` , que é o código deste locale.

Que código é esse?

Sem entrar em muitos detalhes, pois localização não é o foco do livro, os códigos de locale são compostos por vários elementos:

- linguagem/idioma: usa os códigos definidos pela ISO 639 (https://pt.wikipedia.org/wiki/ISO_639/). Exemplos: `en` (inglês) e `pt` (português);
- país: usa os códigos definidos pela ISO 3166 (https://en.wikipedia.org/wiki/ISO_3166/). Exemplos: `US` (EUA) e `BR` (Brasil);
- variante, script e extensões: informações adicionais para indicar variações ou outras características específicas de um determinado locale. Veja a documentação para mais detalhes

(https://docs.oracle.com/javase/8/docs/api/java/util/Locale.html/)
.

Portanto, `pt_BR` indica que o locale refere-se ao idioma português (`pt`) que é falado no Brasil (`BR`). Já o português falado em Portugal é definido pelo locale `pt_PT` (pois `PT` é o código para Portugal). É possível criar estes locales usando o construtor da classe `Locale` :

```
// português do Brasil (pt_BR)
Locale ptBrasil = new Locale("pt", "BR");
// português de Portugal (pt_PT)
Locale ptPortugal = new Locale("pt", "PT");
```

Também é possível usar apenas `new Locale("pt")` . Neste caso, o locale refere-se ao idioma português, porém sem levar em conta as variações entre Brasil e Portugal, já que o país não foi especificado.

A classe `Locale` possui algumas constantes predefinidas, como `ENGLISH` (idioma inglês), `FRENCH` (francês), `US` (inglês americano), `UK` (inglês britânico), entre outras.

Agora que já temos uma ideia básica do que é um locale, vamos ver como usá-lo com `SimpleDateFormat` .

Formatação usando outros idiomas

Quando eu faço `new SimpleDateFormat(pattern)` , a instância criada usará o locale padrão da JVM. Como a minha JVM está configurada para usar `pt_BR` (português do Brasil), a saída do método `format()` possui o nome do mês e dia da semana em português.

O problema é que, assim como o timezone padrão pode ser mudado, o locale padrão também pode, através do método `Locale.setDefault()` . Se eu mudar o locale padrão para inglês, o `SimpleDateFormat` que for criado depois disso vai usar este idioma:

```
// mudar o locale padrão para inglês
Locale.setDefault(Locale.ENGLISH);
// usar o formato dia/mês/ano
```

```
SimpleDateFormat formatter = new SimpleDateFormat("EEEE dd/MMMM/yyyy");
// formatando a data atual
System.out.println(formatter.format(new Date()));
```

Como eu mudei o locale padrão para `Locale.ENGLISH` , a saída terá os nomes dos campos em inglês:

```
Friday 04/May/2018
```

Devemos ter com `Locale.setDefault()` os mesmos cuidados que temos com `TimeZone.setDefault()` . Este método muda o locale padrão para toda a JVM, e todas as aplicações rodando nessa mesma JVM serão afetadas. O locale também pode mudar caso o servidor altere sua configuração, ou através de *system properties*, conforme explicado neste link (https://stackoverflow.com/a/8809162/). Não é um erro fatal usar ou alterar o locale padrão, desde que você esteja ciente do que está sendo feito e de como isso afeta a sua aplicação.

Um detalhe importante é que o construtor de `SimpleDateFormat` usa o locale padrão que estiver setado no momento em que ele é chamado. Se eu mudar o locale padrão, esta mudança não interfere no idioma das instâncias de `SimpleDateFormat` que já existiam antes.

Uma maneira de não depender do locale padrão é passar o que você quer para o `SimpleDateFormat` . Se eu quero que a saída esteja sempre em inglês, basta informar isso explicitamente no construtor:

```
// usar um locale específico
SimpleDateFormat formatter = new SimpleDateFormat("dd/MMM/yyyy",
Locale.ENGLISH);
```

Desta forma, o `SimpleDateFormat` sempre vai gerar a saída em inglês (`Locale.ENGLISH`), independente de qual for o locale padrão. Curiosamente, não existe um método `setLocale()` nesta classe, então o único jeito de usar um locale específico é passá-lo no construtor.

Campos que mudam conforme o locale

Conforme já vimos, o nome dos meses e dias da semana são dois campos que mudam conforme o locale. Os campos que mudam sua representação conforme o locale são chamados de *locale sensitive* (algo como "sensíveis ao locale") ou *localized* ("localizados"). Sempre que você encontrar algum destes termos na documentação da API, saiba que está relacionado a um locale.

Muitas vezes, o locale é deixado de lado pelos desenvolvedores, simplesmente porque o servidor está configurado com o idioma "correto" e o código "funciona". Mas se você está usando campos *locale sensitive* e já sabe qual o idioma a ser usado, prefira sempre usar o respectivo locale explicitamente. É melhor do que depender da configuração padrão e ser surpreendido caso ela mude de repente.

Dependendo da versão do JDK, a saída em português pode gerar o nome do mês começando com letra maiúscula (como "Maio", por exemplo) ou minúscula ("maio"). Muitas informações de localização foram alteradas no JDK 8 (https://docs.oracle.com/javase/8/docs/technotes/guides/intl/enhancements.8.html#cldr/), o que explica esta diferença. E no JDK 9, também houve mudanças, detalhadas neste link (https://docs.oracle.com/javase/9/migrate/toc.htm#JSMIG-GUID-A20F2989-BFA9-482D-8618-6CBB4BAAE310/), o que pode alterar a saída em alguns locales, conforme discutido nesta resposta do Stack Overflow (https://stackoverflow.com/a/46245412/).

E para o JDK 11, foram feitas mais atualizações referentes à localização. Ou seja, dependendo do locale, as `Strings` resultantes podem ser diferentes – os detalhes podem ser encontrados no *JDK Bug System* (https://bugs.openjdk.java.net/browse/JDK-8202537/).

Não confunda locale com timezone

Apesar de o locale ter um código de país, ele não indica que eu quero usar o timezone deste país. Por exemplo, ao usar `new Locale("pt", "PT")` (português de Portugal), isso não quer dizer que eu

vou necessariamente usar o timezone de Portugal. Até porque não é possível mapear um código de país para um único timezone. Portugal, por exemplo, atualmente tem 3: `Europe/Lisbon` , `Atlantic/Madeira` (Ilha da Madeira) e `Atlantic/Azores` (Açores). E vários outros países possuem mais de um timezone.

O locale só vai controlar os aspectos relativos à localização. Quando usado com um `SimpleDateFormat` , os campos que são afetados pelo locale (que são *locale sensitive*) serão convertidos para as respectivas `Strings` , no idioma representado pelo locale e de acordo com o pattern usado.

O timezone, por sua vez, é usado para definir os valores de data, hora e offset, já que um `Date` pode representar um dia e horário diferente em cada timezone.

Sendo assim, eu posso criar um `SimpleDateFormat` que mostra as informações em inglês, mas usando o timezone do Japão:

```
// formatter em inglês
SimpleDateFormat formatter = new SimpleDateFormat("dd/MMM/yyyy",
Locale.ENGLISH);
// converter a data para o timezone do Japão
formatter.setTimeZone(TimeZone.getTimeZone("Asia/Tokyo"));
// formatando a data atual
System.out.println(formatter.format(new Date()));
```

A saída terá o nome do mês em inglês (por causa do `Locale.ENGLISH`), porém os valores do dia, mês e ano serão definidos pelo timezone (`Asia/Tokyo`). A saída será:

```
05/May/2018
```

Mudar o locale não afeta o timezone, e vice-versa.

8.3 Formatar para ISO 8601

Na primeira parte do livro, no capítulo sobre formatos de data, vimos o formato definido pela norma ISO 8601. Para converter uma data para uma `String` neste formato (por exemplo, `2018-05-04T17:00-03:00`), é muito comum que a primeira tentativa de criar o `SimpleDateFormat` seja algo parecido com isso:

```
SimpleDateFormat iso8601Format = new SimpleDateFormat("yyyy-MM-
ddTHH:mm:ss.SSSXXX");
```

Porém, este código lança uma exceção:

```
java.lang.IllegalArgumentException: Illegal pattern character 'T'
```

Isso acontece porque a letra `T` não é usada pelos patterns. Ela não é como o `d` , que representa o dia do mês, ou `H` , que representa as horas. O `T` é uma letra que não representa nenhum campo, e por isso é um caractere inválido para o pattern (conforme a própria exceção indica: *Illegal pattern character*). Veja novamente a documentação para saber quais letras são aceitas (https://docs.oracle.com/javase/8/docs/api/java/text/SimpleDateFormat.html/).

Só que, neste caso, eu não quero um campo específico (como o dia, ou as horas). Eu quero a própria letra `T` . Para isso, eu tenho que colocá-la entre aspas simples (`'`), assim:

```
SimpleDateFormat iso8601Format = new SimpleDateFormat("yyyy-MM-
dd'T'HH:mm:ss.SSSXXX");
```

Colocar um texto entre aspas simples faz com que ele deixe de ser interpretado como um campo e passe a ser interpretado literalmente como o próprio texto. Por isso, este trecho de texto entre as aspas também é chamado de **"literal"**.

Ou seja, o `T` entre aspas simples (`'T'`) deixa de ser "um caractere inválido para patterns" (que lança uma exceção) e passa a ser "a letra T maiúscula". Como o formato ISO 8601 define que para separar a data da hora sempre usamos um `T` maiúsculo, então é exatamente isso que precisamos.

Para a data, usei `yyyy-MM-dd` (ano, mês, dia, separados por hífen), mantendo a quantidade de dígitos que a ISO 8601 define (ano com 4 dígitos, mês e dia com 2).

Para as horas, eu usei `HH` , que é a "hora do dia", com valores entre 0 e 23, que é condizente com os valores definidos pela ISO 8601. Um erro comum é usar `hh` para estes casos, que representa a "hora AM/PM" e tem valores entre 1 e 12 – e portanto só faz sentido se usarmos junto com o designador de AM/PM (o pattern `a`). Como a ISO 8601 não permite AM/PM, o correto é usar `HH` (também com 2 dígitos, conforme a norma).

A seguir, eu uso `mm` para os minutos, e lembre-se de que não é a mesma coisa que `MM` (que representa o mês). Letras maiúsculas e minúsculas fazem diferença no pattern. Depois, eu uso `ss` para designar os segundos, seguido de um ponto (`.`). Como o ponto não é uma letra, não preciso colocá-lo entre aspas simples, pois ele já é automaticamente reconhecido como um literal. Como você já deve ter notado, o mesmo vale para outros caracteres, como a barra, o hífen e os dois pontos.

Logo em seguida, coloco `SSS` para os milissegundos. As classes `Date` e `Calendar` possuem precisão de milissegundos, portanto 3 dígitos é o máximo suportado para este campo — você até pode colocar mais que 3 letras `S` , mas isso pode causar alguns problemas que serão explicados no próximo capítulo.

Por fim, eu uso `XXX` , que corresponde ao offset. Esta letra foi introduzida no JDK 7 (no JDK 6 ela lança uma `IllegalArgumentException`) e você pode usar de 1 a 3 letras, obtendo resultados diferentes para cada quantidade:

- `XXX` : imprime o offset com dois pontos. Ex.: `-03:00` .
- `XX` : imprime o offset sem os dois pontos. Ex.: `-0300` .
- `X` : imprime o offset, mas somente o valor das horas. Ex.: `-03` .

O último caso (`X`) é perigoso porque pode omitir uma informação importante, caso o offset não seja de horas inteiras. Exemplo:

atualmente (maio de 2018) a Índia usa o offset `+05:30` , porém o pattern com somente um `X` imprime `+05` . As outras opções imprimem o valor correto (`+0530` ou `+05:30`).

Usando este `SimpleDateFormat` :

```
SimpleDateFormat iso8601Format = new SimpleDateFormat("yyyy-MM-
dd'T'HH:mm:ss.SSSXXX");
// formatando a data atual: 4 de maio de 2018, 17:00, America/Sao_Paulo
System.out.println(iso8601Format.format(new Date()));
```

A saída é:

```
2018-05-04T17:00:00.000-03:00
```

A ISO 8601 permite que os segundos e frações de segundo sejam omitidos, caso o valor seja zero, mas neste caso, o `SimpleDateFormat` sempre imprime todos os campos que estão no pattern. E como ele usa o timezone padrão da JVM (que, no meu caso, é `America/Sao_Paulo`), internamente é verificado que, para o instante correspondente ao timestamp da classe `Date` , o offset usado neste timezone é `-03:00` .

Se eu quiser gerar uma `String` que corresponde a esta data em outro timezone, basta setar o timezone desejado no `SimpleDateFormat` , como já vimos. Eu posso inclusive setá-lo para UTC:

```
SimpleDateFormat iso8601Format = new SimpleDateFormat("yyyy-MM-
dd'T'HH:mm:ss.SSSXXX");
// usar UTC
iso8601Format.setTimeZone(TimeZone.getTimeZone("UTC"));
// formatando a data atual: 4 de maio de 2018, 17:00, America/Sao_Paulo
System.out.println(iso8601Format.format(new Date()));
```

Neste caso, a saída será:

```
2018-05-04T20:00:00.000Z
```

O horário mudou para `20:00` , já que `17:00` no offset `-03:00` corresponde a `20:00` em UTC. E veja que o offset foi mostrado como `Z` , já que esta é a forma mais comum de escrever o offset zero.

Agora que já vimos os casos mais comuns de formatação, vamos ao próximo capítulo, no qual veremos como fazer parsing com `SimpleDateFormat`.

Capítulo 9
Parsing com SimpleDateFormat

No capítulo anterior, vimos como converter `Date` para `Strings` nos mais diversos formatos. Agora veremos como fazer o oposto, e já adianto que não é tão simples quanto parece. Há vários detalhes com que temos que nos preocupar, como veremos a seguir.

9.1 Parsing: converter texto para data

Para fazer o parsing (converter um texto para uma data), você precisa informar em qual formato está o texto. Mas há vários detalhes aos quais você deve prestar atenção ao fazê-lo. Vamos supor que eu tenha um `Date` com a data atual, que em seguida foi formatado para uma `String` :

```
// usar o formato dia/mês/ano
SimpleDateFormat formatter = new SimpleDateFormat("dd/MM/yyyy");
// 2018-05-04T17:00-03:00 (4 de maio de 2018, 17:00, America/Sao_Paulo)
Date dataAtual = new Date();
// imprimir o Date e o timestamp
System.out.println(dataAtual);
System.out.println(dataAtual.getTime());
// formatar a data
String dataFormatada = formatter.format(dataAtual);
System.out.println(dataFormatada);
```

A saída é:

```
Fri May 04 17:00:00 BRT 2018
1525464000000
04/05/2018
```

A variável `dataAtual` possui o valor do timestamp igual a `1525464000000` . E a `String` resultante (`dataFormatada`) é igual a `04/05/2018` , já que o

`SimpleDateFormat` está usando o timezone padrão da JVM (`America/Sao_Paulo`) e, neste timezone, o timestamp `1525464000000` corresponde a 4 de maio de 2018.

Vamos supor que esta `String` (`04/05/2018`) foi passada como parâmetro para algum serviço, ou gravada em um arquivo, ou o que for. O serviço que recebeu esta `String` vai fazer o parsing para reconstruir o objeto `Date` a partir da data formatada. Para isso, é construído um `SimpleDateFormat` com o pattern correspondente à `String` , e em seguida é chamado o método `parse()` , que retorna um `Date` :

```
// usar o formato dia/mês/ano
SimpleDateFormat parser = new SimpleDateFormat("dd/MM/yyyy");
try {
    // parsing: converte a String para java.util.Date
    Date data = parser.parse("04/05/2018");
    System.out.println(data);
    System.out.println(data.getTime());
} catch (ParseException e) {
    // faz algo com a exceção
}
```

O método `parse()` pode lançar um `java.text.ParseException` e, como esta é uma *checked exception*, temos que colocar o `catch` . A partir de agora, **em todos os exemplos**, este `try/catch` será omitido por questões de brevidade.

A seguir, imprimimos o `Date` retornado pelo método `parse()` e o valor do timestamp. A saída é:

```
Fri May 04 00:00:00 BRT 2018
1525402800000
```

A data obtida através do parsing não é a mesma que foi formatada. O horário do `Date` original era 17:00, mas o horário deste `Date` é meia-noite. Por isso, os valores do timestamp também são diferentes.

Isso acontece porque a `String` que foi passada para o método `parse()` só possui dia, mês e ano. Apesar de esta `String` ser o resultado da

formatação de um `Date` com um valor de timestamp específico, ela não tem a informação completa (data, hora e offset/timezone). A `String` não sabe que ela é o resultado da formatação de um `Date`, nem que o valor do timestamp era `1525464000000`. Ela só sabe que seu valor é `"04/05/2018"`, mas não tem informação nenhuma sobre o contexto no qual foi gerada.

Sendo assim, o método `parse()` vai trabalhar somente com as informações que possui para retornar um `Date`. A `String` de entrada possui dia igual a 4, mês 5 e ano 2018 (valores obtidos ao fazer a correspondência de `04/05/2018` com o pattern `dd/MM/yyyy`). E tendo apenas um dia, mês e ano, não é possível obter um único timestamp: 4 de maio de 2018 corresponde a vários instantes diferentes (vários pontos na linha do tempo).

Por isso, `SimpleDateFormat` preenche as informações que faltam usando valores predefinidos. No caso das horas, este valor é meia-noite. Também é usado o timezone padrão da JVM (que no meu caso é `America/Sao_Paulo`) para saber qual é o offset usado naquele dia e horário. Juntando todas as informações, é obtido o timestamp, que é usado para construir o `Date`.

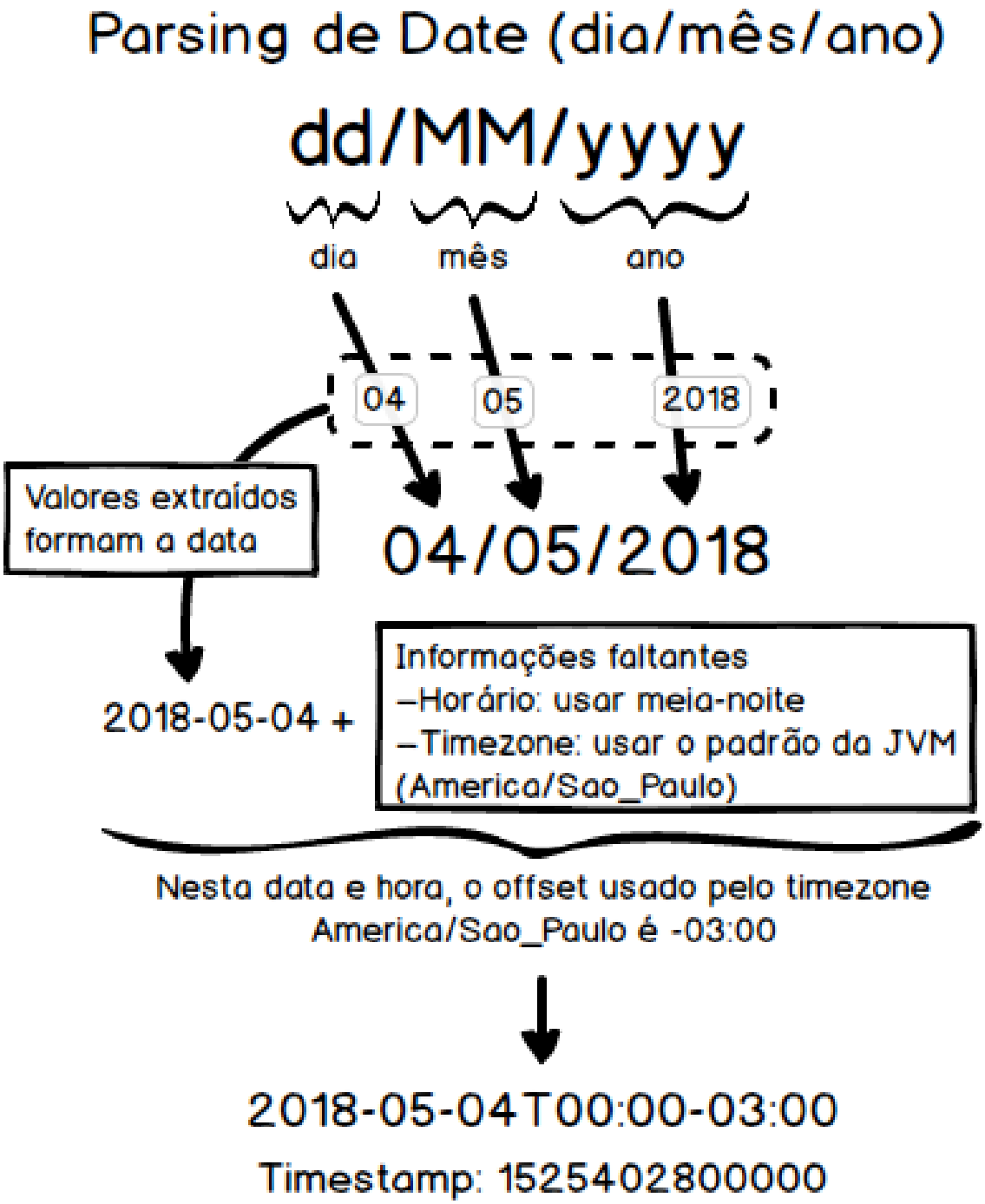

Figura 9.1: Parsing de dia, mês e ano. As informações faltantes (horas e timezone/offset) são completadas usando valores default

Caso você queira mudar o horário, basta criar um `Calendar` e usar o método `set()` , conforme já visto nos capítulos anteriores. E para não

depender do timezone padrão da JVM, que no momento do parsing pode não ser o mesmo que foi usado para gerar a `String` , basta setar o timezone desejado no `SimpleDateFormat` :

```
SimpleDateFormat parser = new SimpleDateFormat("dd/MM/yyyy");
// setar o timezone para não depender da configuração padrão
parser.setTimeZone(TimeZone.getTimeZone("America/Sao_Paulo"));
Date data = parser.parse("04/05/2018");
// criar um Calendar e usar o Date obtido pelo parsing
Calendar cal = Calendar.getInstance();
cal.setTime(data);
// mudar o horário para 17:00
cal.set(Calendar.HOUR_OF_DAY, 17);
// obter o Date com o novo horário
data = cal.getTime();
System.out.println(data);
System.out.println(data.getTime());
```

Agora a saída retorna os mesmos valores que tínhamos para o `Date` original:

```
Fri May 04 17:00:00 BRT 2018
1525464000000
```

O grande problema aqui é que, para obter o timestamp original, eu precisei saber o horário e timezone correspondentes. Caso você precise passar valores de timestamp entre sistemas, prefira passar o valor numérico (o número gigante) em vez de somente uma data. Ou então passe a informação completa: data, hora e offset (passar somente o timezone pode não ser o suficiente, pois nos casos de *overlap* você não terá como saber o offset correto).

9.2 Cuidado com datas "incompletas"

Este é um dos pontos de atenção mais importantes ao se fazer parsing de datas com `SimpleDateFormat` . Como o `Date` representa um ponto na linha do tempo (um valor de timestamp – a quantidade de

milissegundos desde o Unix Epoch), e para ter este valor precisamos da informação completa (data, hora e timezone/offset), o parsing sempre vai preencher as informações faltantes com algum valor predefinido.

Como já vimos no exemplo anterior, caso o horário não seja fornecido, será usado meia-noite (a hora, minuto, segundo e milissegundo recebem o valor zero). Se a data não for fornecida, ela é setada para 1 de janeiro de 1970. E se o timezone não for setado, será usado o padrão da JVM. Por fim, é calculado o offset, com base na data, hora e timezone obtidos.

Não ignore os timezones

Mesmo que você não esteja explicitamente usando um timezone, sempre há algum agindo "por debaixo dos panos". Como exemplo, vamos ver o que acontece em 15 de outubro de 2017:

```
SimpleDateFormat parser = new SimpleDateFormat("dd/MM/yyyy");
Date date = parser.parse("15/10/2017");
System.out.println(date);
```

O `SimpleDateFormat` está usando o timezone padrão da JVM (`America/Sao_Paulo`) e a saída deste código é:

```
Sun Oct 15 01:00:00 BRST 2017
```

O horário é 01:00. Mas nos exemplos anteriores, vimos que, quando o horário não está na `String` de entrada, o método `parse()` seta os valores para meia-noite. Por que, neste caso, a hora foi setada para 01:00? Por causa do horário de verão.

No timezone `America/Sao_Paulo` o horário de verão começou no dia 15 de outubro de 2017: à meia-noite, os relógios foram adiantados em uma hora, diretamente para 01:00. Isso significa que todos os minutos entre 00:00 e 00:59 foram "pulados" e não existem neste dia, para este timezone. Portanto, neste caso, setar o horário para

meia-noite geraria uma combinação inválida de data, hora e timezone. Então o horário é ajustado para a próxima hora válida.

Quando acaba o horário de verão, também podem ocorrer problemas. Por exemplo, se usarmos um `SimpleDateFormat` com um formato que tenha data e hora:

```
// formato com dia/mês/ano e hora:minuto
SimpleDateFormat parser = new SimpleDateFormat("dd/MM/yyyy HH:mm");
Date date = parser.parse("17/02/2018 23:00");
System.out.println(date);
```

O `parser` usa o timezone padrão da JVM (`America/Sao_Paulo`), já que nenhum foi setado. A saída deste código é:

```
Sat Feb 17 23:00:00 BRT 2018
```

O resultado foi 17 de fevereiro de 2018, às 23:00. Mas lembre-se do que acontece no timezone `America/Sao_Paulo` quando acaba o horário de verão: à meia-noite do dia 18, os relógios são atrasados em uma hora, de volta para 23:00 do dia 17. Ou seja, no dia 17, neste timezone, todos os minutos entre 23:00 e 23:59 ocorrem duas vezes. Mas o `Date` obtido pelo parsing refere-se à primeira ou segunda ocorrência das 23:00?

Uma dica é ver pela abreviação do timezone, que no caso é "BRT". Esta é abreviação de "Brasília Time" ou "Brasil Time", que é o horário oficial de Brasília, quando não é horário de verão. Quando é horário de verão, a sigla é BRST – "Brasília Summer Time".

Mas como já vimos no capítulo sobre timezones, estas abreviações não são a melhor maneira de se determinar o timezone. O ideal é imprimir também o offset usado por aquele timezone, naquele instante. Por isso, vamos usar o `SimpleDateFormat` com o formato ISO 8601, que já usamos anteriormente:

```
// parsing do formato com dia/mês/ano e hora:minuto
SimpleDateFormat parser = new SimpleDateFormat("dd/MM/yyyy HH:mm");
Date date = parser.parse("17/02/2018 23:00");
```

```
// formatar para ISO 8601
SimpleDateFormat iso8601Format = new SimpleDateFormat("yyyy-MM-
dd'T'HH:mm:ss.SSSXXX");
System.out.println(iso8601Format.format(date));
```

Note que eu usei duas instâncias diferentes de `SimpleDateFormat` : uma para fazer o parsing (transformar uma `String` em `Date`), e outra para formatar (transformar este `Date` em outra `String` , com outro formato). **É assim que se converte de um formato para outro**. Um erro bem comum é tentar usar o mesmo `SimpleDateFormat` para as duas coisas (parsing e formatação), ou então fazer o parsing com a instância que deveria ser usada para formatar e vice-versa.

A saída deste código é:

```
2018-02-17T23:00:00.000-03:00
```

Podemos ver que o offset é `-03:00` , portanto o `Date` refere-se à segunda ocorrência de 23:00 (quando não está mais em horário de verão).

9.3 Parsing usando locales

Da mesma que você pode formatar usando locales, também pode usá-los para parsing. Se eu tiver uma data com o nome do mês em inglês, devo usar o locale correspondente para fazer o parsing:

```
// formato com dia/mês/ano e locale inglês
SimpleDateFormat parser = new SimpleDateFormat("dd/MMM/yyyy",
Locale.ENGLISH);
System.out.println(parser.parse("01/Oct/2017"));
```

O nome do mês está em inglês (`Oct`). Além disso, está na forma abreviada (em vez de ter o nome completo do mês, que seria `October`), por isso, eu usei o pattern `MMM` para o mês.

A saída é:

```
Sun Oct 01 00:00:00 BRT 2017
```

Se eu não passar um `Locale` para o construtor de `SimpleDateFormat`, será usado o locale padrão da JVM. E este pode ser mudado por você, por outras aplicações rodando na mesma JVM, ou pelas equipes responsáveis pela configuração do servidor. Como é uma informação sobre a qual você não tem muito controle, o ideal é usar o locale sempre que necessário: se você sabe que a entrada vai estar em um idioma específico, não dependa da configuração padrão e use o locale correspondente.

9.4 Parsing de UTC: não ignore o Z!

Um erro bem comum é ignorar o `Z` no final de `Strings` no formato ISO 8601 (por exemplo, `2018-05-04T01:00Z`) e tratá-lo como um literal:

```
// jeito errado de se fazer parsing de UTC (Z entre aspas simples)
SimpleDateFormat iso8601Format = new SimpleDateFormat("yyyy-MM-
dd'T'HH:mm'Z'");
// 4 de maio de 2018, à 01:00 em UTC
System.out.println(iso8601Format.parse("2018-05-04T01:00Z"));
```

Note que, no pattern, o `Z` está entre aspas simples (`'Z'`) e por isso é tratado como um literal, ou seja, como a própria letra `Z`, sem nenhum significado especial. Por isso, o `SimpleDateFormat` entende que somente a data e a hora devem ter seus valores extraídos da `String`, enquanto a letra `Z` é somente um caractere sem nenhum valor a ser considerado. Por ter somente a data e a hora, o `SimpleDateFormat` usa o timezone padrão da JVM (`America/Sao_Paulo`) e verifica qual o offset válido na data e hora obtidas.

Por isso, este código não dá erro (não lança nenhuma exceção) e a saída é:

```
Fri May 04 01:00:00 BRT 2018
```

Mas só porque "funcionou", não quer dizer que esteja certo. A `String` passada para o método `parse()` corresponde a `2018-05-04T01:00Z` (4 de maio de 2018, à 01:00 **em UTC**), mas repare que o `Date` retornado corresponde a 4 de maio de 2018, à 01:00, **no horário de Brasília** (BRT), ou seja, `2018-05-04T01:00-03:00` . Convertendo esta data para UTC, o resultado é `2018-05-04T04:00Z` (3 horas de diferença do valor original).

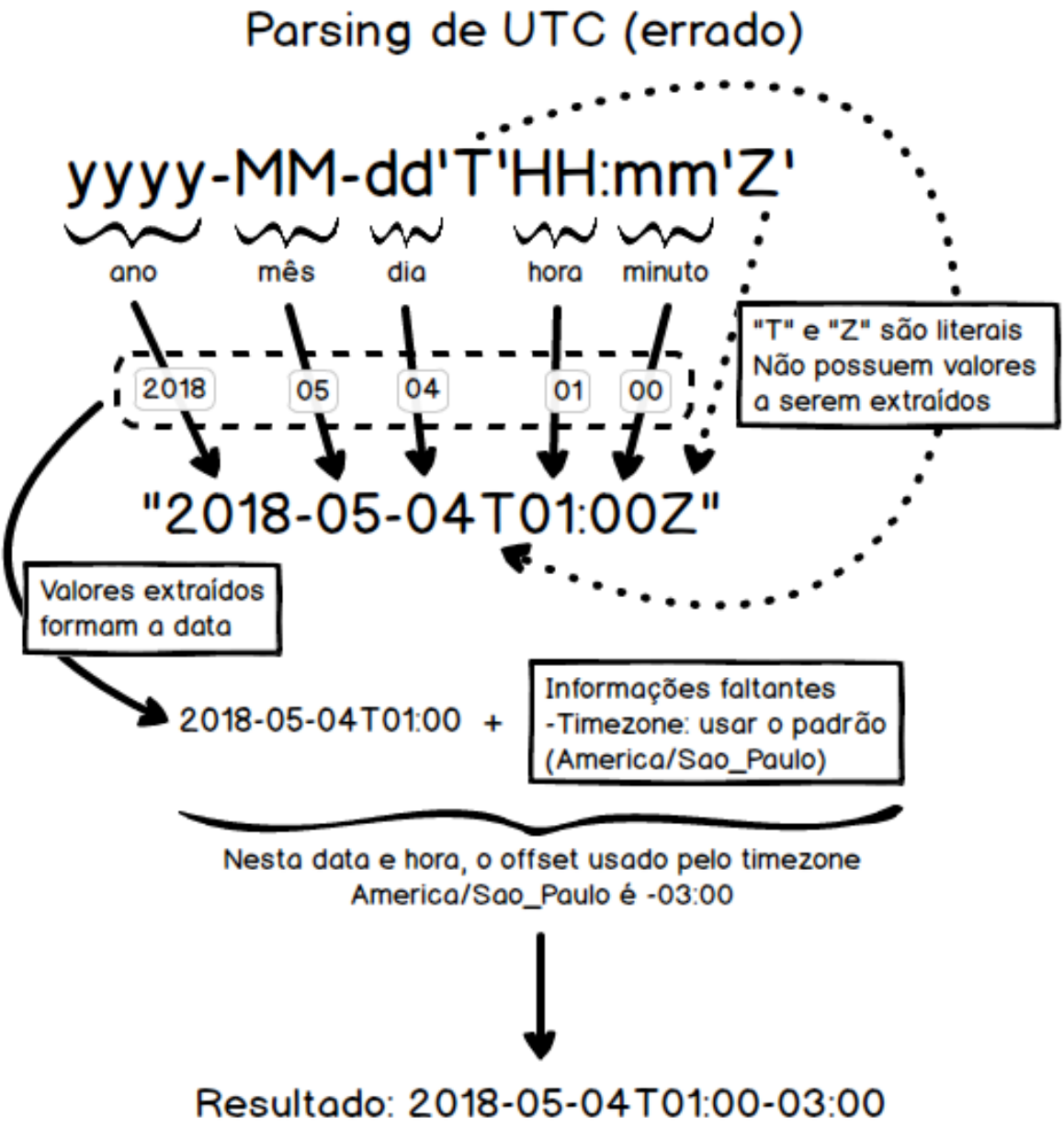

Figura 9.2: Tratar "Z" como literal traz um resultado errado

Não confunda um Z com outro

Tudo bem, já vimos que o `z` não pode ser um literal. E é aí que surge outro erro bem comum: achar que retirar o `z` das aspas simples resolve o problema.

```
SimpleDateFormat iso8601Format = new SimpleDateFormat("yyyy-MM-
dd'T'HH:mmZ");
String input = "2018-05-04T01:00Z";
System.out.println(iso8601Format.parse(input));
```

Mas este código lança uma exceção:

```
java.text.ParseException: Unparseable date: "2018-05-04T01:00Z"
```

Isso acontece porque o pattern `Z` representa um offset, e sempre é impresso sem os dois pontos (exemplos: `-0300` , `+0530` , `+0000`). Por isso, o offset zero só é reconhecido por este pattern se o valor for `+0000` . Se estiver em qualquer outro formato (como `Z` ou `+00:00`), é lançada uma exceção.

Não confunda o `Z` que é usado no pattern (no construtor de `SimpleDateFormat`) com o `Z` que vai no final de uma `String` que contém uma data no formato ISO 8601.

Onde o `Z` é usado	O que significa
Pattern (construtor de `SimpleDateFormat`) – exemplo: `new SimpleDateFormat("yyyy-MM-dd'T'HH:mmZ")`	Offset sem dois-pontos – exemplos: `-0300` , `+0530` , `+0000`
Final de `String` no formato ISO 8601 – exemplo: `2018-05-04T01:00Z`	Data/hora está em UTC – o offset é zero

Então qual o jeito certo de fazer este parsing?

Para que possamos obter o valor correto, a letra `Z` deve ser interpretada como o que ela de fato representa. Nos capítulos anteriores vimos que o offset zero, além de ser escrito como `+00:00` , é mais comumente representado pela letra `Z` . Ou seja, o `Z` no final da `String` é um offset. E o pattern usado para offsets, como já vimos anteriormente, é a letra `X` maiúscula. Então devemos remover o `'Z'` do pattern e trocar por `X` :

```
SimpleDateFormat iso8601Format = new SimpleDateFormat("yyyy-MM-
dd'T'HH:mmXXX");
String input = "2018-05-04T01:00Z";
System.out.println(iso8601Format.parse(input));
```

Eu usei `XXX` para aceitar tanto offsets com as horas e minutos quanto o `Z` . A saída é:

```
Thu May 03 22:00:00 BRT 2018
```

3 de maio de 2018, às 22:00, BRT (Horário de Brasília). Sim, o dia é diferente da `String` de entrada, já que ela está em UTC, mas, ao imprimir o `Date` , ele usa o timezone padrão da JVM (`America/Sao_Paulo`). A `String` de entrada corresponde a `2018-05-04T01:00Z` (4 de maio de 2018, 01:00 em UTC), e este valor, convertido para o timezone `America/Sao_Paulo` (cujo offset em maio de 2018 é `-03:00`), é igual a `2018-05-03T22:00-03:00` .

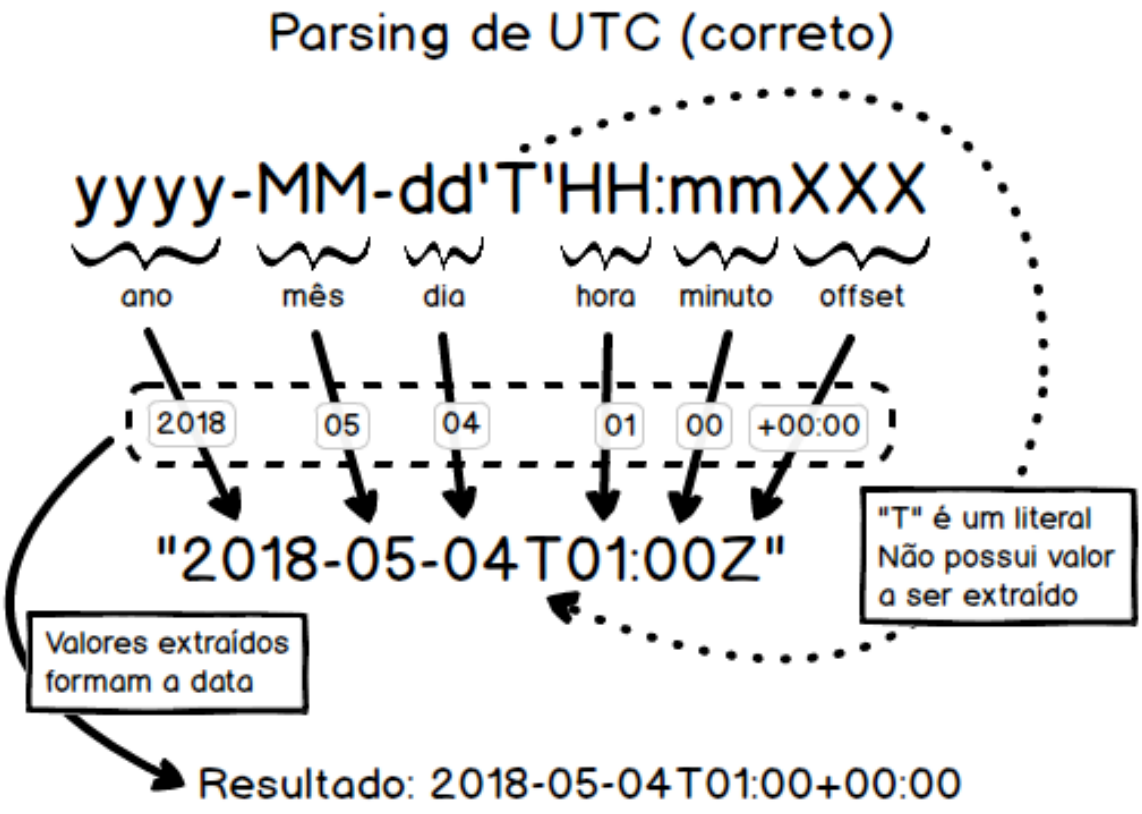

Figura 9.3: Tratar "Z" como offset traz o resultado correto

Na documentação, o pattern `x` é descrito como "ISO 8601 Time zone". Mas já vimos que a ISO 8601 aceita somente offsets, e não há suporte para os timezones de fato (os identificadores da IANA, como `America/Sao_Paulo`). A documentação de `SimpleDateFormat` é um dos muitos casos de APIs, linguagens e frameworks que confundem os conceitos de offset e timezone. Isoladamente, este fato não torna a API ruim, porém não é algo benéfico para o desenvolvedor, pois ajuda a disseminar uma ideia errada sobre conceitos que já não são muito bem compreendidos.

9.5 Com SimpleDateFormat pode (quase) tudo

A classe `SimpleDateFormat` é conhecida por ser bem problemática, e você vai encontrar vários artigos na internet explicando em detalhes cada um destes problemas (deixei alguns links no final do capítulo, como referência). Já vimos alguns deles, como o fato de sempre ter um timezone associado e isso poder causar problemas quando estamos lidando com dados incompletos (somente a data ou a hora, por exemplo). Agora veremos mais algumas características desta classe, os problemas que isto pode causar e como evitá-los ou minimizá-los.

Validação e o comportamento leniente

Uma situação bem comum é quando queremos não apenas fazer o parsing de uma `String` , mas também garantir que ela não é uma data inválida. Vamos ver como o parsing se comporta caso a `String` represente um dia inválido, como 31 de fevereiro:

```
SimpleDateFormat parser = new SimpleDateFormat("dd/MM/yyyy");
// 31 de fevereiro de 2018
System.out.println(parser.parse("31/02/2018"));
```

A saída é:

```
Sat Mar 03 00:00:00 BRT 2018
```

O resultado foi 3 de março. Isso acontece porque o comportamento padrão de `SimpleDateFormat` é "permitir tudo". Não importa qual a `String` recebida, ele tenta dar um jeito de fazer o parsing. Este comportamento é chamado de "leniente" (o mesmo que vimos anteriormente com `Calendar`).

No caso do código anterior, o parsing usa um "raciocínio" mais ou menos assim:

- em 2018, fevereiro tem 28 dias;
- 31 são 3 dias a mais do que 28;
- então eu devo pegar 3 dias depois de 28 de fevereiro de 2018, ou seja, 3 de março de 2018.

Se este ajuste é a coisa certa a se fazer, é uma questão de opinião. O fato é que isso acaba pegando muitos desenvolvedores de surpresa. Este código aceita inclusive `Strings` como `99/02/2018` (usando o mesmo "raciocínio", resulta em 10 de maio de 2018). E até mesmo valores como `100/02/2018` são aceitos (resultando em 11 de maio de 2018) — sim, mesmo que o pattern tenha apenas duas letras `d` para o dia, são aceitos 3 dígitos na `String` .

Para evitar isso e fazer com que somente datas válidas sejam aceitas, você deve cancelar o comportamento leniente, usando o método `setLenient()` :

```
SimpleDateFormat parser = new SimpleDateFormat("dd/MM/yyyy");
// cancelar o comportamento leniente
parser.setLenient(false);
// 31 de fevereiro de 2018
Date date = parser.parse("31/02/2018");
```

Agora o código lança uma exceção para a data inválida:

```
java.text.ParseException: Unparseable date: "31/02/2018"
```

Usando `setLenient(true)` , o comportamento leniente é ativado novamente.

Leniente até demais

`SimpleDateFormat` é tão leniente que às vezes nem valida o formato. No próximo exemplo, eu uso o pattern `yyyyMMdd` (ano, mês e dia, sem separadores) e passo uma `String` em um formato diferente para o método `parse()` :

```
// formato ano, mês e dia, sem separadores
SimpleDateFormat parser = new SimpleDateFormat("yyyyMMdd");
// 1 de fevereiro de 2018, em formato ISO 8601
System.out.println(parser.parse("2018-02-01"));
```

O resultado é:

```
Sat Dec 02 00:00:00 BRST 2017
```

Como a `String` está em um formato diferente do pattern, o esperado seria uma exceção. Mas `SimpleDateFormat` é tão leniente que o parsing foi feito com sucesso (*"não deu erro, então deu certo, né?"*). Só que o resultado está completamente errado. A `String` de entrada corresponde a 1 de fevereiro de 2018, mas o método `parse()` retornou um `Date` que corresponde a **2 de dezembro de 2017**! O que aconteceu?

Por ser leniente, `SimpleDateFormat` tenta de tudo para fazer o parsing e só lança exceção se não tiver jeito mesmo. Neste caso, ao fazer a correspondência do pattern `yyyyMMdd` com a entrada `2018-02-01` , acabamos com os seguintes valores:

```
yyyyMMdd    <-- pattern
2018-02-01  <-- String de entrada

ano=2018
mês=-0
dia=2 (o "-" depois do "2" é descartado)

O restante da String ("01") é ignorado
```

Tendo estes valores, o parsing faz os devidos ajustes. O valor do mês é `-0` , que é o mesmo que `0` . E o que significa "mês zero"? Se

janeiro é o mês 1 e o ano é 2018, então zero seria o mês anterior (dezembro de 2017). O dia 2 está "certo" (quer dizer, é um dia válido para dezembro de 2017) e não precisa de ajustes. Então o resultado final é 2 de dezembro de 2017.

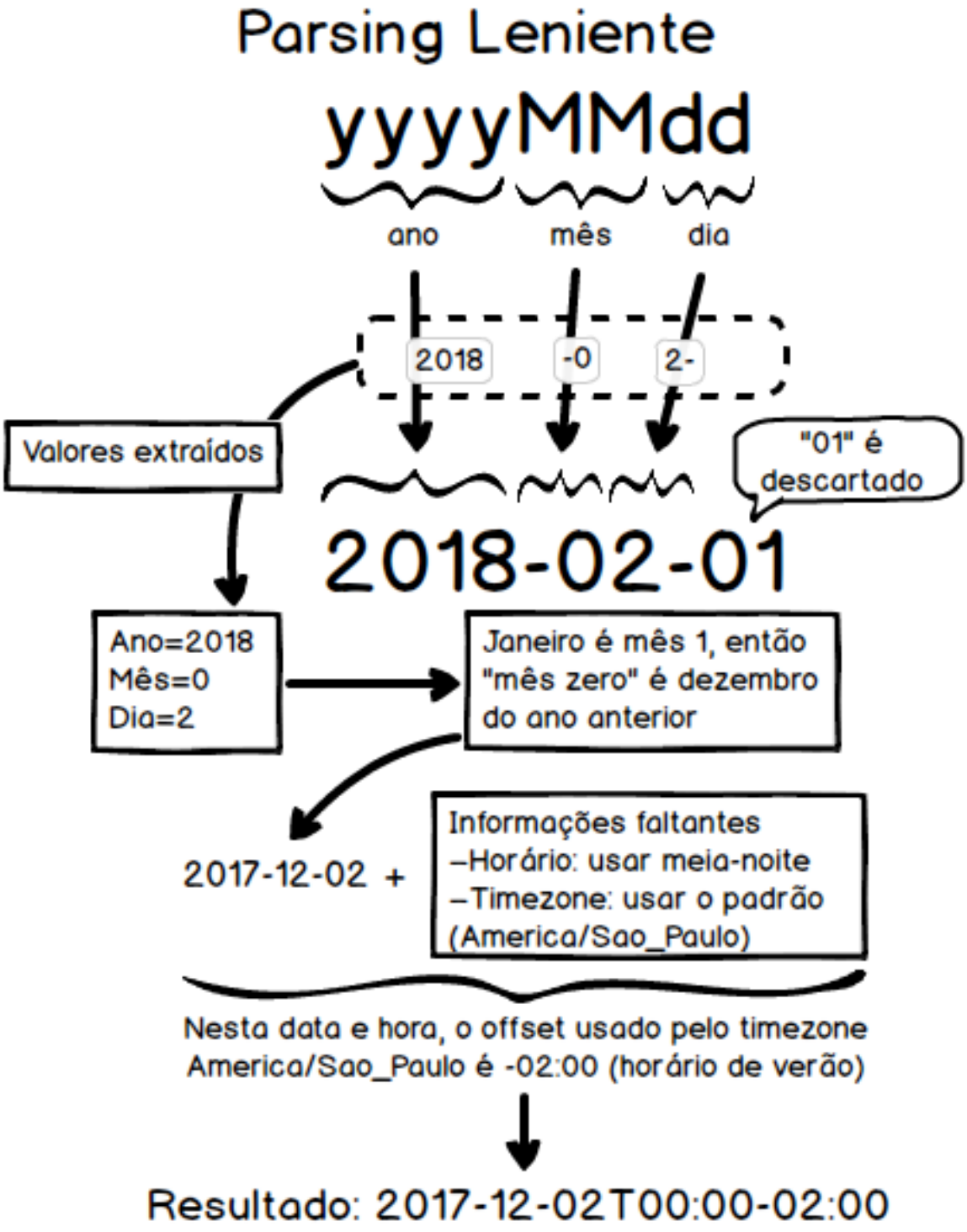

Figura 9.4: O parsing leniente pode trazer resultados incorretos e inesperados

Esta situação também pode ser evitada com o uso de `setLenient(false)` , que lança um `ParseException` caso a `String` não esteja no mesmo formato do pattern.

Funciona quando não devia

Um comportamento de `SimpleDateFormat` que pode ser perigoso é o fato do parsing ignorar caracteres excedentes, o que pode nos levar a ignorar partes da `String` sem nem perceber, como mostra o próximo exemplo:

```
// formato dia/mês/ano
SimpleDateFormat parser = new SimpleDateFormat("dd/MM/yyyy");
parser.setLenient(false);
// 1 de fevereiro de 2018, às 10:30
System.out.println(parser.parse("01/02/2018 10:30"));
```

O pattern usado no `SimpleDateFormat` só tem dia, mês e ano. Mas a `String` passada para o método `parse()` possui data e hora. Mesmo com o modo leniente desligado, foi feito o parsing da data, pois `01/02/2018` satisfaz o pattern `dd/MM/yyyy` , e o restante da `String` , contendo as horas (`10:30`), é ignorado.

Com isso, não é feito o parsing do horário (`10:30`), e o método `parse()` age como se ele não existisse. Ou seja, o horário é setado para meia-noite. O resultado é:

```
Thu Feb 01 00:00:00 BRST 2018
```

Mesmo deixando claro que eu não quero o comportamento leniente, `SimpleDateFormat` não detecta que a `String` não está de acordo com o formato especificado pelo pattern e, em vez de uma exceção (que seria o esperado), eu tenho um `Date` que corresponde ao horário errado.

9.6 Parsing de frações de segundo

`Date` e `Calendar` possuem precisão de milissegundos — 3 casas decimais nas frações de segundo. Isso quer dizer que elas trabalham bem com valores como `02:30:22.385` (2 horas, 30 minutos, 22 segundos e 385 milésimos de segundo). As frações de segundo (`385`) podem ter no máximo 3 dígitos, com valores de 0 a 999. Este é o limite desta API, e qualquer fração com mais que 3 casas decimais não é suportada, portanto, valores como `02:30:22.3856` (4 casas decimais) não podem ser representados corretamente.

Atualmente já existem várias APIs e bancos de dados que trabalham com uma precisão maior. Alguns suportam microssegundos (6 casas decimais) e outros já estão nos nanossegundos (9 casas decimais). E o problema surge quando tentamos usar estes valores com uma precisão maior do que a API suporta.

Como `SimpleDateFormat` trabalha diretamente com `Date` , também possui o limite de 3 casas decimais. O grande problema é que, devido ao comportamento leniente, esta classe aceita mais do que 3 dígitos para fazer o parsing. Vamos ver um exemplo com uma `String` em formato ISO 8601 e usando microssegundos (6 casas decimais):

```
// pattern com 6 dígitos na fração de segundos
SimpleDateFormat parser = new SimpleDateFormat("yyyy-MM-
dd'T'HH:mm:ss.SSSSSS");
System.out.println(parser.parse("2018-02-01T10:20:30.123456"));
```

A saída é:

```
Thu Feb 01 10:22:33 BRST 2018
```

A data está correta (1 de fevereiro de 2018), mas a hora não. Na `String` de entrada o horário é `10:20:30` , enquanto no `Date` , corresponde a `10:22:33` . O que aconteceu?

Ao fazer a correspondência da `String` com o pattern, os valores do ano, mês, dia e demais campos são obtidos pelo parsing. Ou seja, o ano é `2018` , o mês é `02` etc. Tudo funciona bem, até chegarmos aos milissegundos.

O valor `123456` corresponde ao pattern `SSSSSS` . A letra `S` maiúscula representa os milissegundos, e como eu usei seis letras, o parsing verifica se há 6 dígitos que correspondem a elas. Por isso, o valor obtido é `123456` .

Só que 123456 milissegundos correspondem a "2 minutos, 3 segundos e 456 milissegundos". E este valor é somado ao horário que já havia sido obtido anteriormente (`10:20:30`). Por isso, o resultado é `10:22:33.456` .

Na saída, nós não vimos o `.456` porque o método `toString()` da classe `Date` omite os milissegundos. Que tal então se usarmos o mesmo `SimpleDateFormat` para formatar a data?

```
// pattern com 6 dígitos na fração de segundos
SimpleDateFormat parser = new SimpleDateFormat("yyyy-MM-
dd'T'HH:mm:ss.SSSSSS");
// parsing da data
Date date = parser.parse("2018-02-01T10:20:30.123456");
// formatar o Date usando o mesmo SimpleDateFormat (com 6 casas decimais)
System.out.println(parser.format(date));
```

A saída é:

```
2018-02-01T10:22:33.000456
```

As frações de segundo estão erradas, pois `.000456` corresponde a 456 **microssegundos** (o correto seria mostrar `.456` , que representa 456 milissegundos). Isso acontece porque o pattern possui 6 letras `S` , mas como o valor correspondente só possui 3 dígitos (`456`), a saída é preenchida com zeros à esquerda. Este comportamento é descrito na documentação (https://docs.oracle.com/javase/8/docs/api/java/text/SimpleDateFormat.html#number/).

Ou seja, ao usar mais que 3 casas decimais nas frações de segundo (mais que 3 letras `S` no pattern), você obtém valores errados tanto no parsing quanto na formatação.

Mas o grande problema, na minha opinião, é o fato de o parsing não lançar nenhuma exceção e retornar um `Date` como se nada tivesse acontecido. Só que este `Date` possui um valor incorreto, e você só vai perceber quando for tarde demais. Assim como nos exemplos anteriores, isso pode ser evitado usando `setLenient(false)` , que lança um `ParseException` caso a entrada tenha mais que 3 casas decimais.

Então como eu faço parsing com mais de 3 casas decimais?

Você não faz. Pelo menos, não com `SimpleDateFormat` . Provavelmente você já viu algum artigo na internet dizendo para simplesmente usar várias letras `S` . E talvez você tenha feito isso, afinal *"não deu erro, então funcionou!"*, não é mesmo?

O uso de mais que 3 letras `S` no pattern só funciona se todos os dígitos forem zero. Se a entrada for `2018-02-01T10:20:30.000000` , os 6 últimos dígitos serão interpretados como zero milissegundos, e por isso não afetará o resultado final. Mas é o único caso em que isso "funciona".

Qualquer outro valor gerará uma data errada. Mesmo se o valor tiver vários zeros no final, como `.100000` . Isso será interpretado como 100000 (cem mil) milissegundos (que corresponde a 1 minuto e 40 segundos) e isso será somado ao resultado.

Se a `String` de entrada tem mais que 3 casas decimais, a solução é deixar apenas os 3 primeiros dígitos, removendo os demais. Caso você não queira perder a precisão, pode guardar os dígitos excedentes em um campo separado.

A outra alternativa, claro, é usar a API `java.time` , que possui precisão de nanossegundos (9 casas decimais) e que será vista na terceira parte do livro.

> *Na verdade*, existe uma outra alternativa, usando `java.sql.Timestamp`, e que será explicada posteriormente no capítulo sobre o pacote `java.sql`.

9.7 Usando SimpleDateFormat em ambientes multithread

Uma característica de `SimpleDateFormat` é que esta classe não é *thread-safe*. Isso quer dizer que seu uso em ambientes multithread exige alguns controles adicionais.

Para mostrar isso, vamos fazer um pequeno teste. O objetivo não é explicar em detalhes como funcionam threads, mas sim dar uma pequena ideia dos tipos de problemas que podem ocorrer.

Primeiro, vamos criar um `SimpleDateFormat` estático que será compartilhado entre várias threads e um `java.util.concurrent.ExecutorService`, que será o responsável por rodar estas threads:

```
static SimpleDateFormat SDF = new SimpleDateFormat("dd/MM/yyyy");

ExecutorService pool = Executors.newCachedThreadPool();
```

Então, eu uso o `ExecutorService` para rodar 100 threads que fazem a mesma coisa: primeiro usam o `SimpleDateFormat` para fazer parsing de uma `String`, gerando um `Date`. Logo em seguida, este `Date` é formatado, gerando outra `String`. Caso as duas `Strings` sejam diferentes, eu imprimo ambas para que possamos comparar as diferenças. Caso uma exceção seja lançada, ela também é impressa, para sabermos quantos erros ocorreram:

```
String entrada = "01/02/2018";
// criar 100 threads
```

```
for (int i = 0; i < 100; i++) {
    // cada thread faz parsing da String e em seguida formata
    pool.submit(() -> {
        try {
            Date date = SDF.parse(entrada);
            String dataFormatada = SDF.format(date);
            // se as Strings forem diferentes, imprime ambas
            if (! entrada.equals(dataFormatada)) {
                System.out.println(entrada + " diferente de " +
dataFormatada);
            }
        } catch (Exception e) { // imprimir exceções
            System.out.println(e);
        }
    });
}
```

Como o comportamento de threads não é determinístico, a saída vai variar a cada execução. De maneira geral, em várias threads vai funcionar como esperado: o resultado de `format()` será o mesmo da `String` original e nada será impresso. Porém, também teremos casos em que as `Strings` serão diferentes e ambas serão impressas. Alguns casos que aconteceram quando eu testei este código:

```
01/02/2018 diferente de 31/01/2018
01/02/2018 diferente de 12/05/2018
01/02/2018 diferente de 11/08/2191
01/02/2018 diferente de 01/02/0001
```

Repare como os valores obtidos são bem diferentes da `String` original. Isso significa que, mesmo não lançando exceção, o `Date` retornado pode não ser o que você espera. E também teremos casos em que uma exceção é lançada. Quando eu rodei, aconteceram vários erros diferentes:

```
java.lang.ArrayIndexOutOfBoundsException: -1
java.lang.NumberFormatException: empty String
java.lang.NumberFormatException: For input string: ""
java.lang.NumberFormatException: For input string: "E.119714E1"
java.lang.NumberFormatException: multiple points
```

Isso acontece porque a implementação de `SimpleDateFormat` guarda resultados intermediários do parsing em variáveis de instância. Internamente, ela possui um campo `Calendar` , que é usado para guardar estes resultados, à medida que os campos são interpretados. E este mesmo `Calendar` também é usado para formatar.

De forma resumida, o que pode acontecer é: enquanto uma thread está no meio do parsing, com o `Calendar` contendo vários valores intermediários obtidos da `String` de entrada, outra thread pode estar formatando, e acabar imprimindo estes valores. Dependendo do estado em que se encontra o `Calendar` , estas operações podem lançar uma exceção.

Soluções

Para compartilhar uma mesma instância entre várias threads, uma solução é usar um bloco `synchronized` :

```
synchronized (SDF) {
    Date date = SDF.parse(entrada);
    String dataFormatada = SDF.format(date);
    // se as Strings forem diferentes, imprime ambas
    if (! entrada.equals(dataFormatada)) {
        System.out.println(entrada + " diferente de " + dataFormatada);
    }
}
```

Com isso, os erros não ocorrem mais, pois todo o código dentro do bloco `synchronized` só pode ser executado por uma thread de cada vez. Só que isto causa contenção: enquanto a thread não sai do bloco `synchronized` , todas as outras ficam esperando a sua vez.

Para evitar a contenção, basta não compartilhar o `SimpleDateFormat` entre várias threads, criando uma nova instância a cada iteração do loop. Portanto, serão criadas 100 instâncias, que não serão compartilhadas entre as threads, acabando com o problema.

```
for (int i = 0; i < 100; i++) {
    pool.submit(() -> {
        // cada iteração do loop cria uma nova instância
        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
        ... o restante do código é igual
```

Em vez de criar uma nova instância com `new`, uma prática bem comum é clonar o `SimpleDateFormat` original, usando o método `clone()`. Esta técnica é conhecida como **cópia defensiva**:

```
// usar clone() ao invés de new
SimpleDateFormat sdf = (SimpleDateFormat) SDF.clone();
```

Assim, temos uma cópia do `SimpleDateFormat`, que será independente do original e, portanto, não será compartilhada entre threads. Só que continuamos criando uma instância nova a cada iteração.

Mas temos como otimizar a criação de instâncias. Como eu usei `Executors.newCachedThreadPool()`, as threads que terminam sua execução são reaproveitadas nas próximas iterações, o que significa que serão necessárias bem menos que 100 threads para executar o loop. E é possível fazer com que cada thread crie apenas um `SimpleDateFormat`, usando um `java.lang.ThreadLocal`. Com isso, o `SimpleDateFormat` pertence a somente uma thread, garantindo que não haverá problemas de concorrência.

Para usar um `ThreadLocal`, basta sobrescrever o método `initialValue()`, de forma que ele retorne um novo `SimpleDateFormat`, que será a instância usada pela thread. Depois, quando uma thread precisar desta instância, basta chamar o método `get()`.

```
static final ThreadLocal<SimpleDateFormat> TL = new
ThreadLocal<SimpleDateFormat>() {
    protected SimpleDateFormat initialValue() {
        return new SimpleDateFormat("dd/MM/yyyy");
    }
};
```

E dentro de cada thread, eu uso este `ThreadLocal` para obter o `SimpleDateFormat`. Com isso, as threads que são reaproveitadas não

precisam criar uma nova instância:

```
pool.submit(() -> {
    // obter a instância de SimpleDateFormat do ThreadLocal
    SimpleDateFormat sdf = TL.get();
    ... o restante do código é igual
```

`Date` e `Calendar` também sofrem do mesmo problema: não são *thread safe* e podem ter problemas se uma instância for compartilhada entre várias threads. E as técnicas para evitar esses problemas são as mesmas usadas com `SimpleDateFormat` : sincronização e clonagem.

Referências

Se você procurar na internet por "*simpledateformat problems*" ou qualquer outra frase parecida, vai encontrar dezenas de artigos, alguns deles bem raivosos, explicando em detalhes os problemas desta classe. Nós vimos os principais, e acredito que cobrem várias situações que você vai encontrar no dia a dia.

Deixo aqui alguns destes artigos interessantes, que contêm inclusive código, para que você se aprofunde mais no assunto:

- Um resumo dos principais problemas: timezones, validação e thread-safety (https://eyalsch.wordpress.com/2009/05/29/sdf/)
- Artigo com códigos bem detalhados sobre o comportamento leniente e como usar `SimpleDateFormat` em ambientes multi-thread (https://www.javaspecialists.eu/archive/Issue172.html). Alguns códigos de exemplo que usei foram inspirados neste artigo.
- Perguntas do Stack Overflow sobre thread-safety (https://stackoverflow.com/q/6840803) e (https://stackoverflow.com/q/4021151)
- Outra do Stack Overflow, sobre fazer parsing com mais de 3 dígitos nas frações de segundos (https://stackoverflow.com/a/12000749)

Mas chega de criticar `SimpleDateFormat` . No próximo capítulo veremos como `Calendar` implementa a aritmética de datas.

Capítulo 10
Aritmética de datas com Date e Calendar

Na primeira parte do livro, vimos o funcionamento básico da aritmética de datas e como ela pode ser confusa e não intuitiva. Neste capítulo, veremos como fazer todas aquelas contas usando `Date` e `Calendar`.

10.1 Como somar 2 horas a um Date?

Esta classe não possui métodos para somar ou subtrair uma duração. Os únicos métodos que mudam informações do `Date` são *setters*:

- Há métodos para mudar campos específicos, como `setYear()` e `setMinutes()`. Mas todos estão *deprecated* e recomendam usar `Calendar.set()` para mudar o respectivo campo.
- O único *setter* que não está *deprecated* é `setTime()`, que muda o valor do timestamp.

E só. Como não há métodos específicos para somar uma quantidade de tempo, uma prática muito comum é somar esta quantidade ao timestamp e, em seguida, passar o resultado para `setTime()`.

O timestamp representa a quantidade de milissegundos decorridos desde o Unix Epoch. Para somar 2 horas, eu preciso transformar esta duração no valor equivalente em milissegundos, e depois somar este valor ao timestamp.

O cálculo é relativamente simples: 2 horas são 60 minutos, cada minuto tem 60 segundos e cada segundo tem 1000 milissegundos, então basta fazer várias multiplicações para obter o valor a ser

somado. Mas você também pode usar `java.util.concurrent.TimeUnit`, que possui o método `convert()` para facilitar estas contas. O código fica assim:

```
// data atual: 2018-05-04T17:00-03:00 (America/Sao_Paulo)
Date dataAtual = new Date();
System.out.println(dataAtual);
// 2 horas em milissegundos
long duasHorasEmMs = TimeUnit.MILLISECONDS.convert(2, TimeUnit.HOURS);
// somar 2 horas ao valor do timestamp
dataAtual.setTime(dataAtual.getTime() + duasHorasEmMs);
System.out.println(dataAtual);
```

A saída mostra que, de fato, as 2 horas foram adicionadas:

```
Fri May 04 17:00:00 BRT 2018
Fri May 04 19:00:00 BRT 2018
```

Apesar de um pouco trabalhoso, parece que deu certo, não? Mas se tudo fosse tão simples assim, não precisaria de um capítulo inteiro só para isso. Vamos aos casos mais complicados.

10.2 Prefira usar Calendar

No capítulo sobre aritmética de datas vimos que operações com dias, meses e anos são complicadas porque estas unidades não têm tamanho fixo.

Graças aos *gaps* e *overlaps*, nem sempre o dia tem 24 horas. Um mês pode ter 28, 29, 30 ou 31 dias. E um ano pode ter 365 ou 366 dias. E não se esqueça da semântica das operações:

- Somar 1 dia deveria resultar no mesmo horário do dia seguinte. Fatores como o horário de verão fazem com que isso nem sempre seja igual a somar 24 horas.
- Somar 1 mês deveria resultar no mesmo dia do mês seguinte. Como meses não têm tamanho fixo, a quantidade de dias a ser

adicionada não é a mesma para todos os casos.

Usando `Date` e seu método `setTime()` , não há um jeito fácil de resolver estas questões. Várias premissas terão que ser assumidas, como considerar um mês com 30 dias (ou qualquer outro valor), e então ajustar o resultado manualmente, caso necessário, fazendo vários arredondamentos arbitrários e longe de serem precisos.

Mas isso tudo é complicado demais – tentar fazer esses cálculos manualmente é querer **muito** reinventar a roda – e uma alternativa um pouco melhor é usar `Calendar` . Esta classe possui o método `add()` , que é usado para somar durações a uma data. Este método recebe 2 parâmetros: o campo a ser adicionado (que são as mesmas constantes usadas com `get()` e `set()` , como `Calendar.MINUTE` e `Calendar.MONTH`), e a quantidade a ser adicionada. O exemplo anterior (somar 2 horas) ficaria assim:

```
// data atual: 2018-05-04T17:00-03:00 (America/Sao_Paulo)
Calendar dataAtual = Calendar.getInstance();
System.out.println(dataAtual.getTime());
// somar 2 horas
dataAtual.add(Calendar.HOUR_OF_DAY, 2);
System.out.println(dataAtual.getTime());
```

O resultado é o mesmo do exemplo anterior, porém o código ficou um pouco melhor, pois não precisamos fazer os cálculos para converter 2 horas em milissegundos. A saída é a mesma:

```
Fri May 04 17:00:00 BRT 2018
Fri May 04 19:00:00 BRT 2018
```

Cuidado, sempre há um timezone em ação

`Calendar` possui um timezone, então ao usar `add()` as regras de horário de verão são levadas em consideração. Todos os cálculos complicados são feitos internamente e a semântica das operações é mantida.

No próximo exemplo, vamos usar um `Calendar` e setá-lo para `2017-10-14T10:00-03:00` (14 de outubro, um dia antes de o horário de verão começar em São Paulo). A seguir, somamos 1 dia, resultando no dia 15 às 10:00 (mesmo horário do dia seguinte). Também computamos os valores dos timestamps antes e depois, e calculamos a diferença em horas:

```
// Um dia antes do horário de verão: 2017-10-14T10:00-03:00
(America/Sao_Paulo)
Calendar cal = Calendar.getInstance();
cal.set(2017, Calendar.OCTOBER, 14, 10, 0, 0);
cal.set(Calendar.MILLISECOND, 0);
// antes de somar 1 dia
long timestampAntes = cal.getTimeInMillis();
System.out.println("antes: " + cal.getTime());
// somar 1 dia
cal.add(Calendar.DAY_OF_MONTH, 1);
long timestampDepois = cal.getTimeInMillis();
System.out.println("depois:" + cal.getTime());
// calcula a diferenca em horas
long diferenca = timestampDepois - timestampAntes;
// a diferenca está em milissegundos, converter para horas
System.out.println(TimeUnit.HOURS.convert(diferenca,
TimeUnit.MILLISECONDS));
```

A saída mostra a data antes e depois de somar 1 dia, e a diferença entre elas, em horas:

```
antes: Sat Oct 14 10:00:00 BRT 2017
depois:Sun Oct 15 10:00:00 BRST 2017
23
```

Podemos ver que a semântica da operação "somar 1 dia" foi mantida. O resultado foi o dia seguinte, no mesmo horário. Mas graças ao horário de verão, o ajuste de offsets faz com que a diferença entre as datas seja de 23 horas.

Se usássemos `Date.setTime()` , esses ajustes teriam que ser feitos manualmente. Você teria que identificar que houve uma mudança de horário de verão e verificar os offsets antes e depois, para saber a

quantidade exata de horas a ser somada. Não é algo impossível de ser feito (dá para usar o método `getOffset()` da classe `TimeZone` , que retorna o offset usado em um determinado instante), mas é melhor usar `Calendar` , que já faz estas contas para você.

Somar um mês

Ao somar um mês, o resultado é o mesmo dia no mês seguinte. No próximo exemplo, eu seto um `Calendar` para 1 de janeiro e, ao somar um mês, o resultado é 1 de fevereiro.

```
Calendar cal = Calendar.getInstance();
// 1 de janeiro de 2018
cal.set(2018, Calendar.JANUARY, 1);
// somar 1 mês
cal.add(Calendar.MONTH, 1);
// 1 de fevereiro
System.out.println(cal.getTime());
```

Eu usei o método `set()` que recebe ano, mês e dia, portanto o horário atual do `Calendar` é mantido (17:00). A saída é:

```
Wed Feb 28 17:00:00 BRT 2018
```

Mas e se a data for 31 de janeiro?

```
Calendar cal = Calendar.getInstance();
// 31 de janeiro de 2018
cal.set(2018, Calendar.JANUARY, 31);
// somar 1 mês
cal.add(Calendar.MONTH, 1);
System.out.println(cal.getTime());
```

O resultado seria 31 de fevereiro, que é uma data inválida. Como estou somando um mês, a API entende que o resultado deveria estar no mês seguinte, ou seja, em fevereiro (mantendo-se a semântica da operação "somar um mês"). Por isso, o dia é ajustado para o maior válido naquele mês. No caso, o resultado é 28 de fevereiro:

```
Wed Feb 28 17:00:00 BRT 2018
```

E para subtrair?

Para subtrair uma duração, não existe um método `subtract` , mas basta somar um valor negativo. Exemplo: para subtrair 2 horas, faça `add(Calendar.HOUR_OF_DAY, -2)` .

Todas as recomendações vistas anteriormente continuam valendo: atente-se para a semântica das operações e para os resultados afetados por timezones. As mesmas situações estranhas que podem ocorrer ao somar durações também podem acontecer ao subtraí-las.

E a aritmética de datas é tão contraintuitiva que a subtração nem sempre é o inverso da soma.

No exemplo anterior, vimos que somar 1 mês a 31 de janeiro de 2018 resulta em 28 de fevereiro. Mas ao subtrair 1 mês de 28 de fevereiro, o resultado é 28 de janeiro (o mesmo dia do mês anterior). O mesmo acontece para os dias 28, 29 e 30 de janeiro, conforme podemos ver no exemplo a seguir:

```
Calendar cal = Calendar.getInstance();
// testar dos dias 28 a 31 de janeiro
for (int dia = 28; dia <= 31; dia++) {
    cal.set(2018, Calendar.JANUARY, dia);
    // somar 1 mês
    cal.add(Calendar.MONTH, 1);
    System.out.println(cal.getTime()); // 28 de fevereiro
    // subtrair 1 mês
    cal.add(Calendar.MONTH, -1);
    System.out.println(cal.getTime()); // 28 de janeiro
}
```

Para os dias 28 a 31 de janeiro, ocorre o seguinte: ao somar 1 mês, o resultado é 28 de fevereiro. E ao subtrair 1 mês, o resultado é 28 de janeiro. Isso acontece porque ao subtrair 1 mês de 28 de fevereiro, a API não tem como saber se o valor original (o que teve 1

mês somado) era 28, 29, 30 ou 31 de janeiro. E por isso ela opta por simplesmente usar o mesmo dia, e mudar apenas o valor do mês.

10.3 Calcular diferença entre datas

`Date` e `Calendar` possuem uma forma bem limitada de calcular a diferença entre duas datas. O único jeito é obter a diferença entre os timestamps, e portanto o resultado será em milissegundos. A seguir, você pode converter esta quantidade de milissegundos para a unidade desejada, como horas, minutos, dias etc.

Mas esta é uma abordagem limitada. Como já vimos no capítulo sobre durações, há unidades de tempo que não têm tamanho fixo, como meses e anos (e até mesmo dias, se levarmos em conta os *gaps* e *overlaps*). Sendo assim, não há um jeito fácil de converter uma quantidade de milissegundos para meses ou anos.

Já vimos este exemplo antes, mas apenas para relembrar: entre `2018-01-01` (1 de janeiro) e `2018-02-01` (1 de fevereiro) a diferença é de um mês. Mas, em dias, corresponde a 31. Já entre `2018-02-01` (1 de fevereiro) e `2018-03-01` (1 de março), a diferença também é de um mês. Mas em dias, corresponde a 28 (se fosse em um ano bissexto, seria 29).

Se calcularmos a diferença entre os timestamps (considerando meia-noite em UTC, em todos os dias), no primeiro caso o valor será correspondente a 31 dias (2678400000 milissegundos), enquanto no segundo caso será correspondente a 28 dias (2419200000 milissegundos). Se consideramos estes dias em um horário e timezone diferentes, os valores não serão necessariamente os mesmos, pois pode haver *gaps* ou *overlaps* entre estas datas.

E para saber que estes valores em milissegundos correspondem a um mês, você deve levar em conta as datas envolvidas, e

considerar todos os casos possíveis: se é ano bissexto, se devo considerar também as horas etc.

Não há uma solução fácil para isso. O melhor que você pode obter são aproximações, e sempre serão baseadas em premissas arbitrárias, como considerar que um mês tem 30 dias (ou qualquer outro valor arbitrário), além de fazer arredondamentos nos resultados.

Também é possível usar `TimeUnit` para fazer as conversões. Porém, não existem unidades para meses e anos, somente para dias (`TimeUnit.DAYS`). E mesmo assim, esta classe considera que um dia sempre tem 24 horas, portanto, não funciona nos casos envolvendo horário de verão.

Por isso, é muito difícil obter a diferença como uma duração ISO 8601, com o valor de cada campo devidamente separado. Qualquer duração maior que um dia pode corresponder a uma quantidade diferente de horas (e, consequentemente, de minutos, segundos e milissegundos) e o resultado sempre dependerá das datas envolvidas. A falta de um mecanismo decente para lidar com durações é uma limitação da API, e com certeza um dos motivadores para a criação do `java.time` .

10.4 Não confunda duração com data

No capítulo sobre durações já falamos sobre isso, mas não custa repetir: não confunda uma data/hora com uma duração:

- Uma data representa um ponto específico no calendário (dia, mês e ano), e um horário representa um ponto específico do dia (hora, minuto, segundo, frações de segundo).
- Uma duração representa uma quantidade de tempo, e não está atrelada a nenhum ponto específico do calendário.

Ambos usam as mesmas palavras (dias, meses, horas, minutos etc.) mas são dois conceitos totalmente diferentes. E embora estejam relacionados (você pode somar uma duração a uma data, obtendo outra data), eles não são a mesma coisa.

Tratar uma duração como se fosse uma data está errado. Embora às vezes possa funcionar, não é o ideal. Infelizmente, `Date` , `Calendar` e `SimpleDateFormat` não fornecem nenhuma maneira de lidar corretamente com durações. Por isso, muitas vezes estas classes são usadas erroneamente, tentando simular uma duração.

Como fazer parsing de duração?

Se eu recebo uma `String` com o valor `P1DT3H20M45.344S` (uma duração de 1 dia, 3 horas, 20 minutos, 45 segundos e 344 milissegundos) para ser somada a um `Calendar` , como fazer?

O único jeito é manipular a `String` manualmente, para obter os valores numéricos de cada campo, e em seguida usar `add()` para somar estes valores ao `Calendar` . Vale lembrar que o formato ISO 8601 permite que as durações omitam os campos cujo valor é zero, o que torna o código de parsing um pouco trabalhoso para tratar todos os casos.

Uma das abordagens mais comuns é usar expressões regulares, mas esta resposta do Stack Overflow mostra que não é tão trivial assim (https://stackoverflow.com/a/32045167/). Uma das expressões sugeridas, só para você ter uma ideia da complexidade, é esta:

```
^P(?!$)(\d+Y)?(\d+M)?(\d+W)?(\d+D)?(T(?=\d)(\d+H)?(\d+M)?(\d+S)?)?$
```

E esta expressão não considera as frações de segundo nem valores negativos (que algumas APIs permitem, embora a ISO 8601 não), ou seja, pode ficar mais complicada ainda. É uma pena que a API não possua uma maneira mais simples de lidar com durações.

Em muitos casos, alguns desenvolvedores tentam usar `SimpleDateFormat` para fazer parsing de uma duração, tratando os

valores numéricos como se fossem uma data e hora. Pode até "funcionar" em alguns casos, mas é uma abordagem completamente errada, como veremos a seguir.

```
// ERRADO: fazer parsing de uma duração, tratando-a como uma data
SimpleDateFormat parser = new SimpleDateFormat("'PT'H'H'm'M'");
// usar UTC, para evitar problemas com horário de verão
parser.setTimeZone(TimeZone.getTimeZone("UTC"));
// parse de uma duração (10 horas e 20 minutos)
Date date = parser.parse("PT10H20M");
// "duração" em milissegundos
long duracao = date.getTime();
```

O que este código faz, na verdade, é o parsing de um horário: 10:20 (dez e vinte da manhã). O pattern possui os campos `H` (hora) e `m` (minuto), e vários literais (as letras `P`, `T`, `H` e `M` entre aspas simples) para que o formato seja compatível com uma duração ISO 8601. Ou seja, estamos – erroneamente – tratando um horário como se fosse uma duração.

Como já vimos anteriormente, quando a data não está presente na `String` de entrada, `SimpleDateFormat` usa o valor "1 de janeiro de 1970", que, por sinal, é a mesma data do Unix Epoch. Por isso, 10:20 neste mesmo dia estará a 10 horas e 20 minutos depois do Epoch, e é isso que `getTime()` retorna, em milissegundos. Como o Unix Epoch corresponde ao timestamp zero, neste caso o retorno de `getTime()` corresponde exatamente a uma duração de 10 horas e 20 minutos.

Este é um caso em que "funciona", mas é simplesmente **uma grande coincidência** causada pela forma com que `SimpleDateFormat` trabalha internamente, já que esta classe foi projetada para lidar com datas e não com durações.

Nem sempre vai funcionar

O problema da abordagem anterior (tratar datas como se fossem durações) é que não funciona sempre. Ela não resolve, por exemplo, os casos em que a duração possui dias, meses e anos, já

que estas unidades possuem tamanhos variáveis que dependem das datas envolvidas em cada operação. Se a duração for de 1 mês, e eu usar a abordagem errada de tratá-la como se fosse uma data, posso acabar em uma situação como esta:

```
// ERRADO: tentando fazer parsing de uma duração, tratando-a como uma data
SimpleDateFormat parser = new SimpleDateFormat("'P'M'M'");
parser.setTimeZone(TimeZone.getTimeZone("UTC"));
// duração de 1 mês
Date date = parser.parse("P1M");
long duracao = date.getTime(); // qual será o valor da duração?
```

Desta vez, o pattern corresponde a uma duração com somente a quantidade de meses (`PnM`). A seguir, eu tento fazer o parsing de `P1M` (duração de 1 mês). No `SimpleDateFormat` , eu usei o campo `M` , que corresponde ao mês. Como o valor é `1` , isso corresponde a janeiro. E como os demais campos não estão na `String` , são usados valores predefinidos, que já vimos anteriormente que são: dia 1, ano 1970 e horário meia-noite.

Como o mês obtido no parsing foi janeiro, o resultado final é 1 de janeiro de 1970, meia-noite, em UTC (pois eu setei UTC no método `setTimeZone()`). Ou seja, o resultado é o próprio Unix Epoch, cujo valor de timestamp é zero. Por isso, a "duração" retornada é zero, o que é um resultado completamente diferente do que queríamos: a entrada é `P1M` , que deveria resultar em uma duração de 1 mês (ou seja, algum valor bem maior que zero).

Isso acontece justamente porque datas e durações são dois conceitos diferentes, e `SimpleDateFormat` , `Date` e `Calendar` só trabalham com um deles. Tratar datas como se fossem durações – e vice-versa – nem sempre vai funcionar.

O único jeito de lidar corretamente com durações é usar uma API que possua o devido suporte a este recurso. O `java.time` , que veremos na terceira parte do livro, é uma dessa APIs. Já usando `Date` e `Calendar` , não é possível, e você vai ter que fazer tudo manualmente.

Com isso, encerramos a aritmética de datas. No próximo capítulo, veremos as classes de data e hora do pacote `java.sql` .

Capítulo 11
As classes de data do pacote java.sql

O pacote `java.sql` contém várias classes para trabalhar com bancos de dados. E como praticamente todos os bancos possuem tipos relacionados a datas e horas, este pacote possui as respectivas classes:

- `java.sql.Date` : representa um SQL DATE, ou seja, somente a data (dia, mês e ano);
- `java.sql.Time` : representa um SQL TIME, ou seja, somente o horário (horas, minutos, segundos);
- `java.sql.Timestamp` : representa um SQL TIMESTAMP, ou seja, o valor do timestamp (tempo decorrido desde o Unix Epoch) – mas seu significado pode variar de acordo com a implementação. No Oracle, por exemplo, 3 tipos diferentes (*timestamp*, *timestamp with timezone* e *timestamp with local timezone*) são mapeados para esta classe (https://docs.oracle.com/cd/B19306_01/java.102/b14188/datamap.htm#r12c1-t7/). Leia a documentação do banco que você está usando para entender como ele usa esta classe.

Todas são subclasses de `java.util.Date` , portanto herdam sua principal característica: internamente, elas possuem o valor do timestamp, que representa um ponto na linha do tempo, e por isso correspondem a uma data e hora diferente, dependendo do timezone.

O objetivo deste capítulo **não** é explicar como lidar com datas em todos os bancos de dados (ou em algum banco específico), nem se aprofundar nos detalhes do JDBC, mas sim dar uma ideia geral de como as classes de data e hora do pacote `java.sql` funcionam, quais seus principais problemas e características e, principalmente, por que não foi uma boa ideia herdar de `java.util.Date` .

Para evitar confusão, neste capítulo vou chamar `java.util.Date` e `java.sql.Date` por seus nomes completos. Já `java.sql.Time` e `java.sql.Timestamp` serão chamadas apenas de `Time` e `Timestamp` .

11.1 java.sql.Date e Time não representam uma data e hora específicas

Por serem subclasses de `java.util.Date` , os problemas desta classe também foram herdados por `java.sql.Date` e `Time` . Um dos principais é o seu método `toString()` , cuja saída também é afetada pelo timezone padrão da JVM, conforme podemos ver no próximo exemplo:

```
// timestamp 1525464000000 = 2018-05-04T17:00-03:00 (America/Sao_Paulo)
java.sql.Date date = new java.sql.Date(1525464000000L);
Time time = new Time(1525464000000L);
System.out.println(date);
System.out.println(time);
// mudar o timezone padrão
TimeZone.setDefault(TimeZone.getTimeZone("Asia/Tokyo"));
System.out.println(date);
System.out.println(time);
```

Neste exemplo, foi usado o timestamp `1525464000000` , que equivale a `2018-05-04T17:00-03:00` (4 de maio de 2018, às 17:00 no timezone `America/Sao_Paulo` , que é o timezone padrão da JVM que estou usando). Então, eu crio ambas as classes com este timestamp e imprimo seus valores usando `System.out.println()` , que internamente chamará o método `toString()` . A seguir, eu mudo o timezone padrão para `Asia/Tokyo` e imprimo os mesmos valores novamente. A saída é:

```
2018-05-04
17:00:00
2018-05-05
05:00:00
```

Os valores do dia, mês, ano, hora, minuto e segundo dependem do timezone padrão. E como o timestamp `1525464000000` corresponde ao dia 4 às 17:00 em São Paulo e ao dia 5 às 05:00 em Tóquio, a mudança do timezone padrão muda o resultado. A forma com que `java.sql.Date` e `Time` são interpretados depende do timezone padrão, e sabe-se lá qual data ou hora vai ser realmente gravada no banco de dados, pois vai depender de qual timezone estava configurado naquele momento e/ou de como cada banco trata estas classes. É um erro difícil de detectar.

Mas pelo menos o método `toString()` retornou os valores no formato ISO 8601. Já é um avanço em relação a `java.util.Date` , pois é um formato cada vez mais difundido, pelo menos entre linguagens, APIs e sistemas em geral.

As classes `java.sql.Date` e `Time` possuem o mesmo problema conceitual de `java.util.Date` . Elas não representam um único dia/mês/ano ou hora/minuto/segundo, e sim um timestamp: um valor que pode representar uma data e hora diferentes, dependendo do timezone utilizado para interpretá-las. E, como já vimos anteriormente, o timezone padrão é uma configuração sobre a qual não temos muito controle.

Outro detalhe é que não há um construtor para criar a data/hora atual, como existe com `java.util.Date` (pois com esta podemos fazer apenas `new Date()`). Cada classe possui apenas dois construtores: um que recebe os valores numéricos (ano, mês e dia, ou hora, minuto e segundo), tem os mesmos problemas já conhecidos (ano indexado em 1900, mês indexado em zero) e está *deprecated*; e outro que recebe o valor do timestamp e foi usado no exemplo anterior.

Veremos como criar instâncias com a data e hora atual mais adiante. Provavelmente você pensou que basta passar o valor de `System.currentTimeMillis()` , só que é um pouco mais complicado que isso. Mas uma coisa de cada vez, primeiro vamos ver mais alguns detalhes destas classes.

Só tem a data?

Para poder representar apenas uma data, mesmo tendo um valor de timestamp, a classe `java.sql.Date` esconde artificialmente os campos relativos ao horário. Os métodos que retornam estes campos, como `getHours()` , `getMinutes()` , `getSeconds()` e os respectivos *setters*, lançam um `IllegalArgumentException` quando são chamados.

Apesar disso, é possível obter as horas de um `java.sql.Date` , usando um `SimpleDateFormat` :

```
// 1525464000000 = 2018-05-04T17:00-03:00 (America/Sao_Paulo)
java.sql.Date sqlDate = new java.sql.Date(1525464000000L);
SimpleDateFormat formatter = new SimpleDateFormat("dd/MM/yyyy HH:mm");
System.out.println(formatter.format(sqlDate));
```

O pattern usado possui as horas e minutos (`HH:mm`), e a saída é:

```
04/05/2018 17:00
```

Ou seja, mesmo não "tendo" as horas, é possível obtê-las. Estas informações estão escondidas, mas nem tanto. Para uma classe que pretende representar apenas a data, não é a melhor das implementações.

O mesmo vale para `Time` , que esconde artificialmente os campos de data através de uma exceção lançada pelos respectivos *getters* e *setters*. Mas ainda assim é possível obter tais valores usando `SimpleDateFormat` .

Iguais, mas nem tanto

Outro problema é quando queremos comparar duas instâncias de `java.sql.Date` ou `Time` . Vamos usar uma instância contendo o mesmo timestamp dos exemplos anteriores, e outra apenas um milissegundo depois. Em seguida, vamos imprimi-las, para garantir que, no timezone padrão da JVM (no meu caso, `America/Sao_Paulo`), ambas correspondem à mesma data. E depois vamos usar `equals()`

para saber se ambas são iguais, e `before()` , para saber se a data de uma é anterior à outra:

```
// 1525464000000 = 2018-05-04T17:00-03:00
java.sql.Date sqlDate = new java.sql.Date(1525464000000L);
// 1525464000001 = 2018-05-04T17:00:00.001-03:00
java.sql.Date sqlDate2 = new java.sql.Date(1525464000001L);
System.out.println(sqlDate);
System.out.println(sqlDate2);
System.out.println("iguais? " + sqlDate.equals(sqlDate2));
System.out.println("sqlDate antes de sqlDate2? " +
sqlDate.before(sqlDate2));
```

A saída é:

```
2018-05-04
2018-05-04
iguais? false
sqlDate antes de sqlDate2? true
```

Repare que o método `toString()` retornou a mesma saída (`2018-05-04`), portanto as duas instâncias representam a mesma data, certo? Só que o método `equals()` retornou `false` , o que indica que, na verdade, elas não são iguais. Da mesma forma que `before()` retornou `true` , indicando que `sqlDate` representa uma data anterior à `sqlDate2` .

Isso acontece porque os métodos `equals()` e `before()` comparam o valor do timestamp e, no nosso exemplo, as duas instâncias de fato possuem valores diferentes. E isso também afeta o resultado dos outros métodos de comparação (`after()` e `compareTo()`). Como resolver isso?

A documentação nos dá uma dica (https://docs.oracle.com/javase/8/docs/api/java/sql/Date.html/):

"To conform with the definition of SQL DATE, the millisecond values wrapped by a java.sql.Date instance must be 'normalized' by setting the hours, minutes, seconds, and milliseconds to zero in the particular time zone with which the instance is associated."

Resumindo, o valor do timestamp deve ser "normalizado", setando os campos do horário para zero. Temos que manipular o timestamp para que ele corresponda à data que queremos, mas com o horário setado para meia-noite no timezone "ao qual a instância está associada" (ou seja, qualquer timezone que faça sentido no contexto em que o código está rodando). E para isso, usamos um `Calendar`:

```
// criar Calendar com o valor do timestamp
Calendar cal = Calendar.getInstance();
cal.setTimeInMillis(1525464000000L);
// setar horário para meia-noite
cal.set(Calendar.HOUR_OF_DAY, 0);
cal.set(Calendar.MINUTE, 0);
cal.set(Calendar.SECOND, 0);
cal.set(Calendar.MILLISECOND, 0);
// criar o java.sql.Date
java.sql.Date sqlDate = new java.sql.Date(cal.getTimeInMillis());
```

Com isso, o exemplo anterior resultaria em duas instâncias iguais, pois os timestamps `1525464000000` e `1525464000001` seriam "normalizados" para ter o mesmo horário (meia-noite) e o `equals()` retornaria `true`.

E se eu quisesse criar um `java.sql.Date` com a data atual, bastaria usar o código anterior, mas sem chamar `cal.setTimeInMillis()`, pois `Calendar.getInstance()` já retorna um `Calendar` com a data atual. Porém, ele usa a data atual no timezone padrão da JVM, o que quer dizer que o problema não foi totalmente eliminado, já que o timestamp pode corresponder a outro dia, se mudarmos o timezone.

Para criar um `Time` também é necessário normalizar o timestamp, setando os campos de data para 1 de janeiro de 1970. Caso contrário, as comparações não funcionarão, pois os métodos `equals()`, `before()`, `after()` e `compareTo()` comparam o valor do timestamp.

Uma das críticas a `java.sql.Date` e `Time` é justamente o fato de termos que fazer toda esta manipulação com o timestamp antes de passá-lo para os respectivos construtores. Talvez as próprias classes devessem tratar estes casos internamente, inclusive deixando configuráveis informações como o timezone a ser usado, por exemplo. Ou talvez elas não devessem herdar de `java.util.Date`, nem depender do timezone padrão para termos o valor correto do dia, mês e ano (ou da hora, minuto e segundo). Ou, quem sabe ainda, sobrescrever os métodos `equals()`, `before()`, `after()` e `compareTo()`, de forma que eles ignorassem os campos que não são representados pela classe, já que é isso que elas se propõem a fazer.

Parsing com java.sql.Date e Time

Outra maneira de criar estas classes é usar o método estático `valueOf()`, que recebe uma `String` em formato ISO 8601. A vantagem deste método é que ele já normaliza os timestamps: `java.sql.Date` tem o horário setado para meia-noite e `Time` tem a data setada para 1 de janeiro de 1970.

```
// valueOf recebe Strings no formato ISO 8601
java.sql.Date sqlDate = java.sql.Date.valueOf("2018-05-04");
Time time = Time.valueOf("10:00:00");
```

Apesar de `valueOf()` não usar um `SimpleDateFormat`, estamos convertendo uma `String` em um determinado formato para um tipo que representa uma data. Então não deixa de ser um parsing.

Porém, se a `String` estiver em outro formato, você deve primeiro fazer o parsing para criar um `java.util.Date`, e em seguida passar o valor do timestamp para a classe que quer criar:

```
// parsing de String em formato diferente do ISO 8601
SimpleDateFormat parser = new SimpleDateFormat("dd/MM/yyyy");
// parse() retorna sempre um java.util.Date
java.util.Date date = parser.parse("04/05/2018");
```

```
// passar o timestamp para java.sql.Date
java.sql.Date sqlDate = new java.sql.Date(date.getTime());
```

Neste caso, o pattern só possui dia, mês e ano (`dd/MM/yyyy`). E como já vimos no capítulo sobre parsing com `SimpleDateFormat` , quando o horário não está no pattern, ele é automaticamente setado para meia-noite, no timezone padrão da JVM. Por isso, o timestamp do `java.util.Date` já está "normalizado" segundo as regras exigidas pelo construtor de `java.sql.Date` .

O mesmo vale para `Time` : se o pattern tiver apenas campos relativos ao horário, a data será setada para 1 de janeiro de 1970, fazendo com que o timestamp já esteja normalizado.

A diferença para `valueOf()` é que este usa sempre o timezone padrão da JVM, enquanto `SimpleDateFormat` pode mudar o timezone utilizado, através do método `setTimeZone()` .

Tão ruim quanto java.util.Date?

Todos os problemas de `java.sql.Date` e `Time` que acabamos de ver acontecem principalmente por elas serem subclasses de `java.util.Date` , e consequentemente por herdarem sua principal característica: o fato de não representarem uma única data, e sim um timestamp (um ponto na linha do tempo). Estas classes possuem um valor que pode representar uma data e hora diferentes em cada timezone, e além disso precisam ignorar alguns campos e o timezone em várias operações — e, como vimos, não conseguem fazer isso totalmente.

Além de todos esses problemas, ainda fizeram o favor de dar exatamente o mesmíssimo nome para `java.util.Date` e `java.sql.Date` , tornando o código bem confuso e mais verboso, pois não podemos ter os dois `imports` ao mesmo tempo e uma delas tem que ser usada com seu nome completo. Esse fato por si só não torna as classes ruins (há vários outros exemplos de classes com mesmo nome em

outras APIs do Java que não são necessariamente terríveis), mas fecha com chave de ouro a lista de problemas que ambas possuem.

11.2 java.sql.Timestamp, o único nome que faz sentido

Já vimos que `java.util.Date` e `java.sql.Date` não são datas, e `java.sql.Time` não é um horário. A classe `java.sql.Timestamp` é a única que parece ter um nome condizente com seu funcionamento, pois ela armazena o valor de um timestamp. Mas infelizmente ela também é uma subclasse de `java.util.Date` , e por isso os mesmos problemas são herdados dela.

O primeiro é o já conhecido método `toString()` , cuja saída depende do timezone padrão. Vamos usar um `Timestamp` com o valor `1525464000000` , que corresponde a `2018-05-04T17:00-03:00` , e em seguida imprimi-lo duas vezes: uma usando o timezone padrão `America/Sao_Paulo` e outra mudando o padrão para `Asia/Tokyo` :

```
// 2018-05-04T17:00-03:00
Timestamp ts = new Timestamp(1525464000000L);
System.out.println(ts);
TimeZone.setDefault(TimeZone.getTimeZone("Asia/Tokyo"));
System.out.println(ts);
```

A saída é:

```
2018-05-04 17:00:00.0
2018-05-05 05:00:00.0
```

Aqui acontece o mesmo problema de sempre: apesar de o valor do timestamp ser o mesmo, a saída é afetada pelo timezone padrão da JVM, resultando em um dia e horário diferentes. Outro detalhe é que a saída **não** está no formato ISO 8601, pois há um espaço separando a data e a hora, sendo que, na norma ISO 8601, o separador é a letra `T` maiúscula.

Além disso, a saída não dá nenhuma indicação de qual o timezone ou o offset usado. É apenas uma data e hora, sem nenhum contexto adicional. Mas uma data e hora sem offset pode corresponder a um instante diferente em cada parte do mundo, de forma que, dadas estas saídas, não há como saber o valor do timestamp correspondente.

Mais precisão

Ao contrário de `Date` e demais classes, que possuem precisão de milissegundos (3 casas decimais), a classe `Timestamp` possui precisão de nanossegundos (9 casas decimais).

Para isso, internamente ela trabalha com 2 campos: além do campo `long` herdado de `java.util.Date` , com o timestamp em milissegundos desde o Unix Epoch, há outro campo que contém o valor dos nanossegundos. Exemplo:

```
Timestamp ts = new Timestamp(1525464000123L);
System.out.println(ts.getTime());
System.out.println(ts.getNanos());
```

O timestamp `1525464000123` corresponde a `2018-05-04T17:00:00.123-03:00` , portanto o valor dos milissegundos é `123` . Só que internamente o `Timestamp` guarda este valor em nanossegundos, portanto `getNanos()` retorna `123000000` . A saída é:

```
1525464000123
123000000
```

Parsing de um java.sql.Timestamp

Assim como `java.sql.Date` e `Time` , a classe `Timestamp` também possui o método `valueOf()` , que recebe uma `String` no formato indicado no próximo exemplo:

```
Timestamp ts = Timestamp.valueOf("2018-05-04 10:30:45.123456789");
System.out.println(ts);
```

```
System.out.println(ts.getTime());
System.out.println(ts.getNanos());
```

Repare que o formato **não** é o ISO 8601, pois há um espaço entre a data e hora (no formato ISO 8601 é usado a letra `T`). Além disso, são aceitas até 9 casas decimais nas frações de segundo. A saída deste código é:

```
2018-05-04 10:30:45.123456789
1525440645123
123456789
```

Podemos ver que `getTime()` retorna o timestamp em milissegundos, porém o retorno de `getNanos()` indica que há mais casas decimais a serem consideradas.

O método `valueOf()` aceita somente `Strings` neste formato. Para outros formatos, deve ser usado um `SimpleDateFormat` , da mesma maneira que os exemplos anteriores: faz-se o parsing para obter um `java.util.Date` e, em seguida, usa-se este `Date` para passar o valor do timestamp para o construtor de `Timestamp` .

Mas lembre-se de que `SimpleDateFormat` só funciona com 3 casas decimais no máximo (nos capítulos anteriores vimos o que acontece quando há mais que 3). Quando há de 4 a 9 casas decimais, a melhor alternativa é separá-las da `String` e setá-las separadamente no `Timestamp` . Exemplo:

```
String input = "04/05/2018 10:30:45.123456789";
// separar frações de segundo do restante da data/hora
String[] dadosSeparados = input.split("\\.");
// fazer parsing da data/hora (sem os nanossegundos)
SimpleDateFormat parser = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");
Date date = parser.parse(dadosSeparados[0]);
// usar o valor do timestamp no construtor
Timestamp ts = new Timestamp(date.getTime());
// setar os nanossegundos
ts.setNanos(Integer.parseInt(dadosSeparados[1]));
System.out.println(ts);
```

```
System.out.println(ts.getTime());
System.out.println(ts.getNanos());
```

Primeiro, separamos os nanossegundos do restante da data/hora, usando `split()` . Isso cria o array `dadosSeparados` , que contém 2 elementos: um com a data e hora (dia, mês, ano, horas, minutos e segundos), e outro com as frações de segundo. A seguir, fazemos o parsing da data/hora (sem as frações de segundo) e usamos o timestamp resultante para criar o `Timestamp` . Por fim, setamos o valor dos nanossegundos, usando `Integer.parseInt()` para converter a `String` com as frações de segundo em um `int` . A saída é:

```
2018-05-04 10:30:45.123456789
1525440645123
123456789
```

Vale lembrar que este timestamp corresponde à data e hora indicadas, no timezone padrão da JVM. Se você quiser que ele corresponda a outro timezone, mude-o usando o método `setTimeZone()` no `SimpleDateFormat` .

Também não foi considerado o caso em que as frações de segundo têm menos que 9 dígitos. Por exemplo, o valor `.123456` corresponde a 123456000 nanossegundos, então a `String` a ser passada para `parseInt()` deve ser completada com zeros à direita caso tenha menos que 9 dígitos.

Devo parar de usar estas classes?

Apesar de todos os problemas de `java.sql.Date` , `Time` e `Timestamp` , ainda há muito código legado que usa estas classes, e nem sempre é possível deixar de usá-las.

Somente a partir do JDBC 4.2 é possível usar as classes do `java.time` para gravar e ler campos de data em um banco de dados. Consulte a documentação do seu banco de dados para saber se o driver disponibilizado já é compatível com a nova API.

De qualquer forma, mesmo que você vá continuar usando estas classes, estar ciente do seu funcionamento ajuda a trabalhar melhor com elas. Saber que a saída do método `toString()` depende do timezone padrão e por isso nos engana, além dos outros detalhes que já vimos, nos prepara melhor para debugar e até para escrever código que lida melhor com suas limitações.

Agora que já vimos o funcionamento básico das classes de data e hora do pacote `java.sql` , vamos ver alguns casos de uso com `Calendar` e `TimeZone` .

Capítulo 12
Timezones e outros casos de uso

Neste capítulo, veremos mais detalhes da classe `java.util.TimeZone` , além de mais alguns exemplos práticos com a API.

Também veremos de onde a JVM pega as informações sobre o histórico de offsets de cada timezone, e como manter estes dados atualizados de acordo com as versões do TZDB da IANA.

12.1 Obtendo informações do timezone

A informação mais óbvia a ser obtida de um timezone é o seu nome. O método `getID()` retorna o identificador da IANA, e o método `getDisplayName()` retorna um "nome amigável", conforme podemos ver no código a seguir:

```
TimeZone zone = TimeZone.getTimeZone("America/Sao_Paulo");
System.out.println(zone.getID());
System.out.println(zone.getDisplayName());
```

A saída é:

```
America/Sao_Paulo
Fuso horário de Brasília
```

`getDisplayName()` usa o locale padrão da JVM para saber o idioma no qual o nome estará. Se você quiser o nome do timezone em outro idioma, basta usar o locale correspondente. Por exemplo, `getDisplayName(Locale.US)` retorna o nome em inglês (`Brasilia Time`).

Obtendo os offsets

Para saber qual o offset usado por um timezone em uma data e hora específica, basta usar o método `getOffset()`, passando como parâmetro o valor numérico do timestamp.

Lembre-se de que um timezone possui todo o histórico de offsets de uma determinada região, incluindo a data e hora em que há uma mudança de offset, seja por causa do horário de verão, seja porque o governo daquela região resolveu mudar o horário local. Por isso, cada timezone possui uma lista de vários offsets, e em cada instante podemos ter um valor diferente.

O método `getOffset()` precisa receber um instante como referência. Se você quiser usar o timestamp atual, basta passar o retorno de `System.currentTimeMillis()`. Caso você já tenha um objeto `Date`, use `getTime()` para obter o timestamp, e passe este valor para `getOffset()`. E se você tiver um `Calendar`, passe o valor de `getTimeInMillis()`.

O retorno de `getOffset()` é o valor do offset em milissegundos. Se quiser converter para horas e minutos, basta usar `TimeUnit`, conforme já vimos anteriormente.

Informações sobre horário de verão

A classe `TimeZone` possui alguns métodos para obter informações sobre o horário de verão. O mais simples é `inDaylightTime(Date)`, que retorna `true` se, no instante representado pelo `Date`, o timezone está em horário de verão. Ele é bem útil porque permite que se consulte qualquer data no passado, presente ou futuro.

Há também o método `getDSTSavings()`, que retorna a quantidade de milissegundos a ser adicionada ao horário local, quando é horário de verão. Por exemplo, para `America/Sao_Paulo`, o retorno é `3600000`, que corresponde a uma hora.

Infelizmente, a API não oferece muitos recursos para obtermos informações sobre o timezone. Não há, por exemplo, uma maneira direta de saber quando é a próxima mudança de offset. Uma

alternativa é usar a API `java.time` , que explicaremos nos próximos capítulos.

12.2 Alguns casos práticos

Apesar de limitada, a classe `TimeZone` pode ser usada para alguns casos de uso que vimos nos capítulos anteriores.

Dado um offset, como saber o timezone?

Como já vimos, dado um offset, não é possível ter um único timezone como resposta. Há mais de um timezone que usa o mesmo offset, e além disso há mudanças o tempo todo, seja por causa do horário de verão ou simplesmente porque um país decidiu mudar seu horário local. Por isso, a lista de timezones que usa determinado offset pode variar conforme a data em que consultamos.

No código a seguir, vamos procurar os timezones que usam o offset `+02:00` , usando a data atual como referência. Primeiro, temos que converter o offset de duas horas para o valor equivalente em milissegundos, para poder compará-lo com o retorno de `getOffset()` .

E para a data de referência, usamos `System.currentTimeMillis()` , já que `getOffset()` recebe como parâmetro o timestamp. Como queremos somente o valor numérico, não é preciso criar um `Date` só para chamar `getTime()` em seguida.

```
// offset +02:00 em milissegundos
long offset = TimeUnit.MILLISECONDS.convert(2, TimeUnit.HOURS);
// timestamp a ser usado como referência (equivalente a 2018-05-04T17:00-03:00)
long referencia = System.currentTimeMillis();
// procurar todos os timezones que usam o offset na data de referência
for (String id : TimeZone.getAvailableIDs()) {
    // obtém o offset usado neste timezone, na data de referência
```

```
    int offsetUsado = TimeZone.getTimeZone(id).getOffset(referencia);
    if (offset == offsetUsado) {
        System.out.println(id);
    }
}
```

Serão impressos vários timezones, e a quantidade pode variar, pois vai depender da versão do TZDB sendo usado na sua JVM e da data atual no momento em que o código rodar. No meu teste, obtive 60 timezones, entre eles `Africa/Ceuta` e `Africa/Cairo` . Se você mudar a data de referência, alguns nomes da lista também podem mudar, conforme já explicado no capítulo sobre timezones.

Quais timezones usam determinada abreviação?

Nos capítulos anteriores vimos que as abreviações de timezones (como EST, IST, entre outras) são ambíguas, já que mais de um timezone pode usá-las, muitas vezes em países diferentes. Por isso, dada uma abreviação, não há como obter um único timezone. E como vários países usam uma abreviação diferente durante o horário de verão, eu também preciso definir uma data de referência. O melhor que podemos ter é uma lista de timezones que usam a abreviação em uma determinada data.

Para saber a abreviação usada por cada timezone na data de referência, eu posso usar um `SimpleDateFormat` com o pattern `z` (minúsculo), que corresponde à abreviação do timezone. No próximo exemplo, eu também percorro todos os locales, pois a abreviação é *locale sensitive*. Por fim, eu uso um `java.util.Set` para guardar os nomes dos timezones, para evitar imprimi-los mais de uma vez (pois a abreviação pode ser a mesma em mais de um locale):

```
// abreviação
String abrev = "EST";
// data a ser usada como referência (2018-05-04T17:00-03:00)
Date referencia = new Date();
// Set para guardar os resultados
```

```
Set<String> timezones = new HashSet<>();
// verificar todos os locales
for (Locale locale : Locale.getAvailableLocales()) {
    SimpleDateFormat formatter = new SimpleDateFormat("z", locale);
    // verificar todos os timezones
    for (String id : TimeZone.getAvailableIDs()) {
        // obtém a abreviação usada por este timezone, na data de
referência
        formatter.setTimeZone(TimeZone.getTimeZone(id));
        if (abrev.equals(formatter.format(referencia))) {
            timezones.add(id);
        }
    }
}
System.out.println(timezones);
```

Alguns timezones que usam a abreviação "EST" (como `America/New_York` e `America/Toronto`) não estarão no resultado final, pois na data de referência (4 de maio de 2018, às 17:00 em São Paulo) eles estão em horário de verão, portanto a abreviação sendo usada por eles neste instante é "EDT". É claro que, se você rodar este código em outro dia, quando estes timezones não estão em horário de verão, eles estarão no resultado final.

12.3 Sempre há um timezone em ação

Mesmo nos casos em que você menos espera, há um timezone agindo nos bastidores. O design de `Calendar` faz com que os efeitos de *gaps* e *overlaps* afetem seus resultados, mesmo que você não queira.

Como exemplo, vamos implementar um caso de uso bem comum, que é encontrar o próximo dia da semana a partir de uma data. No caso, vamos começar em 27 de dezembro de 2011, e a partir desta data, encontrar a próxima sexta-feira.

```
// 27 de dezembro de 2011
Calendar cal = Calendar.getInstance();
cal.set(2011, Calendar.DECEMBER, 27);
// vou somando um dia, até encontrar a próxima sexta-feira
do {
    cal.add(Calendar.DAY_OF_MONTH, 1);
} while (cal.get(Calendar.DAY_OF_WEEK) != Calendar.FRIDAY);
System.out.println(cal.getTime());
```

`getInstance()` retorna a data e hora atual, e ao mudar a data com `set()` , o horário é mantido. Neste exemplo, o resultado foi `Fri Dec 30 17:00:00 BRST 2011` , já que a hora atual é 17:00 e o timezone padrão é `America/Sao_Paulo` , que em dezembro está em horário de verão e usa a abreviatura BRST (Brasília Summer Time). Na maioria dos casos, vai funcionar como esperado, resultando em 30 de dezembro de 2011. Mas dependendo do timezone, o resultado pode ser outro.

Por exemplo, se o timezone padrão for `Pacific/Apia` , será que o resultado é o mesmo? Lembre-se de que neste timezone, o dia 30 de dezembro de 2011 foi pulado, devido à mudança para o outro lado da Linha Internacional de Data. E como `Calendar.add()` ajusta as datas conforme as regras do timezone, o dia 30 será pulado, e o resultado será a sexta-feira seguinte (5 de janeiro de 2012).

Se você estava realmente querendo saber qual a próxima sexta-feira naquele timezone específico, tudo bem. O dia 30 foi pulado e não existe naquele timezone, então faz sentido que ele não seja considerado a próxima sexta-feira. Mas se você estava fazendo um cálculo simples envolvendo as datas, e não queria saber de nenhum timezone específico, este comportamento pode ser um inconveniente.

"Ah, mas qual a chance de acontecer aqui?"

O caso de `Pacific/Apia` parece ser um exemplo muito extremo e "*nunca vai acontecer aqui onde moro*". Eu diria que "nunca" é uma palavra muito forte, mas concordo que a probabilidade é bem menor. De qualquer forma, mesmo se usarmos um timezone como

`America/Sao_Paulo` , ainda há situações que podem nos pegar desprevenidos.

Se eu usar o mesmo código anterior, mas começando em 14 de outubro de 2017, e setando o horário para meia-noite, qual será o resultado?

```
// 14 de outubro de 2017, à meia-noite - timezone padrão é
America/Sao_Paulo
Calendar cal = Calendar.getInstance();
cal.set(2017, Calendar.OCTOBER, 14, 0, 0, 0);
// vou somando um dia, até encontrar a próxima sexta-feira
do {
    cal.add(Calendar.DAY_OF_MONTH, 1);
} while (cal.get(Calendar.DAY_OF_WEEK) != Calendar.FRIDAY);
System.out.println(cal.getTime());
```

O resultado é:

```
Fri Oct 20 01:00:00 BRST 2017
```

O dia está correto (20 de outubro de 2017, a sexta-feira seguinte), mas o horário é 01:00. Isso acontece porque no dia 15 o horário de verão começa, e à meia-noite o relógio é adiantado em uma hora, direto para 01:00. Portanto, 00:00 não existe neste dia e neste timezone, e o horário é ajustado para a próxima hora válida, que é 1 da manhã.

Como o dia 15 não é uma sexta-feira, o *loop* continua. A próxima iteração vai somar um dia, resultando no dia 16 às 01:00, e assim por diante, até chegar no dia 20.

Uma maneira de corrigir seria, ao final do `do/while` , setar o horário novamente para meia-noite. Ou então usar `Calendar.getInstance(TimeZone.getTimeZone("UTC"))` , já que UTC não sofre os efeitos do horário de verão. Se você só quer saber a data e a hora, é uma abordagem válida. Mas se você precisa do timestamp, a mudança do timezone para UTC vai gerar valores diferentes.

Enfim, cada caso é um caso, e cabe a você escolher a melhor solução de acordo com o que seu sistema precisa. Essa é mais uma das muitas críticas a `Calendar` : o fato de você sempre ter que se preocupar com o timezone, mesmo nos casos em que os seus efeitos não são muito óbvios.

12.4 De onde o Java pega as informações de timezone

Quando você chama o método `TimeZone.getTimeZone()` passando como parâmetro um identificador da IANA, como `America/Sao_Paulo` , ele retorna uma instância que contém todo o histórico de offsets deste timezone. De onde o Java pega essas informações?

Já vimos que o site da IANA (https://www.iana.org/time-zones/) informa a última versão do TZDB disponível. Além disso, há um diretório com todas as versões anteriores (https://data.iana.org/time-zones/releases/).

Mas o método `getTimeZone()` não busca estas informações na internet. Na verdade, quando você instala o Java, ele já vem com uma versão do TZDB embutida, dentro da pasta `JDK_HOME/jre/lib` ou `JRE_HOME/lib` .

Para saber qual a versão do TZDB que vem na instalação do Java, você pode conferir no site da Oracle (http://www.oracle.com/technetwork/java/javase/tzdata-versions-138805.html/). Além das versões da JRE e do TZDB, esta página também possui um resumo das principais mudanças que aquela versão introduz. Segue um trecho copiado – e traduzido – desta página para você ter uma ideia das informações que ela contém (removi alguns trechos por questões de brevidade):

Versão do Tzdata	Versão do JRE	Principais mudanças
tzdata2018e		Coreia do Norte muda para +09 em 2018-05-05.
tzdata2018d		Palestina começa o horário de verão uma semana antes em 2018, em 24 de março, e não 31.
tzdata2018c	10, 8u172, 7u181, 6u191	São Tomé e Príncipe mudou de +00 para +01. Horário de verão no Brasil começa no primeiro domingo de novembro.

Nesta página podemos ver que a nova regra do horário de verão do Brasil (início no primeiro domingo de novembro) está na versão `2018c` do TZDB (identificado na primeira coluna, como "tzdata2018c"), que por sua vez está embutido nas versões `10` , `8u172` , `7u181` e `6u191` da JRE. Instalando qualquer uma destas versões, a classe `TimeZone` já estará usando estas novas regras.

Também podemos ver que as versões `2018d` e `2018e` do TZDB ainda não estão disponíveis em nenhuma versão do Java. Eu consultei a página da Oracle em maio de 2018, então pode ser que, quando você consultá-la, estas informações já tenham mudado.

E agora, espero a Oracle ou tem outro jeito?

Infelizmente, as atualizações da Oracle nem sempre são tão rápidas quanto as da IANA. Na minha opinião, é compreensível, afinal, uma nova *release* do JDK não vai ser lançada apenas para atualizar o TZDB. Geralmente a atualização de timezones é só mais uma dentre os muitos itens que são corrigidos ou melhorados a cada nova versão.

Felizmente, você não precisa esperar por uma nova versão do JDK. Basta usar o *Timezone Updater Tool*, ou simplesmente *TZUpdater*. Em seu site há instruções para baixá-lo e usá-lo (http://www.oracle.com/technetwork/java/javase/documentation/tzupdater-readme-136440.html/).

O *TZUpdater* é basicamente um `jar` que você roda com a JVM que você quer atualizar, passando como parâmetros o respectivo arquivo da IANA. Exemplo:

```
// rodar o TZUpdater com a última versão do TZDB
java -jar tzupdater.jar -l https://www.iana.org/time-zones/repository/tzdata-latest.tar.gz
```

Esta opção é o jeito mais comum de rodar, pois atualiza a JVM para a versão mais atual do TZDB. Você também pode rodá-lo com o parâmetro `-h` , que mostra todas as opções disponíveis. Uma delas é `-V` , que informa qual a versão do TZDB que está na JVM. Na minha máquina, a saída foi:

```
JRE tzdata version: tzdata2018e
```

Como podem ver, estou usando a versão `2018e` do TZDB (indicado por **"tzdata2018e"**). No momento em que fiz este teste, esta era a versão mais atual. Porém, em outubro de 2018, foram lançadas as versões `2018f` e `2018g` . E quando você estiver lendo este parágrafo, pode ser que já exista uma versão mais nova.

Sempre consulte o site da IANA (https://www.iana.org/time-zones/) para saber qual é a última versão do TZDB. Como já foi dito anteriormente, a única certeza que temos sobre os timezones é que suas regras estão sempre mudando. Nada é garantido, nada é para sempre.

Quando for rodar o *TZUpdater*, não se esqueça de parar a JVM, conforme recomendação do FAQ da Oracle (http://www.oracle.com/technetwork/java/javase/dst-faq-138158.html#restart/). Certifique-se também de que o usuário que

for rodá-lo tem permissão para alterar o conteúdo dos diretórios `JDK_HOME/jre/lib` ou `JRE_HOME/lib` .

12.5 Outros casos de uso

Deixando um pouco de lado os problemas de timezones (mas sabendo que eles podem nos afetar sem percebermos), vamos ver mais algumas situações bem comuns no dia a dia, e como implementá-las com `Calendar` e as demais classes da API.

Achar o último dia do mês

Como um mês pode ter 28, 29, 30 ou 31 dias, você teria que verificar várias condições (meses que têm 30 dias, se o ano é bissexto etc.) para saber o valor exato do último dia do mês. Felizmente, `Calendar` possui o método `getActualMaximum()` , que retorna o maior valor que um campo pode ter, levando em conta todos os outros campos do `Calendar` .

Como eu quero saber o maior valor possível para o dia do mês, basta usar como parâmetro o campo `DAY_OF_MONTH` :

```
// criar uma data em fevereiro de 2018
Calendar cal = Calendar.getInstance();
cal.set(2018, Calendar.FEBRUARY, 16);
// mudar para o último dia do mês (28 de fevereiro de 2018)
cal.set(Calendar.DAY_OF_MONTH,
cal.getActualMaximum(Calendar.DAY_OF_MONTH));
```

`getActualMaximum()` verifica o maior valor que `DAY_OF_MONTH` pode ter, baseado nos demais campos (fevereiro de 2018), portanto o valor retornado é `28` . Com isso, podemos mudar o valor deste campo, usando o método `set()` . Se o ano do `Calendar` fosse 2020, `getActualMaximum()` retornaria `29` , já que 2020 é um ano bissexto.

`Calendar` também possui outros métodos com nomes parecidos, como `getMaximum()` e `getLeastMaximum()` , que não necessariamente vão retornar o último dia do mês.

`getMaximum()` retorna o maior valor possível para o campo, independente dos valores atuais do `Calendar` . Ou seja, `getMaximum(Calendar.DAY_OF_MONTH)` sempre retorna `31` , independente de qual for a data e hora do `Calendar` , já que este é o valor máximo que o dia do mês (de **qualquer** mês) pode ter.

`getLeastMaximum()` retorna o "menor máximo". Ou seja, quando eu chamo `getLeastMaximum(Calendar.DAY_OF_MONTH)` , ele verifica o seguinte: dentre todos os valores que um mês pode ter para o último dia, qual o menor deles? A resposta é `28` , pois não existe um mês que pode ter menos que 28 dias. Este valor também independe da data/hora do `Calendar` .

Achar a terceira segunda-feira do mês

Esta também é uma situação bem comum, pois muitas datas são baseadas na ocorrência do dia da semana dentro de um mês. Exemplos: dia das mães (dependendo do país) é no segundo domingo de maio, o feriado de *Thanksgiving* nos EUA é na quarta quinta-feira de novembro, o horário de verão no Brasil (a partir de 2018) começa no primeiro domingo de novembro etc.

Para achar a terceira segunda-feira do mês, você poderia setar o `Calendar` para o primeiro dia do mês, fazer um *loop* até achar a primeira segunda-feira, e então somar 14 dias. Mas há uma maneira mais direta, usando o campo `Calendar.DAY_OF_WEEK_IN_MONTH` .

Este campo considera que os dias 1 a 7 do mês sempre estão na primeira semana (e para estes dias, o valor de `DAY_OF_WEEK_IN_MONTH` é `1`), de 8 a 14 estão na segunda semana (e o valor de `DAY_OF_WEEK_IN_MONTH` é `2`) e assim por diante.

Se eu quero a terceira segunda-feira do mês, basta eu setar o `DAY_OF_WEEK_IN_MONTH` para `3` e depois mudar o dia da semana para

`Calendar.MONDAY` . O resultado será a segunda-feira da terceira semana do mês, que é o mesmo que a terceira segunda-feira do mês:

```
// achar a terceira segunda-feira do mês atual
Calendar cal = Calendar.getInstance();
// primeiro mudamos para a terceira semana
cal.set(Calendar.DAY_OF_WEEK_IN_MONTH, 3);
// depois mudamos para a segunda-feira desta semana
cal.set(Calendar.DAY_OF_WEEK, Calendar.MONDAY);
```

Chega dessa API, vamos para a próxima

Com isso, chegamos ao fim da segunda parte do livro.

Obviamente, não vimos 100% da API legada, e nem era esta a intenção. Mas vimos seu funcionamento básico em detalhes, além de vários casos de uso comuns no dia a dia. Com isso, espero que você tenha um entendimento melhor sobre como a usar, e principalmente como não a usar.

Saber como estas classes funcionam também vai ajudar muito a migrar seu código para o `java.time` , que é a API que veremos nos próximos capítulos. Esta API implementa os conceitos de data e hora de maneira bem diferente de `Date` e `Calendar` , e saber estas diferenças é crucial para fazer a migração de uma para a outra.

A API java.time

Agora que já vimos como `Date` e `Calendar` funcionam, é hora de ver a nova API de datas do JDK 8: as classes do pacote `java.time` . Na verdade, ela não é tão nova assim, pois foi lançada em 2014.

A motivação para se ter uma nova API surgiu a partir dos problemas que vimos nos capítulos anteriores. `Date` e `Calendar` (e demais classes relacionadas, como `SimpleDateFormat` , `TimeZone` e as subclasses de `Date` no pacote `java.sql`) se mostraram problemáticas demais, difíceis de se trabalhar e confusas em seu funcionamento.

Com isso, surgiu a JSR 310 (*Java Specification Request*), uma proposta que descreve como deveria ser a nova API, e que pode ser lida na íntegra no site do JCP — *Java Community Process* (https://jcp.org/en/jsr/detail?id=310/). O JCP é o meio pelo qual são propostas alterações na linguagem Java. Para mais detalhes sobre este processo, veja o FAQ (https://jcp.org/en/introduction/faq/).

A JSR 310 — também chamada em inglês de *three-ten* ("três-dez") — visou atacar vários pontos problemáticos da API anterior, e ao longo dos próximos capítulos veremos em detalhes como isso foi feito. Também veremos como trabalhar com código que usa as duas APIs e como fazer corretamente as conversões entre elas.

Mas eu não uso Java 8, posso pular esta parte?

Se você não está usando o JDK 8 (ou superior), não tem problema. Para o JDK 6 e 7, existe o ThreeTen Backport (http://www.threeten.org/threetenbp), que é um excelente *backport* do `java.time` . Ela foi feita por **Stephen Colebourne**, um dos líderes da JSR 310.

Caso você queira usar o ThreeTen Backport, é interessante ler os próximos capítulos, pois os conceitos e funcionalidades do `java.time` também estão presentes nesta biblioteca.

E para quem ainda está usando JDK 5, uma alternativa é o Joda-Time (http://www.joda.org/joda-time/), outra biblioteca criada por Stephen Colebourne. Por terem sido criadas pela mesma pessoa, muitos acham que o `java.time` é idêntico ao Joda-Time. Mas na verdade há diferenças, tanto de conceitos quanto de implementação, como o próprio autor explica em seu blog (https://blog.joda.org/2009/11/why-jsr-310-isn-joda-time_4941.html/). Ele também escreveu um post interessante listando as principais semelhanças e diferenças entre as APIs (https://blog.joda.org/2014/11/converting-from-joda-time-to-javatime.html/). No site do Joda-Time há um aviso (http://www.joda.org/joda-time#Support) dizendo que ele é um projeto "encerrado" e não há mais melhorias planejadas, além de aconselhar a migração para o `java.time` . Sendo assim, eu só recomendaria esta biblioteca caso você esteja usando o JDK 5 e não haja perspectiva de mudar para uma versão mais nova.

Capítulo 13
Princípios básicos do java.time

Até o JDK 7, para representar datas e horas só podíamos usar `Date` , `Calendar` ou as subclasses de `Date` no pacote `java.sql` . Mas já vimos que estas classes não são boas para situações em que não temos todas as informações (data, hora e timezone/offset). Muitas vezes precisamos apenas da data ou da hora, e a API legada não possui um mecanismo decente para lidar com estas situações.

No `java.time` existem classes específicas para cada situação. Para criar uma data (**apenas** o dia, mês e ano), sem se preocupar com as horas ou timezones, basta usar a classe `java.time.LocalDate` e seu método estático `of()` :

```
// 4 de maio de 2018
LocalDate data = LocalDate.of(2018, 5, 4);
```

Nesta API, o ano não é indexado em 1900 (ou qualquer outro valor arbitrário), o mês não é indexado em zero (janeiro não é mais "o mês zero", os meses têm valores de 1 a 12) e a data criada é 4 de maio de 2018. Não há nenhum horário sendo setado para meia-noite no timezone padrão da JVM, simplesmente porque esta classe não tem nenhuma informação sobre horas ou timezones. Ela só possui 3 valores numéricos: o dia, o mês e o ano.

Pode parecer um detalhe bobo, mas isso faz toda a diferença. Nos capítulos anteriores vimos os problemas que surgem ao usar `Date` ou `Calendar` para criar uma data específica. O principal é o fato de sempre termos que nos preocupar com o horário e o timezone, pois devido aos *gaps* e *overlaps* nem sempre aquela combinação de data e hora existe no timezone em questão. Aqui não há esta preocupação, um `LocalDate` só possui informações sobre a data. Para usar horas e timezones, há outras classes especializadas, que veremos ao longo deste e dos próximos capítulos.

Repare também no uso do método estático `of()` em vez de um construtor. O `java.time` usa estes métodos para criar suas instâncias, não disponibilizando nenhum construtor público para suas classes. Esta técnica é conhecida como *factory method pattern* e você pode ver uma boa explicação sobre ela nesta resposta do Stack Overflow (https://stackoverflow.com/a/38563189/), além de uma breve discussão sobre o assunto nesta pergunta (https://stackoverflow.com/q/45585811/).

De forma resumida, esta técnica permite que a API tenha nomes mais expressivos para os métodos que criam objetos. Talvez `of()` não seja o melhor exemplo, mas veremos outros ao longo do livro, e espero que você perceba que de fato isso torna o código mais legível.

Outra diferença é que esta API não tem o comportamento leniente como padrão. `Date` e `Calendar` aceitavam valores maiores que os permitidos, como mês `15` ou dia `45` , e faziam ajustes no resultado

final. Já o método `of()` lança uma exceção quando o valor está fora dos limites permitidos. Exemplo:

```
// tentando criar data com mês 13
LocalDate mesInvalido = LocalDate.of(2018, 13, 1);
```

Este código lança uma exceção:

```
java.time.DateTimeException: Invalid value for MonthOfYear (valid values 1
- 12): 13
```

Repare que a mensagem da exceção diz qual o campo que deu problema (`MonthOfYear` , que corresponde ao mês), quais são os valores válidos (`1` a `12`) e o valor inválido que foi passado (`13`).

O mesmo acontece se eu tentar criar datas como 29 de fevereiro em anos que não são bissextos ou 31 de abril: o método `of()` lança uma `DateTimeException` , dizendo que a data é inválida.

13.1 Classes de data e hora local

Além de `LocalDate` , há duas outras classes que não possuem informações sobre o timezone, e por isso não são afetadas pelos *gaps* e *overlaps*:

- `java.time.LocalTime` : representa somente um horário (horas, minutos, segundos e frações de segundo), sem nenhuma informação sobre a data ou timezone.
- `java.time.LocalDateTime` : representa uma data e hora, sem informações sobre timezone. No fundo, é uma combinação de um `LocalDate` com um `LocalTime` .

Informalmente eu chamo estas classes de **"tipos locais"** ou **"classes locais"** (até por terem `Local` no nome). A partir de agora usarei estes termos para me referir a estas 3 classes.

> Atenção quanto à nomenclatura destas classes, pois `Local` pode dar a entender que elas se referem a algum lugar específico. Mas na verdade elas não se referem a nenhum lugar, pois não possuem um timezone, e talvez o prefixo `Local` não tenha sido a melhor escolha de nome. Há uma discussão sobre isso *nos comentários* desta resposta no Stack Overflow (https://stackoverflow.com/a/32443004/).

Estas classes também possuem o método `of()` para criar uma instância:

```
// 17:30 (cinco e meia da tarde)
LocalTime horario = LocalTime.of(17, 30);
// 2018-05-04T17:30
LocalDateTime dataHora = LocalDateTime.of(2018, 5, 4, 17, 30);
```

Se qualquer valor inválido for passado para um campo (hora `25` , minuto `99` etc.) o método `of()` lança uma `DateTimeException` informando qual é o problema.

Repare que os métodos só receberam os valores das horas e minutos (`17` e `30`) e por isso os segundos e as frações de segundo foram automaticamente setados para zero. Mas é possível passar estes valores também, como veremos a seguir.

Mais precisão para as frações de segundo

`Date` e `Calendar` possuem precisão de milissegundos (3 casas decimais na fração de segundos). Qualquer valor com mais casas decimais não é suportado — lembre-se da bagunça que `SimpleDateFormat` faz quando encontra mais que 3 dígitos.

Na API `java.time` a precisão foi aumentada para 9 dígitos (nanossegundos). Sendo assim, é possível criar um `LocalTime` com as 9 casas decimais na fração de segundos:

```
// 17:30:25.123456789
LocalTime hora = LocalTime.of(17, 30, 25, 123456789);
```

E se eu quiser um valor com precisão menor? Por exemplo, quero que a hora seja `17:30:25.123` (o valor da fração igual a 123 milissegundos). Como `LocalTime` sempre guarda o valor em nanossegundos, temos que multiplicar os milissegundos por 1 milhão, para obter o equivalente em nanossegundos:

```
// 17:30:25.123 (123 milissegundos = 123000000 nanossegundos)
LocalTime hora = LocalTime.of(17, 30, 25, 123000000);
```

13.2 Obter a data e hora atual

Para obter a data e/ou a hora atual, basta usar o método estático `now()`:

```
// 2018-05-04
LocalDate dataAtual = LocalDate.now();
// 17:00
LocalTime horarioAtual = LocalTime.now();
// 2018-05-04T17:00
LocalDateTime dataHoraAtual = LocalDateTime.now();
```

Este é um ótimo exemplo de como o *factory method* torna o código mais legível e expressivo (afinal, o nome do método é literalmente "agora"). Na minha opinião, acho que assim fica mais claro qual é o valor retornado, se comparado a `new LocalDate()` por exemplo.

Mas há um detalhe a ser observado. Nos primeiros capítulos vimos que a noção de "hoje" é relativa, pois depende de onde você está. Agora, neste exato momento, em cada parte do mundo, o dia de hoje pode não ser o mesmo: enquanto no Brasil é dia 4 de maio, no Japão já pode ser dia 5. Além disso, o horário também pode não ser o mesmo.

Então como o método `now()` sabe qual é a data e hora atual? Simples: ele usa o timezone padrão da JVM.

Pensei que estas classes não tinham timezone

De fato, as classes locais (`LocalDate` , `LocalTime` e `LocalDateTime`) não possuem nenhuma informação sobre timezones.

Mas a data e hora atual podem ser diferentes em cada parte do mundo, então o método `now()` precisa de um timezone para obter os valores numéricos do dia, mês, ano, horas, minutos, segundos e frações de segundo. E por isso ele usa o timezone padrão da JVM. Depois que esses valores são obtidos, o timezone é **descartado**, e cada classe pega os valores que precisa para criar sua respectiva instância, sem nenhuma referência ao timezone usado para criá-la.

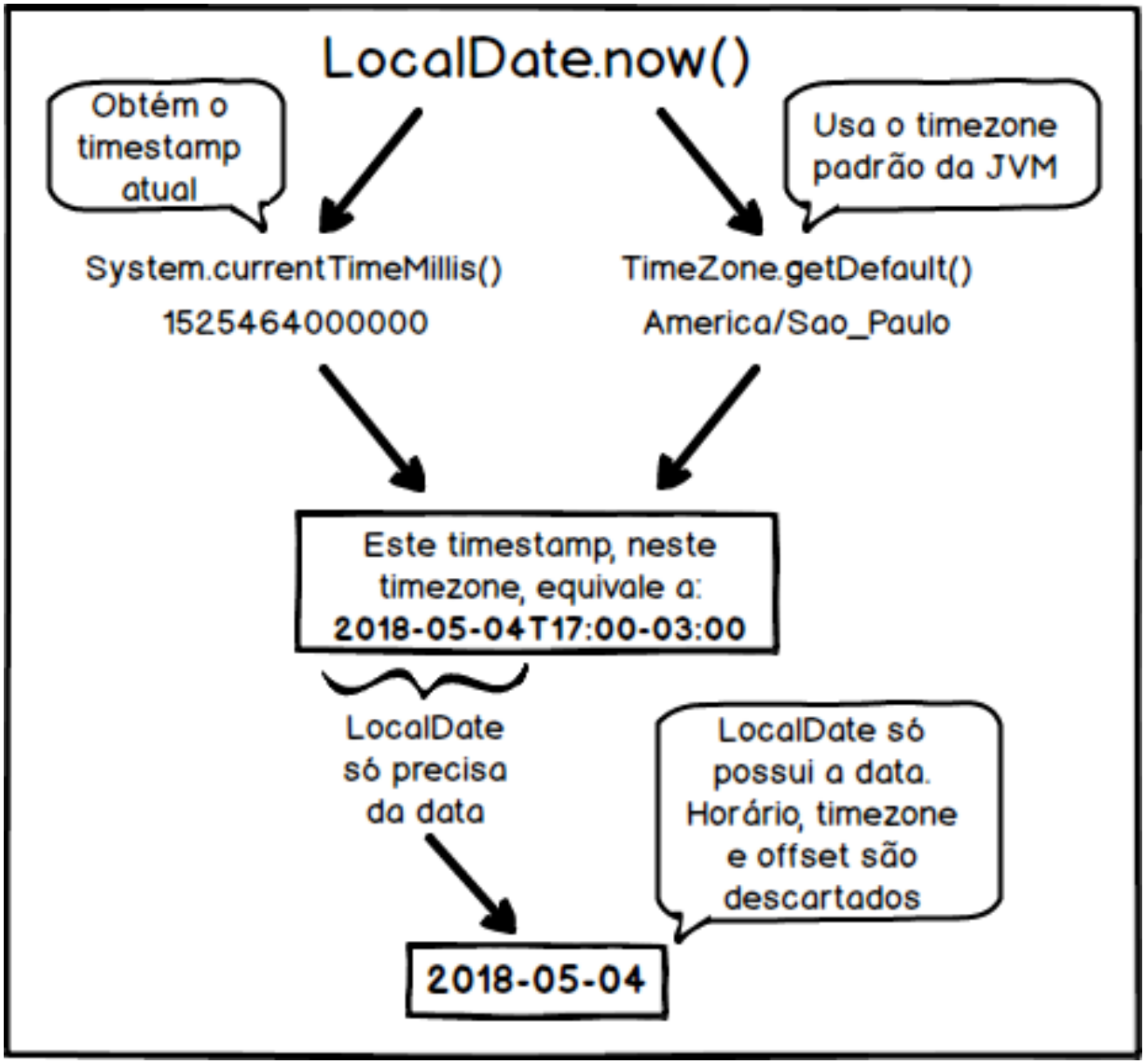

Figura 13.1: LocalDate.now() usa o timezone padrão da JVM para obter os valores da data, mas depois descarta este timezone

Ou seja, o método `now()` **usa** um timezone para obter as informações que precisa, mas este timezone **não faz parte** das classes locais: elas não sabem que os seus valores numéricos da data e hora vieram de um timezone. Todas as operações e manipulações feitas nestes objetos não sofrerão nenhuma interferência de timezones (como os *gaps* e *overlaps*, que podem acontecer quando usamos `Date` e `Calendar`).

Existem também opções para retornar a data e hora atual em um timezone específico (e assim não depender do timezone padrão configurado na JVM), que veremos nos próximos capítulos.

now() sempre retorna os nanossegundos?

Apesar de a API `java.time` trabalhar com precisão de nanossegundos, o método `now()` não vai necessariamente retornar o valor da fração de segundo com as 9 casas decimais.

A implementação do Java 8 é baseada em `System.currentTimeMillis()` , ou seja, possui precisão de milissegundos, então `LocalTime.now()` retorna valores com no máximo 3 casas decimais, como `17:30:25.123` . **Internamente**, o valor da fração de segundo será `123000000` , já que a classe sempre guarda o valor em nanossegundos. Mas a precisão usada será de milissegundos, ou seja, os 6 últimos dígitos sempre serão zero.

No Java 9 a implementação interna foi alterada para usar o relógio nativo do sistema no qual a JVM está rodando. Como alguns sistemas já estão trabalhando com microssegundos (6 casas decimais) e até mesmo com precisões maiores, `now()` retornará valores com esta precisão. Para ver mais detalhes sobre esta mudança, você pode consultar o *JDK Bug System* (https://bugs.openjdk.java.net/browse/JDK-8164428/).

13.3 Manipulando as datas e horas

Ao criar o `java.time` , foram tomados os devidos cuidados para corrigir os diversos problemas de `Date` e `Calendar` , além de não repetir os mesmos erros destas classes. Podemos notar um destes cuidados quando imprimimos a data:

```
System.out.println(LocalDate.of(2018, 5, 4));
System.out.println(LocalDateTime.of(2018, 5, 4, 17, 0));
```

Internamente, `System.out.println()` chama o método `toString()` do objeto passado. E nas classes do `java.time` , este método sempre retorna uma `String` no formato ISO 8601. Por isso a saída deste código é:

```
2018-05-04
2018-05-04T17:00
```

Repare que na segunda linha há a letra `T` separando a data da hora. Este `T` não faz parte do `LocalDateTime` (não há nenhuma variável de instância com esse valor), mas ele é incluído pelo método `toString()` para que o retorno esteja de acordo com o formato ISO 8601. Usar este formato por padrão foi uma ótima decisão, já que ele é cada vez mais adotado internacionalmente, tanto por sistemas quanto por APIs e linguagens, além de ter o aval do *World Wide Web Consortium* (http://www.w3.org/TR/NOTE-datetime/) e do XKCD (http://xkcd.com/1179/).

Diferente do que acontece com `Date` , nas classes do `java.time` o retorno de `toString()` não é afetado pela configuração da JVM. A saída sempre será a mesma, não importa qual o timezone padrão setado no momento. Não há mais como ser enganado pelo método `toString()` , pois agora ele retorna exatamente as informações que a data possui, sem manipular nem esconder nada.

Mas há um caso em que as informações não são mostradas. Ao imprimir horários, os segundos e frações de segundo são omitidos caso os seus valores sejam zero. Exemplo:

```
System.out.println(LocalTime.of(17, 0));
System.out.println(LocalTime.of(7, 0, 45));
System.out.println(LocalTime.of(0, 0, 0, 123000000));
```

A saída é:

```
17:00
07:00:45
00:00:00.123
```

Quando a classe tem os campos de horário, o método `toString()` sempre mostra as horas e minutos. Caso ele não mostre os segundos e frações de segundo (ou não mostre todas as 9 casas decimais dos nanossegundos), é porque esses valores são iguais a zero. Bem diferente de `Date`, que nunca mostra os milissegundos, mesmo quando o valor é diferente de zero.

Obtendo informações da data

Na API legada não houve uma preocupação muito grande com a nomenclatura. `Date` (que não é exatamente uma data) possui o método `getDate()`, que retorna o dia do mês, e `getDay()`, que retorna o dia da semana. As constantes de `Calendar` possuem nomes um pouco melhores, mas ainda há alguns meio confusos (como `HOUR` e `HOUR_OF_DAY`). No `java.time` os métodos possuem nomes mais claros, como podemos ver no próximo exemplo:

```
// 4 de maio de 2018
LocalDate data = LocalDate.of(2018, 5, 4);
int diaDoMes = data.getDayOfMonth(); // 4
int ano = data.getYear(); // 2018
Month mes = data.getMonth(); // Month.MAY
int valorNumericoMes = data.getMonthValue(); // 5
```

`getDayOfMonth()` retorna o valor numérico para o dia do mês (`4`), enquanto `getYear()` retorna o valor numérico do ano (`2018`). Já `getMonth()` retorna um `java.time.Month`, que é um `enum` que corresponde ao mês (`Month.MAY`).

Esta é outra melhoria desta API. Enquanto `Calendar` usava constantes numéricas para os meses (`Calendar.JANUARY`, com valor zero, `Calendar.FEBRUARY`, com valor 1 etc.) o `java.time` usa o tipo `java.time.Month`, um `enum` que possui 12 valores, um para cada mês. No código anterior, `getMonth()` retorna `Month.MAY`, que é o valor deste `enum` correspondente ao mês de maio.

Esta abordagem é melhor do que usar constantes do tipo `int`, pois evita que valores inválidos sejam usados em lugares onde um mês

é esperado. O próprio método `of()` pode receber um `Month` no lugar do valor numérico, então a data poderia ser criada com `LocalDate.of(2018, Month.MAY, 4)` .

Se você precisar do valor numérico do mês, pode usar o método `getMonthValue()` , que neste caso retorna `5` . Como alternativa, um `Month` possui o método `getValue()` , que retorna o valor numérico correspondente — ou seja, eu poderia ter usado `data.getMonth().getValue()` , que também retornaria `5` .

De maneira similar, as outras classes também possuem *getters* para seus campos. `LocalTime` possui, por exemplo, `getHour()` e `getMinute()` , que retornam os valores numéricos da hora e minuto, respectivamente. E `LocalDateTime` possui os mesmos *getters* que `LocalDate` e `LocalTime` , já que ele tem os mesmos campos de data e hora destas classes. Consulte a documentação para ver todos os métodos (https://docs.oracle.com/javase/8/docs/api/java/time/package-summary.html/).

Por que getDayOfMonth em vez de simplesmente getDay?

O nome `getDayOfMonth()` ("obter o dia do mês") pode parecer redundante, já que `getDay()` seria o suficiente para obter o dia, não? Mas na verdade existem mais campos com a palavra "dia" no nome, como o dia da semana e o dia do ano:

```
// 4 de maio de 2018
LocalDate data = LocalDate.of(2018, 5, 4);
int diaDoAno = data.getDayOfYear(); // 124
DayOfWeek diaDaSemana = data.getDayOfWeek(); // DayOfWeek.FRIDAY
```

`getDayOfYear()` retorna `124` , pois 4 de maio de 2018 é o centésimo vigésimo quarto dia do ano (se fosse em um ano bissexto, o método retornaria `125`). E `getDayOfWeek()` retorna um `java.time.DayOfWeek` , que é o `enum` que representa os dias da semana. No caso, o retorno foi `DayOfWeek.FRIDAY` , já que 4 de maio de 2018 é uma sexta-feira.

Caso você precise do valor numérico do dia da semana, `DayOfWeek` possui o método `getValue()` , mas os valores retornados são diferentes das constantes de `Calendar` . Isso acontece porque `DayOfWeek` segue a definição da norma ISO 8601, na qual a semana começa na segunda-feira – o primeiro dia da semana pode variar de acordo com o lugar, norma, cultura, religião etc. Não há uma definição única e a ISO 8601 decidiu escolher a segunda-feira (https://softwareengineering.stackexchange.com/q/170496/).

Por isso, `DayOfWeek.MONDAY.getValue()` retorna 1, `DayOfWeek.TUESDAY.getValue()` retorna 2, e assim por diante. Por outro lado, `Calendar.SUNDAY` tem o valor 1, `Calendar.MONDAY` tem o valor 2 etc. Muita atenção quando for converter valores numéricos de dias da semana entre as duas APIs.

Apesar de terem um método *getter*, os valores do dia da semana e dia do ano não estão efetivamente armazenados na instância de `LocalDate` . Internamente, ela só possui os valores do dia, mês e ano, e todos os outros campos são calculados a partir destes três.

E como eu mudo os valores?

Outra característica importante do `java.time` é que todas as suas classes são imutáveis: não há métodos *setters* para mudar seus campos. Então como fazer para mudar alguma informação da data? Na verdade, você não muda. Em vez disso, você obtém **outra instância com o valor modificado**. Todos os métodos que fazem isso têm nomes que começam com `with` , conforme podemos ver a seguir.

```
// 4 de maio de 2018
LocalDate data = LocalDate.of(2018, 5, 4);
// mudar o dia do mês para 1 (a variável "data" continua sendo dia 4)
LocalDate primeiroDeMaio = data.withDayOfMonth(1);
```

O método `withDayOfMonth()` retorna **outro** `LocalDate` , com o dia do mês modificado para o valor que for passado (no caso, `1`). Com isso, a

variável `primeiroDeMaio` terá o valor correspondente a 1 de maio de 2018, enquanto a variável `data` continua sendo 4 de maio de 2018.

É possível encadear várias chamadas sucessivas dos métodos `withXXX` para mudar vários campos de uma só vez. Por exemplo, se temos um `LocalDateTime` e queremos mudar o dia do mês para 1 e o ano para 2015, e também mudar o horário para 10:30, poderíamos fazer assim:

```
// 2018-05-04T17:00:35.123
LocalDateTime dataHora = LocalDateTime.of(2018, 5, 4, 17, 0, 35,
123000000);
// mudar para 2015-05-01T10:30:35.123
LocalDateTime primeiroDeMaio = dataHora
    // muda o dia do mês para 1
    .withDayOfMonth(1)
    // muda o ano para 2015
    .withYear(2015)
    // mudar hora para 10
    .withHour(10)
    // mudar minuto para 30
    .withMinute(30);
```

Os campos cujos respectivos métodos `with` não foram chamados (o mês, os segundos e os nanossegundos) permanecem os mesmos, e o resultado é `2015-05-01T10:30:35.123` .

13.4 Ajustes customizados com TemporalAdjuster

Os métodos `withXXX` retornam outra instância com apenas um dos campos modificado. É uma alteração bem simples e direta, mas e se quisermos fazer algum ajuste mais complicado, como mudar uma data para o último dia do mês? Não é algo tão direto quanto chamar `withDayOfMonth()` , pois os meses podem ter 28, 29, 30 ou 31 dias, e

teríamos que testar algumas condições para saber qual o valor correto a ser usado. Felizmente, o `java.time` já faz isso para nós.

Todas as classes de data e hora possuem um método `with()` que recebe como parâmetro a interface `java.time.temporal.TemporalAdjuster`. A ideia desta interface é que ela contenha toda a lógica para manipular a data ou hora (usando quaisquer regras que sejam necessárias) e retorne outra instância com os valores modificados. Com isso, podemos implementar lógicas mais complexas do que mudar apenas o valor de um campo.

Posteriormente veremos como criar nosso próprio `TemporalAdjuster`. Por enquanto, vamos usar os que já estão implementados na API. Na classe `java.time.temporal.TemporalAdjusters` há vários métodos estáticos que retornam um `TemporalAdjuster` predefinido.

Para ajustar uma data para o último dia do mês, podemos usar o `TemporalAdjuster` retornado pelo método `lastDayOfMonth()`. No código a seguir eu uso este método com um `import static`, para deixar o código mais legível:

```
import static java.time.temporal.TemporalAdjusters.lastDayOfMonth;

// 4 de maio de 2018
LocalDate data = LocalDate.of(2018, Month.MAY, 4);
// mudar para o último dia do mês
LocalDate ultimoDiaDeMaio = data.with(lastDayOfMonth());
```

Com isso, a variável `ultimoDiaDeMaio` corresponderá a 31 de maio de 2018. O `TemporalAdjuster` retornado por `lastDayOfMonth()` verifica qual o último dia do mês, usando todas aquelas regras que conhecemos: alguns meses têm 30 dias, fevereiro só tem 29 dias em anos bissextos etc.

Repare como o `import static` deixa o código mais legível: em inglês, estamos literalmente dizendo "data com o último dia do mês".

Não confunda a classe `TemporalAdjusters` (no plural) com a interface `TemporalAdjuster` (no singular). `TemporalAdjusters` é uma classe utilitária

que possui vários métodos estáticos. E cada um destes métodos retorna uma implementação de `TemporalAdjuster` que pode ser usada no método `with()` .

Há outros métodos bem úteis que podem ser usados, como `firstDayOfMonth()` (primeiro dia do mês), `firstDayOfNextMonth()` (primeiro dia do mês seguinte), `firstDayOfNextYear()` (primeiro dia do ano seguinte), entre outros. Consulte a documentação para mais detalhes (https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalAdjusters.html/).

Ajustes envolvendo dias da semana

Na minha opinião, os métodos mais interessantes de `TemporalAdjusters` são os que retornam um `TemporalAdjuster` que trabalha com os dias da semana, porque eles implementam algoritmos que apesar de não serem tão difíceis assim, são meio "chatos" de se fazer.

Por exemplo, se a partir de uma data eu quiser saber qual a próxima sexta-feira, basta usar o método `next()` , passando como parâmetro o `DayOfWeek` correspondente à sexta-feira. No próximo exemplo eu também uso o `import static` com `DayOfWeek` e `Month` , para deixar o código ainda mais legível:

```
import static java.time.temporal.TemporalAdjusters.next;
import static java.time.DayOfWeek.FRIDAY;
import static java.time.Month.MAY;

// 4 de maio de 2018
LocalDate data = LocalDate.of(2018, MAY, 4);
// próxima sexta-feira: 2018-05-11
LocalDate proximaSexta = data.with(next(FRIDAY));
```

O resultado será 11 de maio de 2018, que corresponde à sexta-feira imediatamente posterior ao dia 4 de maio de 2018. O método `next()` aceita qualquer dia da semana como parâmetro (portanto, qualquer

valor de `DayOfWeek`). Também é possível saber qual a próxima sexta-feira, mas retornar a mesma data caso ela já seja uma sexta, usando o método `nextOrSame()` .

Há também métodos similares para obter o dia da semana anterior, em vez do próximo: `previous()` e `previousOrSame()` . Eles funcionam da mesma maneira que `next()` e `nextOrSame()` : basta passar um `DayOfWeek` como parâmetro e os métodos vão buscar a ocorrência anterior do dia da semana correspondente.

Outro ajuste interessante é obter a ocorrência de um dia da semana em um mês. Por exemplo, para obter a terceira quinta-feira do mês, posso usar o método `dayOfWeekInMonth()` , conforme mostra o próximo exemplo (os `import static` foram omitidos do código):

```
// 4 de maio de 2018
LocalDate data = LocalDate.of(2018, MAY, 4);
// terceira quinta-feira do mês: 2018-05-17
LocalDate quintaFeira = data.with(dayOfWeekInMonth(3, THURSDAY));
```

Para completar, há os métodos `firstInMonth(DayOfWeek)` e `lastInMonth(DayOfWeek)` , que retornam, respectivamente, a primeira e última ocorrência do dia da semana:

```
// 4 de maio de 2018
LocalDate data = LocalDate.of(2018, MAY, 4);
// primeiro sábado do mês: 2018-05-05
LocalDate primeiroSabado = data.with(firstInMonth(SATURDAY));
// último sábado do mês: 2018-05-26
LocalDate ultimoSabado = data.with(lastInMonth(SATURDAY));
```

`firstInMonth(SATURDAY)` é equivalente a `dayOfWeekInMonth(1, SATURDAY)` , enquanto `lastInMonth(SATURDAY)` é equivalente a `dayOfWeekInMonth(-1, SATURDAY)` . É isso mesmo, `dayOfWeekInMonth()` aceita valores negativos, então `-1` retorna o último dia da semana no mês, `-2` retorna o penúltimo e assim por diante.

Cuidados ao usar um TemporalAdjuster

Cada `TemporalAdjuster` manipula determinados campos de um objeto, mas devemos nos certificar de que o objeto em questão possui tais campos. Por exemplo, se usarmos `firstDayOfMonth()` com um `LocalTime` :

```
LocalTime horario = LocalTime.now().with(firstDayOfMonth());
```

Neste código, estamos ajustando um `LocalTime` para o primeiro dia do mês. Porém, esta classe só possui os campos relativos ao horário, sem nenhuma noção de dia, mês ou ano. Por isso, este código lança uma exceção:

```
java.time.temporal.UnsupportedTemporalTypeException: Unsupported field: DayOfMonth
```

A mensagem diz que o campo `DayOfMonth` não é suportado. E de fato, este campo (o dia do mês) não está presente na classe `LocalTime` , pois esta só possui horas, minutos, segundos e nanossegundos. Ao usar um `TemporalAdjuster` , é importante verificar na documentação quais campos serão manipulados.

Outras implementações de TemporalAdjuster

As classes de data e hora também são implementações de `TemporalAdjuster` , o que significa que elas próprias podem ser passadas para o método `with()` . Sendo assim, é possível mudar o horário de um `LocalDateTime` passando um `LocalTime` para o método `with()` :

```
// 2018-05-04T10:00:35.123456
LocalDateTime dataHora = LocalDateTime.of(2018, 5, 4, 10, 0, 35, 123456000);
// mudar horário para 17:30 -> 2018-05-04T17:30
dataHora = dataHora.with(LocalTime.of(17, 30));
```

Lembrando que o método `with()` sempre retorna outra instância, por isso é necessário atribuir o seu retorno em uma variável. No caso, eu atribuí para a mesma variável, o que é uma maneira de "mudar" o seu valor.

Note também que `LocalTime.of(17, 30)` cria um `LocalTime` com horário igual a `17:30:00.000000000` (o valor dos segundos e nanossegundos não foi passado para `of()` , então estes são automaticamente setados para zero). Por isso, o `LocalDateTime` muda de `2018-05-04T10:00:35.123456` para `2018-05-04T17:30` .

Da mesma forma, eu posso mudar a data passando um `LocalDate` para o método `with()` :

```
// 2018-05-04T00:00
LocalDateTime dataHora = LocalDateTime.of(2018, 5, 4, 0, 0);
// mudar data para 1 de janeiro de 2001 -> 2001-01-01T00:00
dataHora = dataHora.with(LocalDate.of(2001, 1, 1));
```

Um ponto de atenção é que se você passar um `LocalDate` ou `LocalTime` para o método `with()` , **todos** os respectivos campos de data ou hora serão mudados. Ou seja, passando um `LocalDate` , o dia, mês e ano serão mudados, e passando um `LocalTime` , a hora, minuto, segundo e fração de segundos serão mudados.

Caso você queira mudar apenas um dos campos (somente o dia, ou só os minutos, por exemplo), use o respectivo método `withXXX()` (como `withDayOfMonth()` ou `withMinute()`).

13.5 Comparando as datas e horas

Comparar duas datas é relativamente simples. Para saber se uma data ocorre antes ou depois de outra, basta usar os métodos `isBefore()` e `isAfter()` :

```
// 4 de maio de 2018
LocalDate maio = LocalDate.of(2018, 5, 4);
// 10 de janeiro de 2018
LocalDate janeiro = LocalDate.of(2018, 1, 10);
boolean maioDepoisDeJaneiro = maio.isAfter(janeiro); // true
boolean maioAntesDeJaneiro = maio.isBefore(janeiro); // false
```

No caso, a variável `maioDepoisDeJaneiro` será `true` , pois a data contida em `maio` ocorre depois da data contida em `janeiro` . Já `maioAntesDeJaneiro` será `false` , pelo mesmo motivo.

Os métodos `isBefore()` e `isAfter()` também estão presentes na classe `LocalTime` :

```
LocalTime dezDaNoite = LocalTime.of(22, 0);
LocalTime tresDaManha = LocalTime.of(3, 0);
boolean antes = dezDaNoite.isBefore(tresDaManha); // false
```

`LocalTime` não possui nenhuma informação sobre o dia, mês ou ano, portanto usa apenas os valores numéricos do horário para fazer a comparação. Além disso, a meia-noite é considerada o menor valor possível para um `LocalTime` . Por isso 10 da noite (22:00) ocorre depois das 3 da manhã (03:00) e o valor da variável `antes` será `false` .

E se quisermos comparar 10 da noite de um dia com 3 da manhã de outro dia? Nesse caso, precisamos de uma classe que possui tanto a data quanto o horário. Ou seja, temos que usar um `LocalDateTime` :

```
// 2018-05-04T22:00
LocalDateTime dataHora1 = LocalDateTime.of(2018, 5, 4, 22, 0);
// 2018-05-05T03:00
LocalDateTime dataHora2 = LocalDateTime.of(2018, 5, 5, 3, 0);
boolean antes = dataHora1.isBefore(dataHora2); // true
```

Agora o resultado de `isBefore()` é `true` , pois `dataHora1` corresponde às 10h da noite do dia 4 de maio, que ocorreu antes de `dataHora2` (3 da manhã do dia 5).

Verificar igualdade

Para verificar se duas datas são iguais, basta usar o método `equals()` , que verifica se todos os valores numéricos são os mesmos:

```
LocalDate data1 = LocalDate.of(2018, 5, 4);
LocalDate data2 = LocalDate.of(2018, Month.MAY, 4);
boolean iguais = data1.equals(data2); // true
```

Para criar `data2` eu usei `Month.MAY` em vez de `5`, mas internamente a classe `LocalDate` guarda somente o valor numérico do mês. Por isso, as duas datas são iguais e o resultado é `true`.

Um detalhe importante é que `equals()` só compara instâncias da mesma classe. Se eu comparar um `LocalDate` com um `LocalDateTime`, o resultado será `false`, mesmo se os campos da data forem iguais:

```
// 2018-05-04
LocalDate data = LocalDate.of(2018, 5, 4);
// 2018-05-04T17:00
LocalDateTime dataHora = data.atTime(17, 0);
boolean iguais = data.equals(dataHora); // false
```

Repare que usei o método `atTime()`, que combina a data do `LocalDate` com um horário, para formar um `LocalDateTime`. A seguir eu uso `equals()` para comparar os objetos e o resultado é `false`, já que eles são de classes diferentes (mesmo que a data seja igual).

Se eu quiser comparar somente a data, tenho que converter o `LocalDateTime` para um `LocalDate`, usando o método `toLocalDate()`:

```
// 2018-05-04
LocalDate data = LocalDate.of(2018, 5, 4);
// 2018-05-04T17:00
LocalDateTime dataHora = data.atTime(17, 0);
// compara dois LocalDates
boolean iguais = data.equals(dataHora.toLocalDate()); // true
```

O método `toLocalDate()` retorna um `LocalDate` com os mesmos valores do dia, mês e ano do `LocalDateTime`. Assim, o método `equals()` está comparando duas instâncias de `LocalDate`, e como elas são iguais (possuem os mesmos valores numéricos para o dia, mês e ano), o resultado é `true`.

Existe também o método `toLocalTime()`, que retorna um `LocalTime` contendo os mesmos valores do horário do `LocalDateTime`.

As classes `LocalDate` e `LocalDateTime` também possuem o método `isEqual()`, que serve para comparar datas em outros calendários. Há vários calendários em uso no mundo, e em cada um deles os valores do dia, mês e ano são diferentes, apesar de todas as datas ocorrerem no mesmo momento (enquanto estamos em 2018, o calendário islâmico está em 1439, por exemplo). A menos que você precise trabalhar com outros calendários, usar `equals()` já é o suficiente. As classes da API que representam datas em outros calendários são mencionadas no tutorial da Oracle (https://docs.oracle.com/javase/tutorial/datetime/iso/nonIso.html/) e estão além do escopo deste livro.

Ordenação

Todas as classes de data e hora do `java.time` implementam a interface `java.lang.Comparable`. Por isso, qualquer lista contendo estas classes pode ser ordenada, conforme mostra o próximo exemplo, que usa as classes de *collections* do pacote `java.util`:

```
List<LocalDate> lista = Arrays.asList(
    LocalDate.of(2018, 5, 4),    // 2018-05-04
    LocalDate.of(1995, 5, 4),    // 1995-05-04
    LocalDate.of(2018, 1, 20));  // 2018-01-20
Collections.sort(lista);
System.out.println(lista);
```

A ordenação é feita de acordo com a ordem cronológica. O resultado é:

```
[1995-05-04, 2018-01-20, 2018-05-04]
```

Vale lembrar que há diferenças na forma com que cada tipo é ordenado. `LocalTime` só considera o horário (pois esta classe não tem informações da data) e `LocalDateTime` considera tanto a data quanto a hora.

E os timezones?

Este capítulo focou nas classes locais, usando-as para explicar algumas características básicas da API. Só que estas classes não possuem timezones e nem offsets. Para isso, há outros tipos disponíveis, que veremos nos próximos capítulos.

Capítulo 14
Trabalhando com timezones e offsets

Quando você precisar trabalhar corretamente com timezones, use a classe `java.time.ZonedDateTime` , que possui os mecanismos para lidar com todas aquelas regras malucas de timezones que já vimos ao longo do livro. Vamos ver com mais detalhes como ela funciona.

14.1 Funcionamento básico de ZonedDateTime

Existem funcionalidades que são iguais às que vimos anteriormente com as classes locais. Uma delas é o método `now()` , que retorna a data e hora atual, usando o timezone padrão da JVM para definir os valores numéricos da data e hora. Mas como `ZonedDateTime` também possui informações do timezone, esta informação não é descartada (como acontece com as classes locais). Exemplo:

```
// 2018-05-04T17:00-03:00 - America/Sao_Paulo
ZonedDateTime agora = ZonedDateTime.now();
System.out.println(agora);
```

A saída deste código é:

```
2018-05-04T17:00-03:00[America/Sao_Paulo]
```

Há dois pontos interessantes a serem notados. O primeiro é que, além de obter a data e hora atual no timezone padrão da JVM, o offset usado por este timezone naquele instante é calculado, e este valor também faz parte do `ZonedDateTime` . Por isso, a saída mostra `-03:00` . A princípio isto pode parecer redundante, mas não é. O timezone contém o histórico de offsets de uma região (que pode variar com o tempo) e é importante sabermos qual o offset usado em determinado momento.

O segundo ponto é que o método `toString()` também mostra o nome do timezone entre colchetes (`[America/Sao_Paulo]`). Isto é muito importante, pois vários timezones podem usar o mesmo offset simultaneamente, e se fosse mostrado somente `-03:00` , não saberíamos qual o timezone sendo usado.

Vale lembrar que o formato ISO 8601 permite apenas offsets e não define uma forma de representar o nome do timezone. A API `java.time` , por outro lado, estende este formato, mostrando todas as informações, evitando qualquer dúvida quanto ao conteúdo do `ZonedDateTime` .

Obter a data e hora atual em outro timezone

O método `now()` usa o timezone padrão da JVM para obter os valores numéricos da data e hora atual, além do offset utilizado naquele momento.

E se quisermos usar outro timezone? Uma alternativa seria mudar o timezone padrão com o método `TimeZone.setDefault()` , que já vimos anteriormente. Mas também vimos os problemas que isso pode causar. Por isso, uma alternativa melhor é usar a classe `java.time.ZoneId` .

`ZoneId` representa um timezone e pode ser criado através do método `of()` , que recebe como parâmetro uma `String` contendo um identificador da IANA. A seguir, este `ZoneId` é passado para o método `now()` , que usará este timezone em vez do padrão da JVM. Por exemplo, para obtermos a data e hora atual no timezone `Asia/Tokyo` , basta fazer:

```
// data/hora atual no timezone Asia/Tokyo
ZonedDateTime agora = ZonedDateTime.now(ZoneId.of("Asia/Tokyo"));
System.out.println(agora);
```

Com isso, o timezone `Asia/Tokyo` será usado para calcular os valores da data, hora e offset atual, independente de qual for o timezone padrão da JVM. A saída deste código é:

```
2018-05-05T05:00+09:00[Asia/Tokyo]
```

Obtendo o timezone padrão da JVM

Caso você queira deixar explícito no seu código que o timezone padrão da JVM deve ser usado, basta usar `ZoneId.systemDefault()`. O retorno deste método é diretamente influenciado por `TimeZone.setDefault()`, como podemos ver no próximo exemplo:

```
TimeZone.setDefault(TimeZone.getTimeZone("America/Los_Angeles"));
System.out.println(ZoneId.systemDefault());
TimeZone.setDefault(TimeZone.getTimeZone("Europe/London"));
System.out.println(ZoneId.systemDefault());
```

`ZoneId.systemDefault()` sempre usa o timezone padrão que estiver configurado no momento em que ele é chamado. Além disso, seu método `toString()` retorna o identificador da IANA. Por isso a saída deste código é:

```
America/Los_Angeles
Europe/London
```

Posso usar ZoneId com classes que não têm timezone

O `ZoneId` também pode ser usado no método `now()` das classes locais, para que elas obtenham os valores da data ou hora em um timezone específico (em vez de usar o timezone padrão). Por exemplo:

```
// data de hoje no timezone America/Sao_Paulo: 2018-05-04
LocalDate hojeSP = LocalDate.now(ZoneId.of("America/Sao_Paulo"));
// data de hoje no timezone Asia/Tokyo: 2018-05-05
LocalDate hojeTokyo = LocalDate.now(ZoneId.of("Asia/Tokyo"));
```

`LocalDate.now(ZoneId)` usa o timezone representado pelo `ZoneId` para saber quais os valores atuais do dia, mês e ano. Mas como `LocalDate` não guarda nenhuma referência ao timezone, o `ZoneId` é descartado.

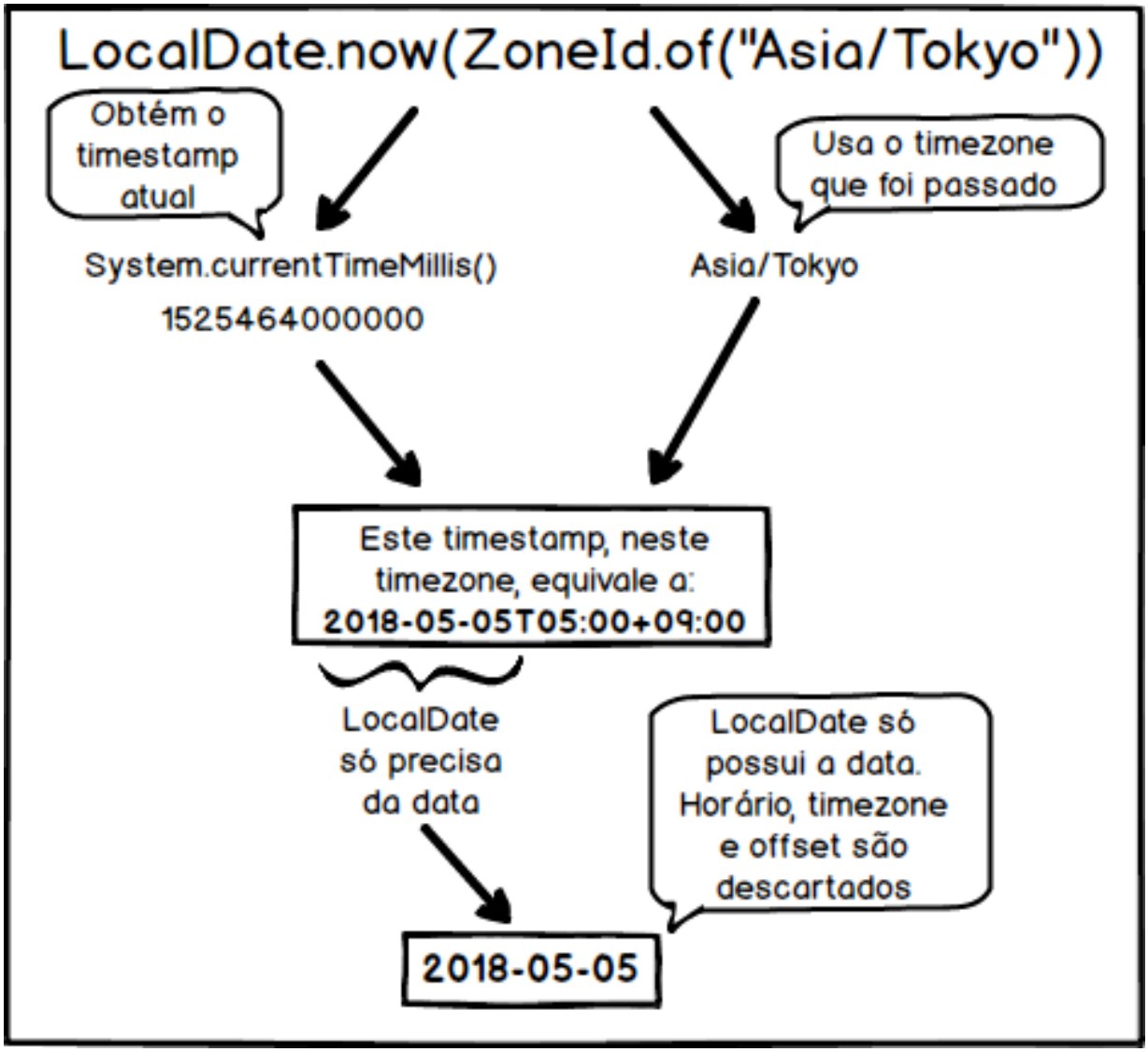

Figura 14.1: LocalDate.now(ZoneId) usa o timezone para obter os valores da data e depois descarta o ZoneId

O mesmo acontece com `LocalTime` e `LocalDateTime` :

```
// hora atual timezone Asia/Tokyo: 05:00
System.out.println(LocalTime.now(ZoneId.of("Asia/Tokyo")));
// data e hora atual timezone Asia/Tokyo: 2018-05-05T05:00
System.out.println(LocalDateTime.now(ZoneId.of("Asia/Tokyo")));
```

Neste exemplo, o `LocalTime` usa o horário atual no timezone `Asia/Tokyo` , que é 05:00, e o restante da informação (data, offset e timezone) é

descartado. Já `LocalDateTime` usa as informações de data e hora, e descarta o offset e o timezone.

ZoneId não aceita ID inválido

Anteriormente vimos que se passarmos um nome de timezone inválido para o método `TimeZone.getTimeZone()` , ele retorna uma instância que corresponde a UTC. Com `ZoneId` isso não acontece, e qualquer nome inválido lança uma exceção. Exemplo:

```
// ID inválido, retorna UTC
TimeZone timeZone = TimeZone.getTimeZone("Id invalido");
// ID inválido, lança exceção
ZoneId zone = ZoneId.of("Id invalido");
```

A chamada de `TimeZone.getTimeZone()` retorna uma instância que corresponde a UTC, enquanto `ZoneId.of()` lança a seguinte exceção:

```
java.time.DateTimeException: Invalid ID for region-based ZoneId, invalid
format: Id invalido
```

Para saber se um ID é válido ou não, você pode fazer um `try/catch` , ou então verificar se o ID faz parte da lista de identificadores reconhecidos pela JVM. Para obter esta lista, use o método `ZoneId.getAvailableZoneIds()` .

Outra diferença é que `TimeZone.getTimeZone()` aceita algumas abreviações, como "IST" e "EST". Mas estas abreviações são ambíguas e nem sempre o retorno vai ser o esperado: "IST" é usado na Índia, Israel e Irlanda — o método retorna o timezone de qual destes lugares?

`ZoneId.of()` , por sua vez, não aceita estas abreviações e lança um `DateTimeException` . Mas é possível usar um `java.util.Map` para mapear as abreviações para algum identificador da IANA, como mostra o próximo exemplo:

```
// minhas escolhas arbitrárias para as abreviações
Map<String, String> abreviacoes = new HashMap<>();
```

```
// EST é mapeado para New York
abreviacoes.put("EST", "America/New_York");
// IST é mapeado para Índia
abreviacoes.put("IST", "Asia/Kolkata");
// ... coloque no Map quantos valores você precisar
// usar o map para obter o timezone - Asia/Kolkata
ZoneId zone = ZoneId.of("IST", abreviacoes);
```

Com isto, o método `of()` verifica se o `Map` possui alguma entrada para `"IST"` e usa o valor correspondente para construir o `ZoneId` . No caso, o timezone criado será `Asia/Kolkata` .

Vale lembrar que qualquer escolha como esta será arbitrária e sempre dependerá dos seus casos de uso. Um sistema que só tem usuários em Israel provavelmente mapearia `"IST"` para `Asia/Jerusalem` , por exemplo. Não há uma regra geral para escolher o timezone adequado para cada abreviação.

14.2 Como ZonedDateTime lida com gaps e overlaps

O modo como os timezones funcionam faz com que surjam situações estranhas como os *gaps* e *overlaps*. Vamos ver como `ZonedDateTime` lida com estas situações. Para os próximos exemplos usarei o método `of()` , que recebe os valores numéricos da data e hora, além do `ZoneId` , para criar o `ZonedDateTime` correspondente.

O funcionamento básico deste método é verificar qual o offset usado pelo timezone naquele dia e horário, e usar esses valores para criar o `ZonedDateTime` . Mas há fatores como o horário de verão (ou qualquer outra mudança de offset), que podem trazer resultados diferentes do esperado.

Lidando com os gaps

Vamos ver como a classe se comporta quando há um *DST gap*, ou seja, quando o horário de verão começa e uma hora é pulada. Para isso, vamos usar o timezone `America/Sao_Paulo` e a transição de outubro de 2017:

```
ZoneId zone = ZoneId.of("America/Sao_Paulo");
// 15 de outubo de 2017, meia-noite, horário de verão começa em São Paulo
ZonedDateTime z = ZonedDateTime.of(2017, 10, 15, 0, 0, 0, 0, zone);
```

No timezone `America/Sao_Paulo` , no dia 15 de outubro de 2017, à meia-noite, os relógios foram adiantados em uma hora, diretamente para 01:00. Isso quer dizer que uma hora foi pulada e todos os minutos entre 00:00 e 00:59 não existem neste dia, neste timezone.

Ou seja, esta combinação de data, hora e timezone (15 de outubro de 2017, meia-noite, `America/Sao_Paulo`) é inválida. Por isso, ela é ajustada para o próximo horário válido — no caso, 01:00. E como agora está em horário de verão, o offset é ajustado para `-02:00` e o resultado é `2017-10-15T01:00-02:00[America/Sao_Paulo]` .

Esta abordagem é interessante porque simula uma pessoa verificando que o relógio está marcando meia-noite, percebendo que esqueceu de adiantá-lo e ajustando-o para 01:00. O ajuste é feito adicionando uma hora, independente dos demais campos: se eu tentasse criar um `ZonedDateTime` no mesmo dia e timezone, mas com horário igual a 00:30, ele seria ajustado para 01:30.

Mas lembre-se de que nem todo *gap* é de uma hora. No timezone `Australia/Lord_Howe` , durante o horário de verão, os relógios são adiantados em meia hora. Ou seja, ao tentar criar um `ZonedDateTime` em uma data e hora que está em um *gap* deste timezone, o ajuste é feito adicionando 30 minutos:

```
ZoneId zone = ZoneId.of("Australia/Lord_Howe");
// ajustado para 2018-10-07T02:40+11:00[Australia/Lord_Howe]
ZonedDateTime z = ZonedDateTime.of(2018, 10, 7, 2, 10, 0, 0, zone);
```

No timezone `Australia/Lord_Howe` , em 7 de outubro de 2018, às 02:00, começa o horário de verão, e os relógios são adiantados em 30

minutos. Ou seja, os minutos entre 02:00 e 02:30 não existem neste dia, neste timezone (um *gap* de meia hora). Por isso 02:10 é ajustado para 30 minutos depois (02:40) e o resultado é `2018-10-07T02:40+11:00[Australia/Lord_Howe]`.

Lidando com overlaps

Agora vamos ver como `ZonedDateTime` se comporta quando há um *DST overlap*, ou seja, quando o horário de verão termina e uma hora local existe duas vezes. Para isso, vamos usar o timezone `America/Sao_Paulo` e a transição de fevereiro de 2018.

No timezone `America/Sao_Paulo`, em 18 de fevereiro de 2018, à meia-noite, os relógios são atrasados em uma hora, de volta para as 23:00 do dia 17. Ou seja, nesta data e neste timezone, todos os minutos entre 23:00 e 23:59 existem duas vezes: uma no offset `-02:00` (horário de verão) e outra no offset `-03:00` (horário "normal").

O que acontece então se eu tentar criar um `ZonedDateTime` usando o dia 17 de fevereiro de 2018 às 23:00 no timezone `America/Sao_Paulo`? Ele usará a primeira ou segunda ocorrência deste horário? Vamos ver o que acontece no exemplo a seguir:

```
ZoneId zone = ZoneId.of("America/Sao_Paulo");
// 17 de fevereiro de 2018, às 23:00
ZonedDateTime zdt = ZonedDateTime.of(2018, 2, 17, 23, 0, 0, 0, zone);
```

Por padrão, é escolhida a primeira ocorrência das 23:00, ou seja, antes de terminar o horário de verão. Por isso o offset ainda é `-02:00` e o resultado é `2018-02-17T23:00-02:00[America/Sao_Paulo]`.

E se quisermos a segunda ocorrência das 23:00? Neste caso, podemos usar o método `withLaterOffsetAtOverlap()`, que retorna outra instância de `ZonedDateTime` com os valores ajustados para a segunda ocorrência:

```
// depois de terminar o horário de verão
ZonedDateTime depoisHv = zdt.withLaterOffsetAtOverlap();
```

Na segunda vez que ocorre a hora local 23:00, o horário de verão já terminou e o offset é `-03:00` . Por isso o resultado é `2018-02-17T23:00-03:00[America/Sao_Paulo]` .

Existe também o método `withEarlierOffsetAtOverlap()` , que retorna a primeira ocorrência. Ele é útil caso você queira ter certeza que o `ZonedDateTime` vai se referir à primeira ocorrência de um horário local, em caso de *overlap*. No caso do método `of()` , já sabemos que por padrão ele usa a primeira ocorrência, mas caso seu código receba um `ZonedDateTime` que foi criado em outro lugar e você não tem certeza quanto ao valor, pode usar este método para ajustá-lo.

Os métodos `withEarlierOffsetAtOverlap()` e `withLaterOffsetAtOverlap()` permitem que você tenha um controle maior sobre situações de *overlap*, quando um horário local é ambíguo. A vantagem destes métodos é que você não precisa saber se o *overlap* é de uma hora, meia hora ou qualquer outro valor. Os métodos se baseiam nas regras do timezone em questão e calculam o offset correto para cada caso.

E quando uma data e hora não fazem parte de um *overlap*, ambos os métodos retornam o mesmo `ZonedDateTime` , com os valores inalterados. Portanto não há problema em usá-los nos casos em que não há *overlap*.

14.3 Manipulando o ZonedDateTime

Assim como as demais classes já vistas, `ZonedDateTime` também possui os métodos `withXXX()` , que retornam outra instância com os valores modificados. Vários deles são os mesmos que já vimos nas outras classes, como `withDayOfMonth()` , que retorna outra instância com o dia do mês modificado, `withMinute()` , que retorna outra instância com o valor dos minutos modificado etc.

Também há o método `with()` , que recebe uma instância de qualquer um dos tipos locais (ou qualquer outro `TemporalAdjuster`), do mesmo modo que já vimos no capítulo anterior. Com isso, eu posso mudar vários campos de data ou hora de uma só vez:

```
// 2018-05-04T17:00-03:00[America/Sao_Paulo]
ZonedDateTime dataHoraSp = ZonedDateTime.now();
System.out.println("antes :" + dataHoraSp);
dataHoraSp = dataHoraSp
    // mudar a data para 12 de janeiro de 2017
    .with(LocalDate.of(2017, 1, 12))
    // mudar a hora para meio-dia
    .with(LocalTime.NOON);
System.out.println("depois:" + dataHoraSp);
```

Primeiro, o método `now()` cria um `ZonedDateTime` com a data e hora atual, no timezone padrão da JVM (`2018-05-04T17:00-03:00[America/Sao_Paulo]`). Em seguida, eu passo um `LocalDate` para o método `with()` , mudando a data para 12 para janeiro de 2017. Como nesta data o timezone `America/Sao_Paulo` está em horário de verão, o offset é ajustado para `-02:00` .

Depois eu passo um `LocalTime` para o método `with()` . Como eu passei a constante `LocalTime.NOON` (que equivale a meio-dia), o horário do `ZonedDateTime` é mudado para este valor. Mas isso não muda o offset, pois nesta data e hora (12 de janeiro de 2017 ao meio-dia) o timezone `America/Sao_Paulo` ainda continua no horário de verão. A saída é:

```
antes :2018-05-04T17:00-03:00[America/Sao_Paulo]
depois:2017-01-12T12:00-02:00[America/Sao_Paulo]
```

Qualquer alteração feita em um `ZonedDateTime` pode resultar em uma mudança de offset. O novo valor da data e hora é sempre verificado junto ao timezone, para que ele saiba qual offset deve ser usado naquele momento.

Se o resultado cair em um *gap*, são feitos os devidos ajustes para o próximo horário válido, como já vimos anteriormente. Se o resultado

cair em um *overlap*, o padrão é usar a primeira ocorrência, mas você pode usar os métodos `withEarlierOffsetAtOverlap()` e `withLaterOffsetAtOverlap()` para controlar qual das ocorrências será usada.

Obtendo informações do ZonedDateTime

Os *getters* que esta classe possui são similares aos das classes locais: `getMonth()` retorna o `java.time.Month` correspondente ao mês, `getHour()` que retorna o valor numérico das horas etc.

Também há métodos que convertem para os tipos locais, como `toLocalDate()` , `toLocalTime()` e `toLocalDateTime()` , cujos nomes são autoexplicativos — para saber o que cada um retorna, basta remover o `to` do início do nome.

Além disso, `ZonedDateTime` possui um timezone e um offset, e estas informações também podem ser obtidas por *getters*:

```
// 2018-05-04T17:00-03:00[America/Sao_Paulo]
ZonedDateTime zdt = ZonedDateTime.now();
// obter o timezone
ZoneId zone = zdt.getZone();
System.out.println(zone);
// obter o offset
ZoneOffset offset = zdt.getOffset();
System.out.println(offset);
```

`getZone()` retorna o `ZoneId` que foi usado para criar o `ZonedDateTime` . Como usamos o método `now()` sem parâmetros, foi usado o timezone padrão da JVM (`America/Sao_Paulo`). E `getOffset()` retorna uma instância de `java.time.ZoneOffset` , que é a classe que representa um offset.

A diferença entre estas classes é a mesma que existe entre timezones e offsets. Enquanto um offset (`ZoneOffset`) é simplesmente um número fixo que representa a diferença com relação a UTC, o timezone (`ZoneId`) possui uma lista de todos os offsets usados por determinada região ao longo da história.

A saída deste código é:

```
America/Sao_Paulo
-03:00
```

Quando o `ZoneId` é impresso, o resultado é o seu identificador da IANA (`America/Sao_Paulo`). Já o offset é impresso no formato ISO 8601. No caso, o resultado foi `-03:00` (3 horas atrás de UTC), já que este é o offset usado pelo timezone `America/Sao_Paulo` na data e hora atual (4 de maio de 2018, às 17:00).

14.4 Comparando instâncias de ZonedDateTime

Há dois métodos para saber se duas instâncias de `ZonedDateTime` são iguais: `equals()` e `isEqual()` .

O método `equals()` verifica os valores numéricos da data e hora, o offset e o timezone, e só retorna `true` se todos forem iguais. Já o método `isEqual()` verifica se as instâncias correspondem ao mesmo instante (ao mesmo ponto na linha do tempo). No próximo exemplo é mostrada a diferença entre esses métodos:

```
// 2018-05-04T17:00-03:00[America/Sao_Paulo]
ZoneId spZone = ZoneId.of("America/Sao_Paulo");
ZonedDateTime dataSP = ZonedDateTime.of(2018, 5, 4, 17, 0, 0, 0, spZone);
// 2018-05-05T05:00+09:00[Asia/Tokyo]
ZoneId tokyoZone = ZoneId.of("Asia/Tokyo");
ZonedDateTime dataTokyo = ZonedDateTime.of(2018, 5, 5, 5, 0, 0, 0,
tokyoZone);
// false - valores numéricos da data/hora/offset e/ou timezone diferentes
System.out.println(dataSP.equals(dataTokyo));
// true - correspondem ao mesmo instante
System.out.println(dataSP.isEqual(dataTokyo));
```

A variável `dataSP` corresponde a `2018-05-04T17:00-03:00[America/Sao_Paulo]` , enquanto `dataTokyo` corresponde a `2018-05-05T05:00+09:00[Asia/Tokyo]` . Como alguns valores numéricos da data e

hora são diferentes (além do offset e timezone também não serem os mesmos), `equals()` retorna `false` .

Mas as duas instâncias correspondem ao mesmo instante. Se convertermos ambas para UTC, o resultado será `2018-05-04T20:00Z` . Por isso `isEqual()` retorna `true` .

Um detalhe importante é que `equals()` também compara o valor do offset. Isso faz diferença nos casos em que há *overlap* e um horário pode existir duas vezes em um timezone.

No timezone `America/Sao_Paulo` , por exemplo, os valores `2018-02-17T23:00-02:00` e `2018-02-17T23:00-03:00` são válidos, mas como o offset é diferente, `equals()` retorna `false` . E como eles representam instantes diferentes, `isEqual()` também retorna `false` .

Comparando em ordem cronológica

`ZonedDateTime` possui os métodos `isBefore()` e `isAfter()` , que verificam se o instante correspondente (o ponto na linha do tempo) ocorre antes ou depois de outro:

```
// 2018-05-04T17:00-03:00[America/Sao_Paulo]
ZoneId spZone = ZoneId.of("America/Sao_Paulo");
ZonedDateTime dataSP = ZonedDateTime.of(2018, 5, 4, 17, 0, 0, 0, spZone);
// 2018-05-04T23:00+09:00[Asia/Tokyo]
ZoneId tokyoZone = ZoneId.of("Asia/Tokyo");
ZonedDateTime dataTokyo = ZonedDateTime.of(2018, 5, 4, 23, 0, 0, 0,
tokyoZone);
System.out.println(dataSP.isBefore(dataTokyo)); // false
System.out.println(dataSP.isAfter(dataTokyo)); // true
```

A variável `dataSP` corresponde a `2018-05-04T17:00-03:00[America/Sao_Paulo]` (4 de maio de 2018, às 17:00), e `dataTokyo` corresponde a `2018-05-04T23:00+09:00[Asia/Tokyo]` (4 de maio, às 23:00).

Os métodos `isBefore()` e `isAfter()` levam em consideração o instante que as datas representam (o ponto na linha do tempo). Em UTC,

`dataSP` equivale a `2018-05-04T20:00Z` , e `dataTokyo` equivale a `2018-05-04T14:00Z` . Portanto, `dataSP` corresponde a um instante ocorrido depois de `dataTokyo` . Por isso `isBefore()` retorna `false` e `isAfter()` retorna `true` .

Além disso, `ZonedDateTime` implementa a interface `Comparable` , e o método `compareTo()` também leva em conta o instante que a classe representa. Por isso um `List` contendo instâncias de `ZonedDateTime` será ordenado considerando-se a ordem cronológica.

14.5 Converter para outro timezone

`Date` e os tipos locais não possuem um timezone, por isso não podem ser convertidos para outro. E `Calendar` pode ser convertido através do método `setTimeZone()` , que ajusta os seus campos para corresponder ao novo timezone.

`ZonedDateTime` também pode ser facilmente convertido para outro timezone, porém há duas formas diferentes de fazê-lo.

Mesmo instante, outro timezone

Quando é 4 de maio de 2018, às 17:00 em São Paulo, qual será a data e a hora em Tóquio?

Ou seja, eu quero saber qual é a data e hora de Tóquio, **no mesmo instante** em que é 4 de maio de 2018, às 17:00 em São Paulo. Para isso existe o método `withZoneSameInstant()` . Para usá-lo, basta passar o `ZoneId` para o qual você quer converter:

```
// 2018-05-04T17:00-03:00[America/Sao_Paulo]
ZoneId spZone = ZoneId.of("America/Sao_Paulo");
ZonedDateTime dataSP = ZonedDateTime.of(2018, 5, 4, 17, 0, 0, 0, spZone);
// converter para Asia/Tokyo
ZoneId tokyoZone = ZoneId.of("Asia/Tokyo");
ZonedDateTime dataTokyo = dataSP.withZoneSameInstant(tokyoZone);
```

O método `withZoneSameInstant()` converte a data, hora e offset para outro timezone, sendo que o `ZonedDateTime` retornado corresponde ao mesmo instante do original. Após a execução deste código, `dataTokyo` terá os valores da data, hora e offset ajustados de acordo com o timezone passado (`Asia/Tokyo`), mas corresponderá ao mesmo instante de `dataSP` — ou seja, `dataSP.isEqual(dataTokyo)` será `true` . O valor de `dataTokyo` , no caso, é `2018-05-05T05:00+09:00[Asia/Tokyo]` .

Mesma data e hora local, outro timezone

`ZonedDateTime` possui uma outra maneira de conversão entre timezones, mas com um propósito diferente do caso anterior.

Suponha que eu tenha uma instância equivalente a 4 de maio de 2018, às 17:00 em São Paulo. Se eu quiser outra instância que corresponda aos **mesmos valores numéricos de data e hora**, porém em outro timezone, eu uso o método `withZoneSameLocal()` :

```
// 2018-05-04T17:00-03:00[America/Sao_Paulo]
ZoneId spZone = ZoneId.of("America/Sao_Paulo");
ZonedDateTime dataSP = ZonedDateTime.of(2018, 5, 4, 17, 0, 0, 0, spZone);
ZoneId tokyoZone = ZoneId.of("Asia/Tokyo");
// manter o mesmo dia e hora, e só mudar o timezone
ZonedDateTime dataTokyo = dataSP.withZoneSameLocal(tokyoZone);
```

A ideia do nome do método (`SameLocal`) é que a data e hora **local** são mantidas (portanto, continuará sendo 4 de maio de 2018, às 17:00) e apenas o timezone — e consequentemente, o offset — são alterados. Por isso o valor de `dataTokyo` é `2018-05-04T17:00+09:00[Asia/Tokyo]` .

O resultado foi a mesma data e hora do `ZonedDateTime` original (`2018-05-04T17:00`). Porém, como o timezone mudou para `Asia/Tokyo` , o offset foi ajustado para `+09:00` . Com isso, `dataSP` e `dataTokyo` representam instantes completamente diferentes, já que 17:00 do dia 4 ocorre em um momento diferente em cada um dos timezones envolvidos.

14.6 Combinando as classes locais com ZoneId

As classes locais não possuem um timezone, mas existem várias maneiras de combiná-las com um `ZoneId` para obter um `ZonedDateTime` . Vamos ver algumas delas.

Juntar LocalDateTime com ZoneId

`LocalDateTime` representa uma data e hora sem timezone. Para obtermos um `ZonedDateTime` que representa essa mesma data e hora em algum timezone, basta juntar o `LocalDateTime` com um `ZoneId` . Há duas maneiras de fazê-lo, demonstradas no próximo exemplo:

```
// 2018-05-04T17:00
LocalDateTime dataHora = LocalDateTime.of(2018, 5, 4, 17, 0);
ZoneId zone = ZoneId.of("America/Sao_Paulo");
// as duas formas abaixo são equivalentes
ZonedDateTime dataHoraSP = dataHora.atZone(zone);
ZonedDateTime dataHoraSP = ZonedDateTime.of(dataHora, zone);
```

O `LocalDateTime` representa 4 de maio de 2018, às 17:00, sem nenhuma informação sobre timezones ou offsets. Ele só possui os valores numéricos da data e hora, nada mais. Ao juntá-lo com um `ZoneId` , as informações do timezone são consultadas para saber qual o offset usado naquela data e hora. Com todas estas informações, é criado o `ZonedDateTime` , que no caso será equivalente a `2018-05-04T17:00-03:00[America/Sao_Paulo]` .

Quando o `LocalDateTime` corresponder a uma data e hora que está dentro de um *gap*, serão feitos os devidos ajustes, conforme já vimos no começo do capítulo, na seção "Lidando com os *gaps*". E quando o `LocalDateTime` corresponder a um *overlap*, você pode usar os métodos `withEarlierOffsetAtOverlap()` e `withLaterOffsetAtOverlap()` para ajustar o `ZonedDateTime` para a primeira ou segunda ocorrência do horário local.

Juntar LocalDate com ZoneId

`LocalDate` possui apenas dia, mês e ano, e somente com estas informações, um `ZoneId` não consegue decidir com certeza absoluta qual o offset usado naquele momento. Quando há alguma transição de horário de verão, o mesmo dia pode ter dois offsets, dependendo do horário.

Por isso, para juntar um `LocalDate` com um `ZoneId` , eu preciso definir um horário. Isso pode ser feito com o método `atTime()` , que recebe valores do horário e cria um `LocalDateTime` , e em seguida com o método `atZone()` . Exemplo:

```
// 2018-05-04
LocalDate data = LocalDate.of(2018, 5, 4);
ZoneId zone = ZoneId.of("America/Sao_Paulo");
// setar horário para 10:00 e juntar com o ZoneId
ZonedDateTime dataHoraSP = data.atTime(10, 0).atZone(zone);
```

Setei o horário para 10:00, criando um `LocalDateTime` . Em seguida, chamei `atZone()` , que verifica junto ao `ZoneId` qual o offset usado pelo timezone `America/Sao_Paulo` naquele dia e horário, e o resultado é `2018-05-04T10:00-03:00[America/Sao_Paulo]` .

Outra maneira, equivalente a esta, é usar uma variação do método `of()` , que recebe um `LocalDate` , um `LocalTime` e um `ZoneId` :

```
// 2018-05-04
LocalDate data = LocalDate.of(2018, 5, 4);
// 10:00
LocalTime hora = LocalTime.of(10, 0);
ZoneId zone = ZoneId.of("America/Sao_Paulo");
// juntar data, hora e timezone
ZonedDateTime dataHoraSP = ZonedDateTime.of(data, hora, zone);
```

O resultado é o mesmo do código anterior: `2018-05-04T10:00-03:00[America/Sao_Paulo]` .

Obter o início do dia

`LocalDate` possui o método `atStartOfDay()` , que funciona de maneira similar ao `atTime()` , mas usando sempre meia-noite como o horário. Só há um pequeno detalhe:

- Se eu chamar este método sem parâmetros, o resultado é um `LocalDateTime` com o horário setado para meia-noite.
- Se eu passar um `ZoneId` , o resultado é um `ZonedDateTime` com o horário setado para meia-noite, mas com a possibilidade de ser ajustado caso caia em um *gap* ou *overlap*.

Exemplo:

```
// 2018-05-04
LocalDate data = LocalDate.of(2018, 5, 4);
ZoneId zone = ZoneId.of("America/Sao_Paulo");
// sem ZoneId, retorna LocalDateTime: 2018-05-04T00:00
LocalDateTime inicioDoDia = data.atStartOfDay();
// com ZoneId, retorna ZonedDateTime: 2018-05-04T00:00-
03:00[America/Sao_Paulo]
ZonedDateTime inicioDoDiaSP = data.atStartOfDay(zone);
```

Na maioria das vezes, o horário do `ZonedDateTime` retornado será meia-noite, como é o caso do exemplo anterior, cujo resultado é `2018-05-04T00:00-03:00[America/Sao_Paulo]` .

Mas se meia-noite fizer parte de um *gap*, o início do dia pode ser um horário diferente, como é o caso do timezone `America/Sao_Paulo` : quando o horário de verão começa, à meia-noite o relógio é adiantado em uma hora, portanto o início deste dia é 01:00, conforme mostra o próximo exemplo:

```
// 2017-10-15 - dia que começa o horário de verão em São Paulo
LocalDate data = LocalDate.of(2017, 10, 15);
ZoneId zone = ZoneId.of("America/Sao_Paulo");
// 2017-10-15T01:00-02:00[America/Sao_Paulo]
ZonedDateTime inicioDoDia = data.atStartOfDay(zone);
```

Neste caso, no dia 15 de outubro de 2017, no timezone `America/Sao_Paulo` , o horário de verão começa e à meia-noite os relógios são adiantados para 01:00. Por isso o início do dia não é

meia-noite, e sim 1 da manhã. Assim, o resultado é `2017-10-15T01:00-02:00[America/Sao_Paulo]`.

14.7 Trabalhando somente com offsets

Existe uma classe que possui apenas a data, hora e offset, sem nenhuma referência ao timezone: `java.time.OffsetDateTime`. Ela pode ser extraída de um `ZonedDateTime` por meio do método `toOffsetDateTime()`.

```
// 2018-05-04T17:00-03:00[America/Sao_Paulo]
ZonedDateTime zdt = ZonedDateTime.now();
// 2018-05-04T17:00-03:00
OffsetDateTime odt = zdt.toOffsetDateTime();
```

O seu funcionamento básico segue os mesmos princípios das demais classes: métodos `now()` para obter a data, hora e offset atual, *getters* para obter campos específicos, métodos `withXXX` para mudar valores (na verdade, retornar outra instância com o valor modificado) etc.

A diferença é que `OffsetDateTime` não possui um timezone (somente o offset), e por isso não é afetada pelos *gaps* e *overlaps*. Seu offset é sempre o mesmo, ao contrário do que acontece com `ZonedDateTime`, que ajusta o offset conforme o histórico do seu timezone.

Converter OffsetDateTime para ZonedDateTime

Converter um `OffsetDateTime` de volta para o `ZoneDateTime` original (ou seja, mantendo o mesmo offset) não é uma tarefa tão trivial.

Como já vimos várias vezes ao longo do livro, vários timezones diferentes podem usar o mesmo offset em determinado instante. Para saber quais são, temos que percorrer a lista de todos os timezones disponíveis (retornada por `ZoneId.getAvailableZoneIds()`) e para cada um verificar o offset utilizado neste mesmo instante.

Para isso, usarei o método `atZoneSameInstant()` para converter o `OffsetDateTime` para um timezone. Este método retorna um `ZonedDateTime` que corresponde ao mesmo instante do `OffsetDateTime` , ajustando a data, hora e o offset automaticamente. Com isso, podemos verificar se o offset usado por outro timezone é o mesmo do nosso `OffsetDateTime` .

```
// 2018-05-04T17:00-03:00
OffsetDateTime offsetDt = OffsetDateTime.now();
// verificar todos os timezones disponíveis
ZoneId.getAvailableZoneIds().forEach(zoneName -> {
    // obter o offset usado neste timezone, neste mesmo instante
    ZoneId zone = ZoneId.of(zoneName);
    ZoneOffset offset = offsetDt.atZoneSameInstant(zone).getOffset();
    // verifica se os offsets são iguais
    if (offset.equals(offsetDt.getOffset())) {
        System.out.println(zoneName);
    }
});
```

Eu rodei este código em uma JVM com a versão `2018e` do TZDB e com `now()` retornando `2018-05-04T17:00-03:00` , então pode ser que você não obtenha exatamente o mesmo resultado. Foram encontrados mais de 40 timezones, entre eles `America/Recife` (Brasil), `America/Cordoba` (Argentina), `Atlantic/Bermuda` (Ilhas Bermudas) e `Antarctica/Palmer` (uma estação científica dos EUA na Antártida). Todos esses lugares usavam o offset `-03:00` no mesmo instante correspondente à variável `offsetDt` .

Ou seja, para converter o `OffsetDateTime` para um `ZonedDateTime` **que use o mesmo offset**, devemos escolher um dos timezones desta lista. Outro detalhe é que, como um timezone pode usar diferentes offsets ao longo da história, esta lista pode mudar dependendo do valor do `OffsetDateTime` utilizado.

Mas isso só vale para os casos em que você quer converter para um timezone que use exatamente o mesmo offset. Para converter para um timezone qualquer, basta usar o método `atZoneSameInstant()` ,

lembrando que ele pode ajustar a data, hora e offset caso necessário.

Tem mais algum tipo?

Até agora vimos os tipos locais (`LocalDate` , `LocalTime` e `LocalDateTime`), que possuem apenas a data e/ou hora, sem nenhuma informação sobre o timezone ou offset. Também vimos `OffsetDateTime` , que possui a data, hora e offset, e `ZonedDateTime` , que também possui um timezone.

Além disso, conhecemos a classe `ZoneId` , que representa um timezone (um identificador da IANA com todo o histórico de offsets de uma determinada região), e `ZoneOffset` , que representa um único offset (a diferença com relação a UTC, sem informações históricas nem qualquer relação com uma região específica).

No próximo capítulo veremos como o `java.time` representa um timestamp.

Capítulo 15
Instant e TemporalFields

Os tipos que vimos até agora trabalham com uma data e/ou hora específicas, em determinado offset ou timezone. Mas sabemos que também é possível referir-se a um ponto específico na linha do tempo, independente do timezone, através de um Unix timestamp. Na API `java.time` este conceito é implementado pela classe `java.time.Instant` .

15.1 Instant representa um timestamp

A classe `Instant` representa um ponto na linha do tempo (um timestamp, o "número gigante" que vimos na primeira parte do livro). Mais precisamente, um `Instant` possui a quantidade de nanossegundos desde o Unix Epoch.

Diferente das outras classes já vistas, um `Instant` não possui campos de data e hora, nem timezone e offset. Ele possui somente o valor numérico do timestamp. Como este valor pode corresponder a uma data, hora e offset diferentes em cada timezone, estes campos não fazem parte do `Instant` .

Para saber o timestamp atual, basta usar o método `now()` :

```
Instant agora = Instant.now();
System.out.println(agora);
```

Considerando que nos exemplos do livro a data atual corresponde a 4 de maio de 2018, às 17:00 em São Paulo (`2018-05-04T17:00-03:00`), o `Instant` retornado por `now()` possui o valor de timestamp correspondente (`1525464000000`).

Porém, o método `toString()` não retorna o valor numérico do timestamp, e sim o seu valor equivalente em UTC, no formato ISO 8601. Por isso, a saída do código anterior é:

```
2018-05-04T20:00:00Z
```

Este é um ponto que pode causar confusão, pois esta saída pode dar a entender que o `Instant` possui os valores numéricos da data e hora, quando na verdade ele só possui o valor do timestamp. Será que a API cometeu o mesmo erro de `Date` ?

A decisão de fazer `toString()` retornar a data e hora em UTC em vez do timestamp é comentada por Stephen Colebourne nesta *issue* do GitHub (https://github.com/ThreeTen/threeten/issues/271#issuecomment-14204085/):

If we were really hard line, the `toString` of an `Instant` would simply be the number of seconds from 1970-01-01Z. We chose not to do that, and output a more friendly toString to aid developers.

Em tradução livre: *Se quiséssemos ser realmente "linha-dura", o método `toString()` de um `Instant` retornaria simplesmente o número de segundos deste 1970-01-01Z. Escolhemos não fazer isso, e retornar uma saída mais amigável para ajudar os desenvolvedores*.

Ou seja, foi uma decisão de implementação que visa deixar o retorno do método mais amigável para os desenvolvedores. Como as APIs de log e debuggers também costumam usar `toString()` para mostrar o valor dos objetos, foi uma escolha razoável, na minha opinião.

A diferença para a API legada é que o valor retornado é sempre o mesmo, independente de qual for o timezone padrão da JVM. Não é como `Date` , cuja saída do método `toString()` mudava de acordo com a configuração do timezone padrão, omitia os milissegundos e no fim confundia mais do que ajudava.

E como saber o valor do timestamp?

Um modo de obter o timestamp é usar o método `toEpochMilli()` , que retorna um `long` contendo o valor em milissegundos:

```
long timestamp = agora.toEpochMilli();
```

O valor do timestamp é `1525464000000` . Se você precisar **somente** do valor numérico do timestamp em milissegundos em uma variável `long` e nada mais, não precisa criar um `Instant` só para isso. Neste caso, basta usar `System.currentTimeMillis()` .

O valor retornado pelo exemplo anterior está em milissegundos. Mas as classes do `java.time` possuem precisão de nanossegundos, então como obter os valores com a precisão máxima?

`Instant` guarda o valor do timestamp em dois campos: um contendo a quantidade de **segundos** desde o Unix Epoch, e outro contendo os nanossegundos. Cada um destes campos possui seu próprio *getter*:

```
long timestampSegundos = agora.getEpochSecond();
int nanossegundos = agora.getNano();
```

Neste exemplo, a variável `timestampSegundos` conterá o timestamp em segundos (no caso, `1525464000`), enquanto `nanossegundos` contém o valor dos nanossegundos. Neste caso será zero, pois o horário atual que estamos usando representa uma "hora cheia", com o valor dos minutos, segundos e frações de segundo igual a zero.

Quando chamamos o método `toEpochMilli()` , estes dois valores são combinados para gerar o timestamp em milissegundos. Ou seja, os nanossegundos são truncados e as 6 últimas casas decimais são perdidas.

Diferente das demais classes, `Instant` não possui um método `now()` que recebe um `ZoneId` como parâmetro. Nas outras classes isso é possível porque o timezone é usado para obter os valores de data, hora e offset. Mas `Instant` representa um timestamp, um valor que é

sempre o mesmo, independente do timezone. Por isso, não faz sentido passar um `ZoneId` para o método `Instant.now()` , pois não importa qual o timezone utilizado, o timestamp será o mesmo.

15.2 Criar um Instant a partir do timestamp

Além de `now()` , que usa o timestamp atual, é possível criar um `Instant` a partir de um valor específico, que pode estar tanto em segundos quanto em milissegundos. No próximo exemplo, ambos os `Instants` gerados são equivalentes:

```
// usar timestamp em milissegundos
Instant instant = Instant.ofEpochMilli(1525464000000L);
// usar timestamp em segundos
Instant instant2 = Instant.ofEpochSecond(1525464000L);
```

Ambos os métodos do exemplo anterior retornam um `Instant` correspondente a `2018-05-04T20:00:00Z` .

A existência dos dois métodos (`ofEpochMilli()` e `ofEpochSecond()`) é bem útil, pois algumas APIs trabalham com o timestamp em segundos, enquanto outras só trabalham com este valor em milissegundos. É claro que o `java.time` poderia disponibilizar apenas um dos métodos, e ao desenvolvedor bastaria multiplicar ou dividir por 1000, mas a API possui as duas opções.

Além disso, `ofEpochSecond()` também pode receber o valor dos nanossegundos. Isso é útil quando precisamos trabalhar com precisões maiores que milissegundos:

```
// usar timestamp em segundos, mais os nanossegundos
Instant instant = Instant.ofEpochSecond(1525464000L, 123456789);
```

Neste caso, a fração de segundo terá o valor `123456789` (que também é o valor retornado por `instant.getNano()`) e o resultado será `2018-05-04T20:00:00.123456789Z` .

Se eu chamar o método `toEpochMilli()` neste `Instant` , os nanossegundos serão truncados para milissegundos e os últimos dígitos (`456789`) serão perdidos: o retorno será `1525464000123` .

Obter Instant a partir de outras classes

As classes `ZonedDateTime` e `OffsetDateTime` possuem todas as informações necessárias para se obter um timestamp (data, hora e offset). Por isso ambas possuem o método `toInstant()` , que retorna o `Instant` correspondente.

Já as classes locais não possuem informações suficientes para construir um `Instant` . No código a seguir temos um `LocalDate` , que só possui a data. Para termos um timestamp, precisamos também da hora e do offset. Por isso, temos que construir um `ZonedDateTime` ou um `OffsetDateTime` , para só então obtermos o `Instant` :

```
// 4 de maio de 2018
LocalDate data = LocalDate.of(2018, 5, 4);
// 2018-05-04T00:00-03:00[America/Sao_Paulo]
ZonedDateTime zdt = data.atStartOfDay(ZoneId.of("America/Sao_Paulo"));
System.out.println(zdt.toInstant()); // 2018-05-04T03:00:00Z
// 2018-05-04T00:00+05:00
OffsetDateTime odt = data.atTime(0, 0).atOffset(ZoneOffset.of("+05:00"));
System.out.println(odt.toInstant()); // 2018-05-03T19:00:00Z
```

Algumas APIs podem até retornar um timestamp usando uma data sem o horário, ou uma data e hora sem o offset. Mas por baixo dos panos elas estão usando algum valor predefinido (exatamente como `SimpleDateFormat` faz, setando o horário para meia-noite e usando o timezone padrão da JVM, quando estas informações não são passadas).

Na API `java.time` decidiram não fazer estas "mágicas", e se você tiver um `LocalDate` , `LocalTime` ou `LocalDateTime` (ou qualquer outra classe que não tenha todas as informações necessárias para se ter um timestamp) e quiser obter um `Instant` a partir deles, terá que prover as informações faltantes, como foi feito no exemplo anterior.

Comparação e ordenação

Comparar duas instâncias de `Instant` é bem simples. Como esta classe representa um ponto na linha do tempo e internamente só possui o valor numérico do timestamp, todas as comparações são feitas com base neste valor.

Os métodos `isBefore()` e `isAfter()` verificam se um `Instant` ocorre antes ou depois de outro, e `equals()` , que verifica se ambos são iguais. Não há um método `isEqual()` , já que a única maneira de comparar instâncias de `Instant` é usando o valor numérico do timestamp, e não há a necessidade de dois métodos, como ocorre com `ZonedDateTime` .

E assim como as demais classes do `java.time` , `Instant` também implementa `Comparable` , por isso é possível ordenar uma lista de `Instant` usando `Collections.sort()` . A ordenação é feita de acordo com o valor do timestamp.

15.3 TemporalField e os getters genéricos

Além dos métodos `getEpochSecond()` e `getNano()` , só há mais dois *getters* na classe `Instant` : `get()` e `getLong()` . Estes métodos são herdados da interface `java.time.temporal.TemporalAccessor` , que é implementada por todos os tipos de data e hora do `java.time` (portanto, todas as classes que vimos até agora também possuem estes dois *getters*).

A ideia destes métodos é serem *getters* genéricos, que podem retornar o valor de qualquer campo que puder ser obtido do objeto em questão. Eles recebem como parâmetro um `java.time.temporal.TemporalField` , que é uma interface que representa um campo de data ou hora (similar às constantes de `Calendar` , como `DAY_OF_MONTH` e `HOUR_OF_DAY`).

Há várias implementações de `TemporalField` na própria API. A que contém os campos mais comuns do dia a dia é `java.time.temporal.ChronoField`, que veremos com mais detalhes a partir de agora.

Usando um ChronoField

`ChronoField` representa um campo de data ou hora, similar às constantes de `Calendar`, como `Calendar.DAY_OF_MONTH` e `Calendar.HOUR_OF_DAY`. A diferença é que as constantes da API legada são números (do tipo `int`), enquanto um `ChronoField` é um `enum`. Por isso não há risco de usar um campo no lugar dos valores, como vimos que pode acontecer com `Calendar`.

Um `ChronoField` pode ser passado para os métodos `get()` e `getLong()` para obter o valor numérico de um campo, como mostra o código a seguir:

```
// 2018-05-04T17:00
LocalDateTime data = LocalDateTime.of(2018, 5, 4, 17, 0);
int mes = data.get(ChronoField.MONTH_OF_YEAR); // 5
int minuto = data.get(ChronoField.MINUTE_OF_HOUR); // 0
```

Neste exemplo, eu usei `MONTH_OF_YEAR`, que corresponde ao mês (por isso, o valor retornado é `5`) e `MINUTE_OF_HOUR`, que corresponde aos minutos (que, no caso, é `0`). Porém, a classe `LocalDateTime` já tem os métodos `getMonthValue()` e `getMinute()`, que retornam exatamente estes valores. Qual a diferença então?

Neste caso específico, de fato não faz diferença. Mas `ChronoField` também serve para obtermos valores que não possuem um *getter* correspondente. Um exemplo é o campo `ChronoField.MINUTE_OF_DAY`, que representa o "minuto do dia", ou seja, a quantidade de minutos decorrida desde o início do dia (meia-noite). No exemplo a seguir é mostrado também o uso de `import static`, que deixa o código mais legível:

```
import static java.time.temporal.ChronoField.MINUTE_OF_DAY;

// 2018-05-04T17:00
LocalDateTime dataHora = LocalDateTime.of(2018, 5, 4, 17, 0);
int minutoDoDia = dataHora.get(MINUTE_OF_DAY); // 1020
```

O valor retornado foi `1020` , pois às 17:00 já se passaram 1020 minutos, se contarmos a partir da meia-noite.

Outro exemplo: `Instant` possui o método `getNano()` , que retorna o valor dos nanossegundos. Mas e se eu quiser este valor em milissegundos? Eu poderia usar `getNano()` e dividir o resultado por 1 milhão (ou usar `TimeUnit.convert()`), mas também posso usar o `ChronoField` correspondente, que já faz essa conta para mim:

```
Instant instant = Instant.ofEpochSecond(1525464000L, 123456789);
// ChronoField.MILLI_OF_SECOND retorna a fração de segundos em
milissegundos
int milissegundos = instant.get(ChronoField.MILLI_OF_SECOND); // 123
```

Por que existe get e getLong?

O método `get(TemporalField)` retorna um `int` , mas alguns campos possuem valores que extrapolam os limites de um inteiro. Por exemplo, o campo `NANO_OF_DAY` representa os "nanossegundos do dia", ou seja, a quantidade de nanossegundos decorridos desde a meia-noite. Se tentarmos usá-lo no método `get()` :

```
int nanossegundosDoDia = dataHora.get(ChronoField.NANO_OF_DAY);
```

O resultado será uma exceção:

```
java.time.temporal.UnsupportedTemporalTypeException: Invalid field
'NanoOfDay' for get() method, use getLong() instead
```

A mensagem da exceção diz que o campo `NanoOfDay` é inválido para o método `get()` e recomenda o uso de `getLong()` . Isso acontece porque os valores válidos para `NANO_OF_DAY` ultrapassam o limite de um `int` .

Podemos verificar quais os valores que um campo pode ter através do método `range()` , que retorna um `java.time.temporal.ValueRange` (uma classe que representa um intervalo de valores):

```
// obter os limites para os valores de NANO_OF_DAY
ValueRange limites = dataHora.range(ChronoField.NANO_OF_DAY);
System.out.println(limites);
```

Ao imprimir o `ValueRange` , são informados os valores mínimo e máximo:

```
0 - 86399999999999
```

Como podemos ver, o valor máximo do campo `NANO_OF_DAY` é `86399999999999` (mais de 86 trilhões), mas o maior valor para um `int` é `Integer.MAX_VALUE` , que é `2147483647` . Por isso, `get(NANO_OF_DAY)` lança uma exceção, já que o campo pode retornar valores maiores do que um `int` suporta.

Como saber se o valor ultrapassa o limite de int

Para saber se um `ValueRange` possui valores que "cabem" em um `int` , podemos usar o método `isIntValue()` . No caso do exemplo anterior, `limites.isIntValue()` retorna `false` , pois os valores contidos no `ValueRange` ultrapassam os limites de um `int` .

Por isso, para obter o valor deste campo, temos que usar o método `getLong()` :

```
// 61200000000000
long nanossegundosDoDia = dataHora.getLong(ChronoField.NANO_OF_DAY);
```

Caso você saiba que os valores de um campo "cabem" em um `int` (por exemplo, o mês e o dia, que possuem valores máximos bem pequenos), pode usar `get()` diretamente. Mas se você não souber, usar `getLong()` é mais garantido, pois nenhum campo retorna valores maiores que `Long.MAX_VALUE` .

Todos os tipos de data e hora possuem o método `range()` . Um detalhe interessante é que o `ValueRange` retornado tem os limites ajustados de acordo com os valores do objeto em questão. Por exemplo, para o campo `DAY_OF_MONTH` (dia do mês), o valor máximo pode variar de acordo com o mês, como mostra o próximo exemplo:

```
// verificar dia do mês para datas em fevereiro
System.out.println(LocalDate.of(2018, 2,
1).range(ChronoField.DAY_OF_MONTH));
System.out.println(LocalDate.of(2020, 2,
1).range(ChronoField.DAY_OF_MONTH));
```

Ambas as datas do exemplo estão em fevereiro. No primeiro caso, o ano não é bissexto (2018), portanto o maior valor para o dia é 28. Já em 2020 (que é bissexto), o maior valor é 29. Então a saída é:

```
1 - 28
1 - 29
```

Instant não possui campos de data e hora

Vamos ver o que acontece quando tentamos obter um campo de `Instant` usando `get()` :

```
// 2018-05-04T20:00:00Z
Instant instant = Instant.ofEpochSecond(1525464000L);
// tentar obter a hora
System.out.println(instant.get(ChronoField.HOUR_OF_DAY));
```

Neste código, eu tento obter o valor do campo `HOUR_OF_DAY` , mas o método `get()` lança uma exceção, dizendo que este campo não é suportado:

```
java.time.temporal.UnsupportedTemporalTypeException: Unsupported field:
HourOfDay
```

Isso acontece porque a classe `Instant` **não** representa uma data e hora específicas, e sim um timestamp: um valor que corresponde a uma data e hora diferentes em cada timezone. Por isso não é possível obter um valor para as horas, e nem para qualquer outro

campo de data ou hora, como o dia, mês, minutos etc. Para obter estes valores, temos que converter o `Instant` para um timezone ou offset, usando os métodos `atZone(ZoneId)` e `atOffset(ZoneOffset)` :

```
// 2018-05-04T20:00:00Z
Instant instant = Instant.ofEpochSecond(1525464000L);
// converter para um timezone
ZonedDateTime sp = instant.atZone(ZoneId.of("America/Sao_Paulo"));
// converter para um offset
OffsetDateTime odt = instant.atOffset(ZoneOffset.of("+05:00"));
```

Os métodos `atZone()` e `atOffset()` retornam respectivamente um `ZonedDateTime` (cujo valor é `2018-05-04T17:00-03:00[America/Sao_Paulo]`) e um `OffsetDateTime` (cujo valor é `2018-05-05T01:00+05:00`), ambos com os valores de data, hora e offset correspondentes ao valor do timestamp que o `Instant` representa.

A partir destes objetos, é possível obter os campos desejados: dia, mês, hora, minuto etc. Para obter os valores em UTC, basta passar a constante `ZoneOffset.UTC` como parâmetro para `atZone()` ou `atOffset()` .

Verificar se um campo pode ser obtido

Para evitar que ocorra um `UnsupportedTemporalTypeException` ao usar `get()` ou `getLong()` , podemos usar o método `isSupported()` , que verifica se o campo é suportado ou não:

```
if (instant.isSupported(ChronoField.HOUR_OF_DAY)) {
    System.out.println(instant.get(ChronoField.HOUR_OF_DAY));
}
```

Neste caso, o código não imprime nada, pois o método `isSupported()` retorna `false` , já que o campo `HOUR_OF_DAY` não é suportado pela classe `Instant` . Vale lembrar que, mesmo que o campo seja suportado, o método `get()` ainda pode lançar uma exceção, caso o valor exceda os limites de um `int` .

Todas as classes de data e hora possuem o método `isSupported()`, que podemos usar para verificar se é possível chamar `get()` ou `getLong()` com determinado campo. Por exemplo, `ChronoField.HOUR_OF_DAY` não é suportado por `LocalDate`, pois esta classe só possui os campos de data (dia, mês e ano), e `ChronoField.DAY_OF_WEEK` não é suportado por `LocalTime` pois esta classe não possui os campos de data, e por isso não é possível obter o dia da semana.

15.4 Mudando campos do Instant

Na classe `Instant` só há dois métodos `with()`: um que recebe um `TemporalAdjuster` (igual ao que já vimos anteriormente) e outro que recebe um `TemporalField` e seu respectivo valor. Este último permite que mudemos o valor de qualquer campo:

```
// 2018-05-04T20:00:00Z
Instant instant = Instant.ofEpochSecond(1525464000L);
// mudar os milissegundos
instant = instant.with(ChronoField.MILLI_OF_SECOND, 123);
```

No exemplo anterior, primeiro eu crio um `Instant` cujo timestamp equivale a `2018-05-04T20:00:00Z`. Depois eu mudo o valor do campo `MILLI_OF_SECOND` (milissegundos) para `123` e seto o retorno na mesma variável `instant`. Com isso, o resultado é um `Instant` equivalente a `2018-05-04T20:00:00.123Z`.

Vale lembrar que internamente o valor das frações de segundo (no caso, 123 milissegundos) é convertido em nanossegundos (`123000000`, que será o valor retornado por `instant.getNano()`).

Isso vale para todos

Todas as classes de data e hora que já vimos também podem ter qualquer campo mudado com o método `with(TemporalField, long)`,

desde que o campo seja suportado, claro. Por exemplo, podemos mudar o "segundo do dia" de um `LocalTime` :

```
// 17:00
LocalTime hora = LocalTime.of(17, 0);
// mudar para o centésimo segundo do dia
hora = hora.with(ChronoField.SECOND_OF_DAY, 100);
```

O resultado é `00:01:40` (meia-noite e um, e 40 segundos), pois este é o horário correspondente ao centésimo segundo do dia, se começarmos a contar a partir da meia-noite.

Este método é útil para mudarmos campos para os quais não há um método `withXXX` específico. Sempre consulte a documentação para saber quais os ajustes feitos para cada campo.

15.5 Usando TemporalField com TemporalQuery

Todos os *getters* retornam um valor numérico ou uma classe específica da API (como `getMonth()` , que retorna um `java.time.Month`). Mas o `java.time` também provê um mecanismo para que o retorno seja uma instância de qualquer classe que quisermos. Este mecanismo é representado pela interface `java.time.temporal.TemporalQuery` .

Para criar um `TemporalQuery` , basta implementar o método `queryFrom()` , que recebe como parâmetro um `TemporalAccessor` (a interface que só possui os *getters* genéricos `get()` e `getLong()`).

Criando um TemporalQuery

Vamos criar um `TemporalQuery` para saber se uma data é fim de semana. Para isso, temos que obter o valor do campo `DAY_OF_WEEK` e retornar um *booleano* indicando se é sábado ou domingo.

Como o retorno é *booleano*, declaramos a variável como `TemporalQuery<Boolean>` — o tipo parametrizado informa qual é o retorno. Por fim, chamamos o método `query()` para usar a `TemporalQuery` . Este método também pertence a `TemporalAccessor` , e por isso está disponível em todas as classes de data e hora.

E como `TemporalQuery` é uma interface funcional, podemos usar a sintaxe de *lambda* do Java 8 para criá-lo:

```
// criar TemporalQuery ("temporal" é um TemporalAccessor)
TemporalQuery<Boolean> fimDeSemana = temporal -> {
    // valor numérico do dia da semana
    int diaDaSemana = temporal.get(ChronoField.DAY_OF_WEEK);
    // comparar com o valor numérico de sábado e domingo
    return diaDaSemana == DayOfWeek.SATURDAY.getValue()
           || diaDaSemana == DayOfWeek.SUNDAY.getValue();
};
LocalDate data = LocalDate.of(2018, 5, 4);
System.out.println(data.query(fimDeSemana)); // false
```

Como o valor do `LocalDate` é 4 de maio de 2018 (uma sexta-feira), o retorno de `query(fimDeSemana)` é `false` .

`TemporalQuery` funciona com qualquer classe que implemente `TemporalAccessor` . Ou seja, eu posso usá-lo com qualquer tipo de data e hora da API, desde que os campos sendo usados sejam suportados, é claro.

```
// 6 de maio de 2018 (domingo)
LocalDateTime dataHora = LocalDateTime.of(2018, 5, 6, 17, 0);
System.out.println(dataHora.query(fimDeSemana)); // true
// converte para um timezone, data continua a mesma
ZonedDateTime dataHoraSP =
dataHora.atZone(ZoneId.of("America/Sao_Paulo"));
System.out.println(dataHoraSP.query(fimDeSemana)); // true
// tentar usar com um LocalTime
LocalTime hora = LocalTime.now();
System.out.println(hora.query(fimDeSemana)); // exceção
```

Neste exemplo, o `LocalDateTime` e `ZonedDateTime` possuem a mesma data (6 de maio de 2018, um domingo) e por isso o `TemporalQuery` retorna `true` para ambos. Mas quando tento usar o `TemporalQuery` em um `LocalTime` , é lançada uma exceção:

```
java.time.temporal.UnsupportedTemporalTypeException: Unsupported field: DayOfWeek
```

A mensagem diz que o campo `DayOfWeek` não é suportado. E, de fato, `LocalTime` não possui nenhum campo de data, por isso não tem como obter o dia da semana.

Neste exemplo, o `TemporalQuery` retorna um tipo nativo da linguagem (`Boolean`), mas podemos criar implementações que retornam instâncias de qualquer classe, de acordo com a necessidade.

Por que não usar um método?

O `TemporalQuery` que criamos no exemplo anterior poderia muito bem ser substituído por um método estático em uma classe utilitária. Qual a diferença, então?

Do ponto de vista do valor retornado, nenhuma. Mas usando `TemporalQuery` podemos por exemplo escolher a implementação em tempo de execução — técnica também conhecida como *Strategy Pattern* (https://en.wikipedia.org/wiki/Strategy_pattern/). Além disso, o código fica mais testável, pois o `TemporalQuery` pode ser substituído por *mocks* nos seus *unit tests* — caso você não esteja familiarizado com *mocks*, este artigo é uma boa introdução (http://blog.caelum.com.br/testes-unitarios-com-jmock-2/).

Mas caso você já tenha um código legado com este método estático, é possível usá-lo como se fosse um `TemporalQuery` , graças ao recurso de *method reference* do Java 8. O único pré-requisito é que a assinatura do método seja a mesma do método `queryFrom()` da interface `TemporalQuery` . Ou seja, o parâmetro deve ser um `TemporalAccessor` e algum valor deve ser retornado.

A seguir temos um exemplo de uso do *method reference*. Neste código também é usado um `java.util.EnumSet` com os valores de `DayOfWeek` que queremos verificar (uma alternativa ao `if` do exemplo anterior). Além disso, é usado `DayOfWeek.from(temporal)` , que retorna um `DayOfWeek` baseado nos valores da variável `temporal` (com isso, não precisamos usar `get()` para pegar o valor numérico do campo). Estas alterações servem para mostrar uma outra forma de implementar a mesma lógica do exemplo anterior:

```
public class DateUtils {
    // Como DayOfWeek é um enum, posso usá-lo em um EnumSet
    private static final EnumSet<DayOfWeek> FIM_DE_SEMANA
        = EnumSet.of(DayOfWeek.SATURDAY, DayOfWeek.SUNDAY);

    public static boolean isFimDeSemana(TemporalAccessor temporal) {
        // obter um DayOfWeek ao invés do valor numérico do campo
        DayOfWeek diaDaSemana = DayOfWeek.from(temporal);
        // verificar se é sábado ou domingo
        return FIM_DE_SEMANA.contains(diaDaSemana);
    }
}

// usar method reference como um TemporalQuery
boolean fds = LocalDate.now().query(DateUtils::isFimDeSemana);
```

Já vimos todos os tipos de data e hora?

Eu diria que vimos os principais. Há vários outros disponíveis na API, para as mais variadas situações:

- `java.time.YearMonth` : possui apenas ano e mês, sendo útil para representar datas de validade de cartões de crédito, por exemplo.
- `java.time.MonthDay` : possui apenas mês e dia, podendo representar uma data recorrente (por exemplo: `MonthDay natal = MonthDay.of(12, 25)`).

Entre outros. Cada classe possui alguns métodos para converter para outras classes (`MonthDay` , por exemplo, possui o método

`atYear()` , que recebe o ano e retorna um `LocalDate`). Além disso, todas possuem o método `from()` , que recebe um `TemporalAccessor` e extrai os campos necessários para criar uma instância:

```
LocalDate data = LocalDate.of(2018, 10, 4);
// cada classe extrai os campos que precisa para criar uma instância
MonthDay md = MonthDay.from(data); // mês 10, dia 4
YearMonth ym = YearMonth.from(data); // ano 2018, mês 10
```

Os princípios básicos são os mesmos das demais classes que já vimos: métodos `now()` para a data/hora atual, *getters*, métodos `withXXX` para alterar algum campo etc.

No próximo capítulo veremos como a API `java.time` implementa a aritmética de datas.

Capítulo 16
Aritmética de data e durações no java.time

No `java.time` , as classes de data e hora possuem métodos para somar e subtrair durações em campos específicos. Os nomes destes métodos sempre começam com `plus` (para somar) e `minus` (para subtrair).

As implementações seguem aquelas regras estranhas e contraintuitivas que vimos anteriormente no capítulo sobre aritmética de datas. Vamos ver alguns exemplos.

16.1 Somar e subtrair unidades de data e hora

As classes de data e hora possuem vários métodos específicos para somar e subtrair o valor de um único campo. No próximo exemplo temos alguns destes métodos da classe `LocalDateTime` : `plusDays()` para somar dias e `minusHours()` para subtrair horas.

```
// 2018-05-04T17:00
LocalDateTime dataHora = LocalDateTime.of(2018, 5, 4, 17, 0);
// somar 1 dia -> 2018-05-05T17:00
LocalDateTime diaSeguinte = dataHora.plusDays(1);
// subtrair 3 horas -> 2018-05-04T14:00
LocalDateTime tresHorasAntes = dataHora.minusHours(3);
```

Como nesta API os objetos são imutáveis, todos os métodos `plusXXX()` e `minusXXX()` retornam **outra** instância com o resultado da operação. A instância original (no caso, a variável `dataHora`) permanece inalterada.

Um detalhe é que estes métodos aceitam valores negativos, o que equivale a chamar a operação inversa. Por exemplo, `plusDays(-2)` equivale a `minusDays(2)` .

Ajustes para manter a semântica

Já comentamos anteriormente sobre a importância de manter a semântica das operações na aritmética de datas. Por exemplo, ao somar 1 mês, faz todo o sentido que o resultado esteja no mês seguinte. Por isso, os devidos ajustes devem ser feitos na data resultante, como mostra o próximo exemplo:

```
// 2018-01-31
LocalDate jan = LocalDate.of(2018, 1, 31);
// 2018-02-28
LocalDate fev = jan.plusMonths(1);
```

Ao somar 1 mês a 31 de janeiro de 2018, o resultado seria 31 de fevereiro. Mas como fevereiro não tem 31 dias, o resultado é ajustado para o último dia válido deste mês. Então o valor final é 28 de fevereiro de 2018.

Cuidado com os timezones, caso tenha

Nos exemplos anteriores usamos as classes locais, que não possuem um timezone e por isso não sofrem os efeitos de *gaps* e *overlaps*. Mas se usarmos um `ZonedDateTime` , temos que nos atentar para esses detalhes. No próximo exemplo veremos a diferença entre somar 1 dia ou 24 horas, quando ocorre um *DST gap*:

```
// 2017-10-14T10:00-03:00[America/Sao_Paulo]
ZonedDateTime z = ZonedDateTime.of(2017, 10, 14, 10, 0, 0, 0,
ZoneId.of("America/Sao_Paulo"));
// 2017-10-15T10:00-02:00[America/Sao_Paulo]
System.out.println(z.plusDays(1));
// 2017-10-15T11:00-02:00[America/Sao_Paulo]
System.out.println(z.plusHours(24));
```

Primeiro, é criado um `ZonedDateTime` que corresponde a 14 de outubro de 2017, às 10:00 no timezone `America/Sao_Paulo` (o offset é `-03:00` , pois neste dia não está em horário de verão).

Ao somar 1 dia, o resultado é o dia seguinte (15 de outubro), no mesmo horário (10:00). Mas como no dia 15 o timezone está em horário de verão, o offset é ajustado para `-02:00` . Por isso, o resultado é `2017-10-15T10:00-02:00[America/Sao_Paulo]` .

Ao somar 24 horas, o resultado é dia 15, às 11:00 (em vez de 10:00). Isso acontece porque à meia-noite do dia 15 os relógios são adiantados para 01:00 (uma hora é pulada). Basta imaginar um cronômetro que é ligado às 10:00 do dia 14. Quando os relógios são adiantados para 01:00, o cronômetro não tem uma hora adicionada à sua contagem. Ele só marcará 24 horas às 11:00 do dia 15, e o resultado é `2017-10-15T11:00-02:00[America/Sao_Paulo]` .

Quando estiver usando um `ZonedDateTime` , atente para estes casos. Se estiver usando qualquer outra classe que não possui um timezone (inclusive `OffsetDateTime` , que só possui o offset) esta preocupação não existe.

16.2 Somando qualquer unidade de tempo

Assim como existem *getters* genéricos (`get()` e `getLong()`), existem também os métodos genéricos para somar e subtrair qualquer unidade de tempo: `plus()` e `minus()` . Estes métodos estão definidos na interface `java.time.temporal.Temporal` , que é uma subinterface de `TemporalAccessor` e é implementada por todos os tipos de data e hora.

Estes métodos recebem dois parâmetros: a quantidade a ser adicionada ou subtraída (um valor numérico do tipo `long`) e a unidade de tempo correspondente (representada pela interface `java.time.temporal.TemporalUnit`).

Diferença entre TemporalField e TemporalUnit

Na primeira parte do livro, no capítulo sobre aritmética de datas, vimos que datas e durações são dois conceitos diferentes. Datas

representam pontos específicos na linha do tempo, enquanto durações representam quantidades de tempo, sem qualquer relação com calendários. Embora ambas usem as mesmas palavras ("dias", "minutos" etc.) elas não são a mesma coisa.

Na API legada, estes conceitos não são separados, pois as constantes de `Calendar` são usadas tanto para obter o valor de um campo (no método `get()`) quanto para somar valores a uma data (no método `add()`).

No `java.time` os conceitos foram devidamente separados. `TemporalField` , como já vimos, representa um campo específico de data ou hora, como o dia do mês, o minuto etc. E `TemporalUnit` representa uma unidade de tempo decorrido (um campo específico de uma duração). A API possui algumas implementações de `TemporalUnit` , e a que possui as unidades mais comuns do dia a dia é `java.time.temporal.ChronoUnit` . Vamos entender melhor como usá-las.

Usando ChronoUnit

Assim como `ChronoField` é útil para obter campos que não possuem um *getter* específico, um `ChronoUnit` é útil para somar ou subtrair unidades para as quais não há um método `plus` ou `minus` específico.

Por exemplo, `LocalTime` possui o método `plusNanos()` para somar uma quantidade de nanossegundos. Mas e se quisermos somar uma quantidade de milissegundos? Uma alternativa é transformar este valor em nanossegundos (multiplicando por 1 milhão) e passar para `plusNanos()` , mas também podemos usar o `ChronoUnit` correspondente aos milissegundos no método `plus()` :

```
// 10:30:00.123456789
LocalTime hora = LocalTime.of(10, 30, 0, 123456789);
// somar 200 milissegundos -> 10:30:00.323456789
hora = hora.plus(200, ChronoUnit.MILLIS);
```

`ChronoUnit.MILLIS` é a unidade que corresponde a uma duração em milissegundos. Ela **não** é a mesma coisa que

`ChronoField.MILLI_OF_SECOND` (que representa o campo dos milissegundos de um horário). Apesar de os nomes serem muito parecidos, a diferença conceitual entre eles é a mesma que existe entre durações e datas.

`LocalTime` sempre guarda o valor das frações de segundo em nanossegundos. Ao somar 200 milissegundos, o método `plus()` converte o valor para 200000000 nanossegundos e em seguida faz a soma, resultando em `10:30:00.323456789` .

Cuidado para não somar unidades não suportadas

O que acontece se tentarmos somar 1 mês em um `LocalTime` ?

```
LocalTime hora = LocalTime.now().plus(1, ChronoUnit.MONTHS);
```

Como `LocalTime` não possui campos de data, não tem como somarmos meses. Por isso, este código lança uma exceção, dizendo que esta unidade não é suportada:

```
java.time.temporal.UnsupportedTemporalTypeException: Unsupported unit:
Months
```

Podemos verificar se uma `TemporalUnit` é suportada usando o método `isSupported()` (o mesmo que usamos com `TemporalField`):

```
boolean suporta = LocalTime.now().isSupported(ChronoUnit.MONTHS)); //
false
```

Neste caso, a unidade `MONTHS` não é suportada e o método retorna `false` . Portanto, esta `ChronoUnit` não pode ser usada nos métodos `plus()` e `minus()` de um `LocalTime` .

16.3 Classes específicas para durações

Na API legada há classes para representar datas (`Date` , `Calendar` e as subclasses de `Date` no pacote `java.sql`), mas não existe nenhuma

para representar durações. No `java.time` foram criadas duas classes, que veremos a partir de agora: `java.time.Period` e `java.time.Duration` .

Period

A classe `Period` representa uma duração em termos de anos, meses e dias. É possível criá-la a partir dos valores numéricos de cada uma destas unidades, ou fazendo o parsing de uma `String` no formato ISO 8601. No código a seguir temos um exemplo de cada:

```
// 2 jeitos de criar um período de "1 ano, 2 meses e 20 dias"
System.out.println(Period.parse("P1Y2M20D"));
System.out.println(Period.of(1, 2, 20));
```

Ao imprimir o `Period` , o método `toString()` é chamado e o retorno estará no formato ISO 8601. Por isso este código imprime `P1Y2M20D` duas vezes.

Além dos métodos `of()` e `parse()` , há outros para criar um `Períod` com o valor de somente uma das unidades, por exemplo, `Period.ofYears(2)` , que cria um período de 2 anos, `Period.ofDays(10)` , que cria um período de 10 dias etc. É possível também usar a quantidade de semanas, que internamente é convertida para dias: `Period.ofWeeks(2)` , por exemplo, é o mesmo que `Period.ofDays(14)` .

Como o `Period` não possui as unidades relacionadas ao horário (horas, minutos etc.) o método `parse()` não funciona com `Strings` que tenham estes valores. Por exemplo, `Period.parse("P1DT1H")` (1 dia e 1 hora) lança um `DateTimeParseException` , pois possui um valor para as horas.

Somar valores a um Period

Um `Period` representa uma quantidade de tempo. Sendo assim, podemos somar outra quantidade de tempo a ele, resultando em outro `Period` . Por exemplo, se tivermos um período de 1 mês e quisermos somar 40 dias, basta fazer o seguinte:

```
// somar 40 dias ao periodo de 1 mês
Period period = Period.ofMonths(1);
period = period.plusDays(40);
```

No `java.time` todas as classes são imutáveis, então o método `plusDays()` retorna outra instância. É necessário atribuir o retorno em alguma variável (no caso, eu atribuí para a mesma). O resultado será um período de 1 mês e 40 dias (`P1M40D`).

Repare que os 40 dias não foram convertidos para *"1 mês e X dias"*, já que um mês pode ter de 28 a 31 dias e por isso não é possível saber exatamente o valor do *X*. Sendo assim, o valor `40` permanece o mesmo.

Note que isso **não** é a mesma aritmética de datas que vimos anteriormente, pois não estamos somando o `Period` a uma data (resultando em outra data). Estamos simplesmente somando duas quantidades de tempo, resultando em outra quantidade.

Um detalhe interessante é que em vez de somar um `Period` com um valor numérico, podemos somar duas instâncias de `Period` diretamente, usando o método `plus()` :

```
// 1 ano e 3 meses
Period period = Period.parse("P1Y3M");
// somar 5 meses e 10 dias
period = period.plus(Period.parse("P5M10D"));
```

Eu somei "1 ano e 3 meses" com "5 meses e 10 dias", e o resultado é "1 ano, 8 meses e 10 dias" (`P1Y8M10D`). Para obter os valores de cada unidade separadamente, basta usar os respectivos *getters*. Para o `Period` do exemplo anterior, `getYears()` retorna `1` , `getMonths()` retorna `8` e `getDays()` retorna `10` .

Duration

A classe `Duration` representa uma duração em termos de segundos e nanossegundos. Apesar disso, é possível criá-la a partir da quantidade de minutos, horas e dias, e internamente esses valores

são convertidos para segundos. Outra forma de criar um `Duration` é fazendo o parsing de uma `String` no formato ISO 8601. No código a seguir temos um exemplo de cada:

```
// 2 modos de criar uma duração de 10 minutos
System.out.println(Duration.parse("PT10M"));
System.out.println(Duration.ofMinutes(10));
```

Ao imprimir o `Duration` , o método `toString()` é chamado e retorna a duração no formato ISO 8601. Internamente a classe guarda o valor em segundos e nanossegundos (ou seja, neste caso seu valor é de 600 segundos) mas o método `toString()` retorna a `String` devidamente "quebrada" em dias, horas, minutos e segundos. Por isso, este código imprime `PT10M` (10 minutos) duas vezes.

Além disso, as unidades referentes ao ano e mês não são suportadas, então chamar o método `parse()` com `Strings` como `P1Y` (1 ano) ou `P1M` (1 mês) lança um `DateTimeParseException` . Já uma duração em dias (`P1D`) é convertida para segundos (considerando que 1 dia tem 24 horas, ou seja, 86.400 segundos).

Somar valores a um Duration

De maneira similar ao que é feito com `Period` , é possível somar valores a um `Duration` . O único detalhe é que as unidades referentes a dias, horas e minutos são convertidas em segundos. No próximo exemplo são usados os vários métodos `plus` para construir um `Duration` equivalente a "3 horas, 10 minutos e 3.5 segundos":

```
Duration duracao = Duration.ofSeconds(3) // 3 segundos
    // somar 10 minutos (usando o valor numérico)
    .plusMinutes(10)
    // somar 0.5 segundos (usando outro Duration)
    .plus(Duration.ofMillis(500))
    // somar 3 horas (usando ChronoUnit)
    .plus(3, ChronoUnit.HOURS);
System.out.println(duracao); // PT3H10M3.5S
```

Neste código vemos que é possível usar um método específico para somar minutos (`plusMinutes()`), além de duas versões do método `plus()` : uma que recebe um `Duration` e outra que recebe um valor numérico e a respectiva `ChronoUnit` .

A duração é impressa no formato ISO 8601, por isso a saída é `PT3H10M3.5S` . Apesar disso, internamente o `Duration` só possui os valores dos segundos e nanossegundos, que podem ser acessados pelos respectivos *getters*: `getSeconds()` retorna `11403` (3 horas, 10 minutos e 3 segundos equivalem a 11403 segundos) e `getNano()` retorna `500000000` (0.5 segundos equivalem a 500 milhões de nanossegundos).

Há também o método `toHours()` , que retorna a quantidade de horas (neste caso, `3`). Mas há uma "pegadinha" nos métodos `toXXX()` . Por exemplo, se usarmos `toMinutes()` nesta duração, o retorno será `190` , pois este é o valor obtido convertendo-se o total de segundos (11403) para minutos (note que o valor é arredondado).

Mas a duração corresponde a "3 horas, **10** minutos e 3.5 segundos". Como obter o valor `10` para os minutos? No Java 8, o único jeito é obter o total de segundos e fazer as contas manualmente. A partir do Java 9, basta usar o método `toMinutesPart()` — nesta versão do JDK foram adicionados vários métodos `toXXXPart()` , que retornam os valores de cada unidade de tempo separadamente. Para mais detalhes, veja o *JDK Bug System* (https://bugs.openjdk.java.net/browse/JDK-8142936/).

16.4 Somar durações a uma data

Já vimos vários jeitos de somar e subtrair quantidades de tempo a uma data. Temos os métodos específicos para uma única unidade de tempo (como `plusDays()` e `minusHours()`) e os métodos `plus()` e `minus()` que recebem um valor e a respectiva `TemporalUnit` .

Além destes, os métodos `plus()` e `minus()` também podem receber como parâmetro a interface `java.time.temporal.TemporalAmount` . Esta interface representa uma quantidade de tempo e a API possui duas classes que a implementam: `Duration` e `Period` .

Portanto, é possível somar e subtrair um `Period` ou um `Duration` diretamente a uma data, como podemos ver no próximo exemplo:

```
// 3 horas, 10 minutos e 3.5 segundos
Duration duracao = Duration.parse("PT3H10M3.5S");
// 1 mês e 10 dias
Period periodo = Period.parse("P1M10D");
// 2018-05-04T17:00
LocalDateTime dataHora = LocalDateTime.of(2018, 5, 4, 17, 0);
dataHora = dataHora
    // somar o Duration -> 2018-05-04T20:10:03.500
    .plus(duracao)
    // subtrair o Period -> 2018-03-25T20:10:03.500
    .minus(periodo);
```

Somar ou subtrair um `Duration` ou um `Period` , em linhas gerais, equivale a fazer a mesma operação para cada unidade de tempo separadamente, uma após a outra. Como `Duration` guarda a duração em segundos e nanossegundos, `plus(duracao)` é o mesmo que somar o valor de `getSeconds()` e, em seguida, somar o valor de `getNano()` . E `minus(periodo)` é o mesmo que subtrair primeiro os anos, depois os meses e, por fim, os dias.

Cuidado com unidades não suportadas

`Duration` trabalha com a duração em termos de segundos e nanossegundos, enquanto `Period` trabalha com os anos, meses e dias. Por isso, não faz sentido somá-los ou subtraí-los a uma classe que não possui os campos correspondentes.

Por exemplo, `LocalDate` só possui os campos de data, por isso não é possível somar um `Duration` a ele. Da mesma forma que não é possível somar um `Period` a um `LocalTime` , pois esta classe não

possui os campos de data. Qualquer uma destas tentativas lança um `UnsupportedTemporalTypeException`.

Infelizmente não é possível passar um `Period` ou `Duration` para o método `isSupported`, para saber se é possível somá-los ou subtraí-los. Mas a partir destes objetos podemos obter uma lista de `TemporalUnit` (usando o método `getUnits()`) e verificar se todas as unidades são suportadas:

```
LocalDateTime data = LocalDateTime.now();
boolean suportaDuracao = Duration.parse("PT1H30M4.5S")
    // obter as TemporalUnits da duracao
    .getUnits()
    // verificar se todas são suportadas pela data
    .stream().allMatch(data::isSupported);
```

No caso de `Duration`, o método `getUnits()` retorna uma lista com `ChronoUnit.SECONDS` e `ChronoUnit.NANOS` (segundos e nanossegundos). Como ambos são suportados por `LocalDateTime`, o valor de `suportaDuracao` é `true`.

Um dia nem sempre tem 24 horas

Ao longo do livro, vimos vários exemplos mostrando como os *gaps* e *overlaps* dos timezones fazem com que 1 dia nem sempre seja o equivalente a 24 horas. Vamos usar novamente um desses casos para ver como `Duration` e `Period` se comportam de maneira diferente quando há um timezone em ação.

Para isso, vamos usar o timezone `America/Sao_Paulo` e criar um `ZonedDateTime` com a data próxima ao início do horário de verão em outubro de 2017. A seguir, veremos a diferença entre somar um `Period` e um `Duration`, ambos correspondendo a 1 dia:

```
ZoneId zone = ZoneId.of("America/Sao_Paulo");
// 2017-10-14T10:00-03:00[America/Sao_Paulo]
ZonedDateTime z = ZonedDateTime.of(2017, 10, 14, 10, 0, 0, 0, zone);
System.out.println(z.plus(Period.ofDays(1)));
System.out.println(z.plus(Duration.ofDays(1)));
```

Somar `Period.ofDays(1)` leva em conta o horário de verão e tem o mesmo efeito semântico de chamar `plusDays(1)` . O resultado é **o mesmo horário do dia seguinte**, com os devidos ajustes no offset: `2017-10-15T10:00-02:00[America/Sao_Paulo]` .

`Duration` , por sua vez, converte os dias para segundos, sempre considerando que um dia tem 24 horas. Somar `Duration.ofDays(1)` tem o mesmo efeito de `plusHours(24)` e o resultado é `2017-10-15T11:00-02:00[America/Sao_Paulo]` .

O fato de `Duration` e `Period` terem abordagens diferentes com relação a *gaps* e *overlaps* é um dos motivos para a API não ter feito uma única classe para representar durações. Inicialmente, a ideia era ter apenas uma, mas depois foi decidido separar em duas, conforme o próprio Stephen Colebourne comenta nesta resposta do Stack Overflow (https://stackoverflow.com/a/32369144/).

Caso você precise trabalhar com durações que possuem unidades de data e hora (como `P1M20DT5H3S`), uma alternativa é usar a biblioteca ThreeTen-Extra (http://www.threeten.org/threeten-extra/), que possui a classe `PeriodDuration` (basicamente, uma junção de `Period` e `Duration`).

16.5 Calcular diferenças entre datas

Há dois modos principais de calcular a diferença entre dois objetos que representam data e/ou hora. Vamos a eles.

Obter o valor numérico de apenas uma unidade de tempo

Suponha que só queremos saber quantos dias há entre duas datas (apenas o valor numérico, nada mais). Para isso, basta usar a `ChronoUnit` correspondente aos dias e chamar o método `between()` , passando como parâmetro as duas datas a serem verificadas:

```
LocalDate inicio = LocalDate.of(2018, 1, 1);
LocalDate fim = LocalDate.of(2018, 1, 10);
// quantos dias entre 1 e 10 de janeiro
long dias = ChronoUnit.DAYS.between(inicio, fim); // 9
```

Neste exemplo calculamos quantos dias há entre 1 e 10 de janeiro de 2018. O resultado é `9` , pois o cálculo sempre inclui a primeira data e exclui a última (*start inclusive, end exclusive*). Esta regra é usada por todos os métodos da API que calculam diferença entre datas e horas.

Outro detalhe é que usamos a classe `LocalDate` , que só possui os campos de data. Mas se usarmos uma classe que possui as horas (como um `LocalDateTime`), elas também são levadas em consideração no cálculo. Exemplo:

```
// 2018-01-01T11:00
LocalDateTime inicio = LocalDateTime.of(2018, 1, 1, 11, 0);
// 2018-01-02T10:59:59.999999999
LocalDateTime fim = LocalDateTime.of(2018, 1, 2, 10, 59, 59, 999999999);
long dias = ChronoUnit.DAYS.between(inicio, fim); // 0
```

Neste exemplo calculamos quantos dias há entre 1 de janeiro de 2018, às 11:00 e 2 de janeiro de 2018, às 10:59:59.999999999. Como o horário faz parte do cálculo, a API considera que somente às 11:00 do dia 2 terá completado 1 dia. Mesmo que falte apenas 1 nanossegundo para as 11:00, o resultado é arredondado para baixo, e por isso o valor retornado é zero.

Para desconsiderar o horário, basta transformar cada instância de `LocalDateTime` em um `LocalDate` , usando o método `toLocalDate()` :

```
// calcular diferença, ignorando o horário
long dias = ChronoUnit.DAYS.between(inicio.toLocalDate(),
fim.toLocalDate()); // 1
```

Com isso, o horário não é mais considerado (pois `LocalDate` só possui os campos de data) e o resultado é `1` .

Vale lembrar que usar uma unidade não suportada pelas classes em questão vai lançar uma exceção. Por exemplo, se calcularmos quantos minutos há entre duas instâncias de `LocalDate` :

```
LocalDate inicio = LocalDate.of(2018, 1, 1);
LocalDate fim = LocalDate.of(2018, 1, 10);
long minutos = ChronoUnit.MINUTES.between(inicio, fim);
```

Como `LocalDate` não possui campos de horário, não é possível calcular a diferença em minutos. Então, este código lança uma exceção, dizendo que a `ChronoUnit` não é suportada:

```
java.time.temporal.UnsupportedTemporalTypeException: Unsupported unit:
Minutes
```

Obter a diferença como um Period ou Duration

Outro modo de calcular a diferença entre datas é usando `Period` e `Duration` . Ambos possuem o método `between()` , com uma diferença:

- `Period.between()` aceita apenas `LocalDate` como parâmetros, já que esta classe calcula a diferença em termos de anos, meses e dias.
- `Duration.between()` aceita como parâmetros quaisquer classes que implementem a interface `Temporal` . O valor final é convertido para segundos e nanossegundos.

Exemplo:

```
// 2018-01-01T11:00
LocalDateTime inicio = LocalDateTime.of(2018, 1, 1, 11, 0);
// 2018-02-10T10:00
LocalDateTime fim = LocalDateTime.of(2018, 2, 10, 10, 0);
// P1M9D - 1 mês e 9 dias
Period periodo = Period.between(inicio.toLocalDate(), fim.toLocalDate());
// PT959H - 959 horas
Duration duracao = Duration.between(inicio, fim);
```

Como `Period.between()` só aceita `LocalDate` como parâmetros, tivemos que usar o método `toLocalDate()` para converter as datas para este

tipo. O horário é desconsiderado e a diferença é de 1 mês e 9 dias. Para conferir se o resultado está correto, basta pegar a data inicial (`2018-01-01`) e somar 1 mês: o resultado é `2018-02-01` . Em seguida, somamos 9 dias e o resultado é a data final (`2018-02-10`).

`Duration.between()` , por sua vez, não faz o cálculo em termos de meses ou anos. Esta classe considera que os dias têm sempre 24 horas e no final converte a diferença para segundos e nanossegundos. Neste caso, o `Duration` corresponde a um total de 3452400 segundos, que equivale a exatas 959 horas (`PT959H`).

Um pequeno detalhe é que a documentação de `Duration.between()` menciona que os seus parâmetros devem suportar a unidade `ChronoUnit.SECONDS` , caso contrário o método lança um `DateTimeException` . Por isso, não é possível usar `LocalDate` neste método, por exemplo.

A diferença pode ser negativa

Caso a data inicial seja maior que a data final, o resultado do método `between()` é negativo. Exemplo:

```
LocalDate inicio = LocalDate.of(2018, 1, 1);
LocalDate fim = LocalDate.of(2018, 1, 10);
// quantos dias entre 10 e 1 de janeiro
long dias = ChronoUnit.DAYS.between(fim, inicio); // -9
// período de -9 dias (P-9D)
Period periodo = Period.between(fim, inicio);
```

Tanto `Period` quanto `Duration` podem ter valores negativos. Ambos possuem o método `isNegative()` , que retorna `true` caso o valor seja negativo, e `negated()` , que retorna outra instância contendo todos os valores com o sinal invertido. Estes métodos são úteis se você precisar somente dos valores absolutos da diferença, sem se importar com a ordem das datas.

Nunca se esqueça de que aritmética de datas é estranha

Vamos criar um `LocalDate` correspondente a 29 de fevereiro de 2016 e somar 1 ano:

```
// 2016-02-29
LocalDate data = LocalDate.of(2016, 2, 29);
// 2017-02-28
LocalDate umAnoDepois = data.plusYears(1);
```

Ao somar 1 ano, o resultado *seria* 29 de fevereiro de 2017. Mas como 2017 não é um ano bissexto, o dia é ajustado para 28 e o resultado é `2017-02-28` . Agora vamos calcular quantos anos há entre estas duas datas, usando `ChronoUnit` e `Period` :

```
System.out.println(ChronoUnit.YEARS.between(data, umAnoDepois));
System.out.println(Period.between(data, umAnoDepois));
```

Curiosamente, o resultado não será 1 ano:

```
0
P11M30D
```

`ChronoUnit.YEARS` considera que somente a partir do dia 29 de fevereiro de 2017 seria completado 1 ano. Como fevereiro tem 28 dias em 2017, o método só considera que a diferença é de 1 ano para datas a partir de 1 de março. Por isso, o método `between()` retorna zero.

`Period.between()` , por sua vez, retorna um período de 11 meses e 30 dias, que não necessariamente é igual a 1 ano (`P1Y`), já que 30 dias nem sempre são suficientes para se completar 1 mês.

Mas o resultado não deveria ser 1 ano? Depende. Basta imaginar uma pessoa que nasceu em 29 de fevereiro de 2016. Em 28 de fevereiro de 2017 ela já terá completado 1 ano?

O `java.time` entende que não. Outras APIs podem achar que sim. Não há uma regra oficial que determina como a aritmética de datas deve funcionar (como há para a matemática) e cada API vai implementar de um jeito diferente. O Joda-Time (http://www.joda.org/joda-time), por exemplo, considera que a

diferença entre estas datas é de 1 ano. Cabe a você conhecer a API que está usando e escrever o seu código de acordo com cada situação.

Com isso, acredito que já vimos os principais casos de aritmética de datas e durações. A API ainda permite usos mais avançados, como criar suas próprias implementações de `TemporalAmount` e `TemporalUnit`. Um exemplo seria criar classes que representam durações específicas, como décadas ou séculos, de maneira similar ao que é feito no projeto ThreeTen Extra, que possui a classe `Years`, que representa uma duração em anos (https://www.threeten.org/threeten-extra/apidocs/org/threeten/extra/Years.html). A criação de tais classes está fora do escopo deste livro, mas o código da API ThreeTen Extra está disponível no GitHub e é uma ótima fonte de estudo para estes casos (https://github.com/ThreeTen/threeten-extra).

No próximo capítulo veremos como o `java.time` faz a formatação de datas.

Capítulo 17
Formatação com java.time

No `java.time` a classe responsável pela formatação de datas é `java.time.format.DateTimeFormatter` , que possui várias melhorias com relação à API anterior. Uma delas é o fato de ser *thread-safe* e não precisar de todos aqueles controles de sincronização que fizemos com `SimpleDateFormat` . As outras, veremos a seguir.

17.1 Usando pattern e locales

O jeito mais simples de criar um `DateTimeFormatter` é através do método `ofPattern()` , que recebe como parâmetro uma `String` contendo o pattern a ser usado para formatar. Exemplo:

```
LocalDate data = LocalDate.of(2018, 5, 4);
// usar pattern para "dia/mês/ano"
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/uuuu");
// as duas formas abaixo são equivalentes
System.out.println(formatter.format(data));
System.out.println(data.format(formatter));
```

Repare que podemos formatar de duas maneiras: usando o método `format()` da classe `DateTimeFormatter` passando a data como parâmetro, ou usando o método `format()` de `LocalDate` passando o `DateTimeFormatter` como parâmetro. Ambas são equivalentes e retornam uma `String` com a data formatada. Esse código imprime `04/05/2018` duas vezes.

Muita atenção com o pattern

No exemplo anterior o pattern `dd/MM/uuuu` **não** é o mesmo que usaríamos com `SimpleDateFormat` para obter a data neste formato. Devemos ter cuidado ao migrar nosso código para a nova API.

Apesar de várias letras do pattern serem as mesmas nas duas APIs, nem todas são assim. Algumas funcionam de outro jeito, outras correspondem a campos diferentes, e vários patterns novos foram adicionados no `java.time`. Sempre consulte a documentação para ver o que cada letra significa e qual o seu funcionamento (https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html/).

A letra `u`, por exemplo, corresponde ao dia da semana na API legada, mas no `java.time` corresponde ao ano. Aliás, o ano pode ser representado de duas maneiras na nova API: existe o campo *year-of-era*, representado pela letra `y` e por `ChronoField.YEAR_OF_ERA`, e o campo *year*, representado pela letra `u` e por `ChronoField.YEAR`.

A diferença está na forma como são tratados os anos negativos (ou "AC" – o popular "Antes de Cristo"), conforme podemos ver no próximo exemplo:

```
LocalDate data = LocalDate.of(-10, 1, 1);
DateTimeFormatter fmt = DateTimeFormatter.ofPattern("uuuu yyyy GGGG");
System.out.println(fmt.format(data));
```

Foi criada uma data no ano `-10` e formatada usando `uuuu` e `yyyy`. Também foi usado o pattern `G`, que corresponde ao campo "Era" (que indica as eras Antes e Depois de Cristo). A saída é:

```
-0010 0011 Antes de Cristo
```

Esta diferença acontece porque `y` considera que não houve ano zero: o primeiro ano da era em que estamos atualmente é o "ano 1 Depois de Cristo", e o ano anterior a este é o "ano 1 Antes de Cristo". Já o pattern `u` considera que antes do ano 1 vem o ano zero e antes disso os anos são negativos. Em resumo:

`y` e `G`	`u`
2 DC	2
1 DC	1

y e G	u
1 AC	0
2 AC	-1
3 AC	-2
E assim por diante...	

Por isso, o ano -10 é impresso como `-0010` para o pattern `uuuu` e `0011` para o pattern `yyyy` . Só que o valor `0011` é ambíguo, pois pode ser tanto 11 AC quanto 11 DC. Somente sabendo a era (`G`) essa ambiguidade é resolvida.

Então devo usar `u` ou `y` ? Há argumentos a favor de ambos. Por um lado, na grande maioria dos casos, `y` funciona normalmente para datas atuais, e a maior parte dos sistemas não trabalha com datas Antes de Cristo (além de funcionar da mesma maneira com `SimpleDateFormat`). Por outro lado, o pattern `y` precisa que uma era (AC ou DC) esteja definida, para que seu valor não seja ambíguo, e nem sempre esta informação está disponível (no próximo capítulo veremos um exemplo).

Nesse caso, usar `u` resolve a ambiguidade e elimina a necessidade de definir uma era. Mas há quem argumente que `u` é "menos intuitivo" que `y` , já que este remete a *year*, bem mais "fácil de lembrar". Há uma boa discussão sobre isso nesta pergunta do Stack Overflow (https://stackoverflow.com/q/41177442/). Nos exemplos a seguir usarei `u` , mas eles também funcionam com `y` , por só usarem datas Depois de Cristo.

Nem sempre será possível formatar

Há outra diferença importante com relação à API legada. `Date` representa um timestamp e `SimpleDateFormat` sempre usa algum timezone (se nenhum é informado, ele usa o padrão da JVM). Por isso, ao formatar um `Date` , sempre é possível obter os valores da

data, hora e offset. Mas no `java.time` há várias classes diferentes e nem todas possuem todos os campos, então devemos tomar cuidado ao formatar uma data ou hora. O exemplo a seguir usa um pattern com a hora e o minuto e tenta formatar um `LocalDate` :

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("HH:mm");
System.out.println(formatter.format(LocalDate.now()));
```

O pattern `HH:mm` significa "horas:minutos", mas `LocalDate` só possui os campos de data. Então esse código lança uma exceção:

```
java.time.temporal.UnsupportedTemporalTypeException: Unsupported field:
HourOfDay
```

A mensagem informa que o campo `HourOfDay` (hora do dia: `HH`) não é suportado. E, de fato, `LocalDate` não possui esse campo, por isso não é possível formatá-lo com esse pattern. Sempre que criarmos um `DateTimeFormatter` , devemos nos certificar de que os objetos sendo formatados possuem todos os campos do pattern.

Mudando o idioma com Locale

Quando usamos o método `ofPattern()` , ele cria um `DateTimeFormatter` que usa o locale padrão da JVM. No exemplo anterior que mostra o uso do pattern `GGGG` , o nome da era foi formatado em português (`Antes de Cristo`), pois o locale padrão da minha JVM é `pt_BR` . Mas também é possível definir qual locale deve ser usado (em vez do padrão da JVM), como mostra o próximo exemplo:

```
LocalDate data = LocalDate.of(2018, 5, 4);
DateTimeFormatter formatter = DateTimeFormatter
    // definir o pattern e usar locale pt_BR
    .ofPattern("dd 'de' MMMM 'de' uuuu", new Locale("pt", "BR"));
System.out.println(data.format(formatter));
```

Repare que o texto "de" está entre aspas simples (`'de'`), o que faz com que estas letras sejam literais (a própria letra "d" seguida da letra "e"). Sem as aspas, as letras seriam interpretadas como campos (`d` é o dia do mês e `e` é o dia da semana).

Com o locale devidamente informado no método `ofPattern()` , todos os campos que são *locale sensitive* (como o nome do mês, representado pelo pattern `MMMM`) terão seus valores gerados no idioma correspondente ao `Locale` indicado. A saída é `04 de Maio de 2018` .

Caso já exista um `DateTimeFormatter` , podemos reaproveitar o seu pattern e mudar somente o locale. Para isso usamos o método `withLocale()` , que retorna outra instância com o locale modificado:

```
DateTimeFormatter formatter = DateTimeFormatter
    .ofPattern("dd 'de' MMMM 'de' uuuu", new Locale("pt", "BR"));
// criar outro DateTimeFormatter com o locale inglês (o pattern é o mesmo)
DateTimeFormatter fmtIngles = formatter.withLocale(Locale.ENGLISH);
LocalDate data = LocalDate.of(2018, 5, 4);
System.out.println(fmtIngles.format(data));
```

Com isso, `fmtIngles` usará o locale inglês, mas o pattern será o mesmo usado por `formatter` . A saída é `04 de May de 2018` . O nome do mês está em inglês ("May") e o pattern foi mantido, inclusive o texto literal "de" – o locale só traduz os campos que são *locale sensitive*, o restante permanece como está.

17.2 Não precisa setar o timezone

Como `LocalDate` não possui um timezone, o resultado da formatação é sempre o mesmo, independente do timezone padrão da JVM. Por isso nos exemplos anteriores não foi necessário setar o timezone no `DateTimeFormatter` (como é feito com `SimpleDateFormat`).

Mesmo quando o objeto a ser formatado possui um timezone, o resultado da formatação não é afetado pelo timezone padrão da JVM. O `DateTimeFormatter` sempre usa os valores que estão no objeto, como mostra o próximo exemplo:

```
// mudar timezone padrão para Asia/Tokyo
TimeZone.setDefault(TimeZone.getTimeZone("Asia/Tokyo"));
// 2018-05-04T17:00-03:00[America/Sao_Paulo]
ZonedDateTime agora = ZonedDateTime.now(ZoneId.of("America/Sao_Paulo"));
// usar pattern para "dia/mês/ano hora:minuto"
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/uuuu
HH:mm");
System.out.println(formatter.format(agora));
```

Nesse exemplo, o `ZonedDateTime` corresponde a `2018-05-04T17:00-03:00[America/Sao_Paulo]` . Mesmo que o timezone padrão seja `Asia/Tokyo` , o `DateTimeFormatter` não é afetado por esta configuração e usa os valores que estão no `ZonedDateTime` para formatar. O resultado é `04/05/2018 17:00` .

Se usássemos um `SimpleDateFormat` e não mudássemos o seu timezone, ele usaria o padrão da JVM (`Asia/Tokyo`) e a data e hora seriam ajustadas para este timezone, resultando em `05/05/2018 05:00` . No `java.time` não temos este problema, pois são usados os valores que estão no objeto sendo formatado.

Formatando um Instant

Esta é uma "pegadinha" conhecida da API. O que acontece se eu tentar formatar um `Instant` ?

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/uuuu
HH:mm");
System.out.println(formatter.format(Instant.now()));
```

Esse código lança uma exceção:

```
java.time.temporal.UnsupportedTemporalTypeException: Unsupported field:
DayOfMonth
```

A mensagem diz que não foi possível obter o valor de `DayOfMonth` (o dia do mês, que corresponde ao `dd` do pattern).

Isso acontece porque `Instant` representa um timestamp, portanto pode corresponder a uma data e hora diferentes em cada timezone.

Só que o `DateTimeFormatter` é criado sem um timezone (ao contrário de `SimpleDateFormat` , que sempre usa o padrão da JVM) e por isso ele não tem como obter os valores dos campos de data e hora.

Para formatar o `Instant` , devemos primeiro convertê-lo para algum timezone ou offset, usando os métodos `atZone()` e `atOffset()` , conforme já visto anteriormente. Em seguida, podemos passar o resultado (`ZonedDateTime` ou `OffsetDateTime`) para o `DateTimeFormatter` .

Não precisa setar o timezone, mas se quiser, pode

Existe a opção de setar um timezone no `DateTimeFormatter` usando o método `withZone()` . Com isso, o `Instant` poderá ser formatado normalmente, como mostra o próximo exemplo:

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/uuuu
HH:mm")
    // usar timezone Asia/Tokyo
    .withZone(ZoneId.of("Asia/Tokyo"));
// Instant.now() corresponde a 2018-05-04T20:00:00Z
System.out.println(formatter.format(Instant.now()));
```

O timezone `Asia/Tokyo` foi setado no `DateTimeFormatter` . Ao chamar o método `format()` , o `Instant` é convertido para este timezone. Como o valor do `Instant` (`2018-05-04T20:00:00Z`) corresponde a 5 de maio de 2018, às 05:00 em Tóquio, o resultado é `05/05/2018 05:00` .

Quando o `DateTimeFormatter` possui um timezone, este é usado para converter os valores da data sendo formatada. Mas isso só ocorre com objetos que podem ser convertidos para `Instant` (como `ZonedDateTime` e `OffsetDateTime`). As classes locais, por exemplo, não serão afetadas por esta configuração. O código a seguir mostra esta diferença:

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/uuuu
HH:mm")
    // usar timezone Asia/Tokyo
    .withZone(ZoneId.of("Asia/Tokyo"));
// 2018-05-04T17:00-03:00[America/Sao_Paulo]
```

```
ZonedDateTime agora = ZonedDateTime.now(ZoneId.of("America/Sao_Paulo"));
System.out.println(formatter.format(agora)); // 05/05/2018 05:00
// 2018-05-04T17:00
LocalDateTime dataHora = LocalDateTime.now();
System.out.println(formatter.format(dataHora)); // 04/05/2018 17:00
```

Primeiro criamos um `DateTimeFormatter` que usa o timezone `Asia/Tokyo`. Depois criamos um `ZonedDateTime` com o timezone `America/Sao_Paulo`, mas ao formatá-lo, seus valores foram convertidos para o timezone do `DateTimeFormatter`. Por isso o resultado é `05/05/2018 05:00` (o dia e horário foram ajustados para o timezone `Asia/Tokyo`).

`LocalDateTime`, por sua vez, não é afetado pela configuração do timezone e seus valores não são ajustados. O resultado é `04/05/2018 17:00`.

Pessoalmente, eu prefiro usar o `DateTimeFormatter` sem nenhum timezone e fazer as conversões nos próprios objetos de data e hora. Fica a seu critério usar esse recurso ou não, só recomendo que leia a documentação para saber todos os detalhes quanto ao seu funcionamento.

17.3 Formatar para ISO 8601

Na segunda parte do livro vimos o esforço que é preciso fazer com a API legada para formatar uma data para o formato definido pela norma ISO 8601. No `java.time`, essa dificuldade foi minimizada, pois já existem vários formatadores predefinidos. Todos eles são constantes estáticas da classe `DateTimeFormatter`, como mostra o próximo exemplo:

```
// 2018-05-04T17:00-03:00[America/Sao_Paulo]
ZonedDateTime agora = ZonedDateTime.now(ZoneId.of("America/Sao_Paulo"));
System.out.println(agora.format(DateTimeFormatter.ISO_LOCAL_DATE));
System.out.println(agora.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));
System.out.println(agora.format(DateTimeFormatter.ISO_OFFSET_DATE_TIME));
```

Nesse exemplo, eu usei um `ZonedDateTime` cujo valor é `2018-05-04T17:00-03:00[America/Sao_Paulo]`, e converti para vários formatos diferentes. Repare que há um `DateTimeFormatter` predefinido que retorna somente a data, outro que retorna a data e hora, e outro que retorna a data, hora e offset. A saída é:

```
2018-05-04
2018-05-04T17:00:00
2018-05-04T17:00:00-03:00
```

Há muitos outros formatadores predefinidos disponíveis, todos com nomes que começam com `ISO_`. Consulte a documentação para ver todas as opções, além de uma explicação detalhada dos campos formatados por cada um (https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html). Um detalhe interessante é que, para fazer o parsing do ano, todos usam o campo `ChronoField.YEAR`, que equivale ao pattern `u`.

Mudar formato do offset

A norma ISO 8601 permite que um offset seja representado com os dois pontos (`-03:00`), sem os dois pontos (`-0300`) ou somente com o valor das horas (`-03`). Os formatadores predefinidos sempre usam o formato com dois pontos. Para ter o offset em um formato diferente, podemos usar o pattern `X` ou `x`, cujo resultado depende da quantidade de letras:

- `XXX` ou `xxx` : offset com dois pontos;
- `XX` ou `x` : offset sem dois pontos;
- `X` ou `x` : offset com somente horas.

A diferença entre `X` maiúsculo e `x` minúsculo é a forma como eles tratam o offset zero. O `X` maiúsculo retorna este valor como `Z` (de acordo com a definição da ISO 8601 para designar UTC), enquanto o `x` minúsculo retorna o valor como `+00:00`, `+0000` ou `+00`, dependendo da quantidade de letras.

Segue um exemplo (usando o mesmo `ZonedDateTime` do exemplo anterior):

```
DateTimeFormatter formatter = DateTimeFormatter
    // formato ISO 8601, offset sem os dois pontos
    .ofPattern("uuuu-MM-dd'T'HH:mm:ssXX");
System.out.println(agora.format(formatter));
```

Repare como a letra "T" está entre aspas simples no pattern (`'T'`), pois ela deve ser tratada como um literal (a própria letra "T" em vez de um campo específico). Além disso, foi usado `XX` para o offset. O resultado é `2018-05-04T17:00:00-0300` .

Vale lembrar que usar um offset com somente as horas (`-03`) só é recomendado caso o valor dos minutos seja zero. Para offsets como `+05:30` , formatar com apenas um `X` (ou `x`) fará com que o valor dos minutos seja descartado, pois este será formatado como `+05` .

17.4 Patterns opcionais

Uma novidade introduzida por `DateTimeFormatter` é a possibilidade de usar patterns opcionais. Com isso, os campos só serão formatados caso estejam disponíveis. O pattern opcional é delimitado por colchetes, como mostra o próximo exemplo:

```
// data obrigatória, hora opcional
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/uuuu[
HH:mm:ss]");
// 2018-05-04
LocalDate data = LocalDate.of(2018, 5, 4);
// 2018-05-04T17:30
LocalDateTime dataHora = data.atTime(17, 30);
System.out.println(formatter.format(data));
System.out.println(formatter.format(dataHora));
```

O pattern possui o dia, mês e ano (`dd/MM/uuuu`), que sempre serão impressos (caso o objeto sendo formatado não tenha algum destes

campos, será lançada uma exceção). Em seguida, os colchetes delimitam uma seção opcional: um espaço em branco seguido da hora, minuto e segundo (`[ HH:mm:ss]`) – não ignore os espaços, eles também fazem parte do pattern e são considerados na formatação.

Os campos dentro da seção opcional só são impressos caso o objeto sendo formatado os possua. Como `LocalDate` não possui campos de hora, ao formatá-lo a seção opcional é ignorada (se os campos não fossem opcionais, seria lançada uma exceção). Já `LocalDateTime` possui horário, e por isso a seção opcional é usada. A saída é:

```
04/05/2018
04/05/2018 17:30:00
```

17.5 Opções avançadas com DateTimeFormatterBuilder

Suponha que temos um `LocalTime` e precisamos transformá-lo em uma `String` . As classes do `java.time` possuem precisão de nanossegundos (9 casas decimais), mas um requisito do sistema diz que só podem ser mostrados no máximo 6 dígitos. Uma alternativa seria usar um pattern com somente 6 casas decimais na fração de segundos:

```
DateTimeFormatter formatter =
DateTimeFormatter.ofPattern("HH:mm:ss.SSSSSS");
LocalTime hora = LocalTime.of(10, 30, 15, 123400000);
System.out.println(formatter.format(hora));
```

Eu usei 6 letras `S` , portanto a saída terá apenas 6 dígitos na fração de segundos:

```
10:30:15.123400
```

> Vale lembrar que na API legada (`SimpleDateFormat`) a letra `S` corresponde aos milissegundos (e vimos os problemas que acontecem quando usamos mais que 3 dígitos). No `java.time` , esta mesma letra corresponde à fração de segundos e pode ter até 9 dígitos. Este é um caso em que as letras do pattern não funcionam da mesma maneira nas duas APIs.

Vamos supor que há outro requisito dizendo que a `String` não pode ter esses zeros no final (ou seja, `.123400` deveria ser mostrado como `.1234`). Como fazer isso?

Usando `DateTimeFormatter.ofPattern()` , não é possível. Felizmente, há uma classe própria para criarmos formatadores com regras mais complexas, que não podem ser feitas com um simples pattern. Esta classe se chama `java.time.format.DateTimeFormatterBuilder` .

Quantidade de dígitos variável

Para imprimir as frações de segundo até o último dígito que não seja zero, mas limitando ao máximo de 6 dígitos, basta usar um `DateTimeFormatterBuilder` .

Primeiro, criamos uma instância, e curiosamente esta é uma das poucas classes da API (senão a única) que possui um construtor público em vez de um *factory method*. Depois, basta ir usando os seus métodos para definir as regras do `DateTimeFormatter` .

Para as horas, minutos e segundos, usamos o método `appendPattern()` , que recebe uma `String` com um pattern. Como as horas, minutos e segundos terão tamanho fixo, é mais fácil usar o pattern diretamente. Depois, para as frações de segundo, usamos o método `appendFraction()` , que recebe os seguintes parâmetros:

- o `TemporalField` correspondente ao campo que será formatado (no caso, os nanossegundos);

- dois números que correspondem à quantidade mínima e máxima de dígitos a serem considerados;
- um `boolean` indicando se o ponto (.) deve estar na `String` formatada (no caso, usamos `true` para mostrá-lo).

Por fim, usamos o método `toFormatter()` , que retorna o `DateTimeFormatter` :

```
DateTimeFormatter formatter = new DateTimeFormatterBuilder()
    // hora:minuto:segundo
    .appendPattern("HH:mm:ss")
    // nanossegundos, com 6 casas decimais no máximo
    .appendFraction(ChronoField.NANO_OF_SECOND, 0, 6, true)
    // criar o DateTimeFormatter
    .toFormatter();
LocalTime hora = LocalTime.of(10, 30, 15, 123400000);
System.out.println(formatter.format(hora));
```

Com isso, a fração de segundo terá de zero a 6 dígitos (ignorando os zeros à direita) e a saída será:

```
10:30:15.1234
```

Texto customizado para um campo

A JVM possui várias `Strings` predefinidas para campos que são *locale sensitive*, como os nomes dos meses e dias da semana. Mas também é possível customizar o texto de qualquer campo, independente do locale. Para isso, usamos um `java.util.Map` que mapeia cada valor para seu respectivo texto.

Por exemplo: no Brasil, é comum referir-se ao dia 1 de cada mês como "Primeiro de [nome do mês]". Já para os demais dias, usa-se o valor numérico. Para formatar uma data desta maneira, primeiro temos que criar um `Map` que faz o mapeamento do valor para a respectiva `String` :

```
Map<Long, String> nomesDosDias = new HashMap<>();
// valor 1 (escrito como "1L", já que os valores devem ser long)
nomesDosDias.put(1L, "Primeiro");
```

Ou seja, o valor `1` será formatado para a `String` correspondente no `Map` , que no caso é `"Primeiro"` . Os demais dias não estão no `Map` , então serão formatados como o seu próprio valor numérico. A seguir, basta usar o método `appendText()` , que recebe um `TemporalField` correspondente ao campo que será formatado e o `Map` com os textos customizados.

Depois é usado um pattern com o nome do mês (`MMMM`) e, em seguida, chamamos o método `toFormatter(Locale)` , que cria um `DateTimeFormatter` com o locale indicado. O locale, neste caso, é para que o nome do mês esteja no idioma correto.

```
DateTimeFormatter formatter = new DateTimeFormatterBuilder()
    // nomes customizados para o dia do mês
    .appendText(ChronoField.DAY_OF_MONTH, nomesDosDias)
    // nome do mês
    .appendPattern(" 'de' MMMM")
    // criar o DateTimeFormatter com locale pt_BR
    .toFormatter(new Locale("pt", "BR"));
// formatar 4 de maio de 2018
System.out.println(LocalDate.of(2018, 5, 4).format(formatter));
// formatar 1 de maio de 2018
System.out.println(LocalDate.of(2018, 5, 1).format(formatter));
```

Com isso, o campo `ChronoField.DAY_OF_MONTH` será formatado com as `Strings` que estão no `Map` (ou com o valor numérico, caso não haja nenhum texto correspondente), enquanto o restante do pattern será formatado de acordo com o locale informado (no caso, `pt_BR`). A saída é:

```
4 de Maio
Primeiro de Maio
```

`DateTimeFormatterBuilder` possui várias outras opções interessantes, mas as melhores – na minha opinião – são aquelas usadas para parsing. É o que veremos no próximo capítulo.

Capítulo 18
Parsing com java.time

Para fazer parsing (transformar uma `String` em um dos tipos de data/hora) também usamos a classe `DateTimeFormatter` . A ideia básica é a mesma: fornecer um pattern e fazer o parsing da `String` , obtendo a data/hora.

A diferença é que na API legada só podíamos usar o método `parse()` da classe `SimpleDateFormat` , que retorna um `Date` . Mas agora temos vários tipos diferentes para representar datas e horas e é necessário indicar explicitamente qual deles queremos obter.

18.1 Devemos indicar o tipo a ser obtido

A maioria das classes de data e hora que já vimos possui um método estático `parse()` que recebe a `String` e o `DateTimeFormatter` que fará o parsing. O método sempre retorna o respectivo tipo, conforme mostra o próximo exemplo:

```
DateTimeFormatter parser = DateTimeFormatter.ofPattern("dd/MM/uuuu
HH:mm");
LocalDateTime dataHora = LocalDateTime.parse("04/05/2018 17:30", parser);
```

O pattern possui a data (`dd/MM/uuuu`) e a hora (`HH:mm`). O método `parse()` recebeu uma `String` neste formato e um `DateTimeFormatter` com este pattern, e o retorno é um `LocalDateTime` com os valores de data e hora devidamente setados para `2018-05-04T17:30` .

Se em vez disso quiséssemos obter um `LocalDate` , bastaria usar o método `LocalDate.parse()` . Mas atenção: mesmo que `LocalDate` não tenha informações sobre o horário, o pattern deve ter todos os campos (inclusive as horas e minutos) para que o parsing possa ser feito corretamente:

```
DateTimeFormatter parser = DateTimeFormatter.ofPattern("dd/MM/uuuu
HH:mm");
LocalDate data = LocalDate.parse("04/05/2018 17:30", parser);
```

Com isso, o `DateTimeFormatter` consegue obter todos os valores da `String` corretamente. Depois disso serão usados apenas os campos necessários para construir o `LocalDate` (dia, mês e ano) e o restante (hora e minuto) será descartado. Ou seja, mesmo que a classe resultante não use todos os campos, **o pattern deve corresponder exatamente à `String` de entrada**.

Caso a classe precise de algum campo que não está na entrada, será lançada uma exceção, como mostra o próximo exemplo:

```
DateTimeFormatter parser = DateTimeFormatter.ofPattern("dd/MM/uuuu
HH:mm");
OffsetDateTime odt = OffsetDateTime.parse("04/05/2018 17:30", parser);
```

`OffsetDateTime` precisa da data, hora e offset para ser construído, porém a `String` de entrada não possui offset. Por isso este código lança uma exceção:

```
java.time.format.DateTimeParseException: Text '04/05/2018 17:30' could not
be parsed: Unable to obtain OffsetDateTime from TemporalAccessor: {},ISO
resolved to 2018-05-04T17:30 of type java.time.format.Parsed
```

A mensagem diz que não conseguiu criar um `OffsetDateTime` , mas mostra que conseguiu obter uma data e hora (`resolved to 2018-05-04T17:30`). Só que `OffsetDateTime` precisa também do offset, por isso o erro. Na minha opinião, a mensagem poderia ser um pouco mais clara, dizendo qual a informação que está faltando (no caso, o offset). Nesse caso não foi difícil deduzir, mas nem sempre é tão óbvio o que está faltando.

Esta é uma diferença importante com relação à API legada. `SimpleDateFormat` sempre preenche as informações faltantes com valores predefinidos (timezone padrão da JVM, horário igual a meia-noite etc.) enquanto o `java.time` é mais restrito e não seta nenhum valor automaticamente. Se quisermos definir valores para campos

que não são informados, é possível configurá-los com `DateTimeFormatterBuilder`, conforme veremos posteriormente.

O retorno pode ser qualquer coisa

A classe `DateTimeFormatter` possui um método `parse()` que recebe a `String` com a data/hora e retorna um `TemporalAccessor` que encapsula os valores obtidos pelo parsing. A partir deste `TemporalAccessor` é possível usar os métodos `get()` e `getLong()` para obter tais valores. Exemplo:

```
DateTimeFormatter parser = DateTimeFormatter.ofPattern("dd/MM/uuuu
HH:mm");
TemporalAccessor parsed = parser.parse("04/05/2018 17:30");
System.out.println(parsed.get(ChronoField.DAY_OF_MONTH)); // 4
```

Neste exemplo foi usado o campo `ChronoField.DAY_OF_MONTH` para obter o valor do dia (no caso, `4`). Este código poderia ser usado para casos em que só queremos saber o valor numérico de algum campo, por exemplo, sem ter que criar um objeto específico (como `LocalDate` ou `LocalDateTime`). Mas este não é o uso mais interessante deste método.

Como já vimos em capítulos anteriores, um `TemporalAccessor` pode ser passado para uma instância de `TemporalQuery`, que por sua vez pode retornar objetos de qualquer tipo. O interessante é que o método `parse()` pode receber um `TemporalQuery` como parâmetro, e o resultado será o mesmo valor retornado pelo `TemporalQuery`.

Por exemplo, se usarmos o `TemporalQuery` que vimos nos capítulos anteriores, que verifica se uma data é um fim de semana:

```
TemporalQuery<Boolean> isFimDeSemana = temporal -> {
    DayOfWeek diaDaSemana = DayOfWeek.from(temporal);
    return diaDaSemana == DayOfWeek.SATURDAY || diaDaSemana ==
DayOfWeek.SUNDAY;
};
```

Podemos passar este `TemporalQuery` diretamente para o método `parse()`:

```
DateTimeFormatter parser = DateTimeFormatter.ofPattern("dd/MM/uuuu
HH:mm");
Boolean fimDeSemana = parser.parse("04/05/2018 17:30", isFimDeSemana); //
false
```

O método `parse()` faz o parsing de `"04/05/2018 17:30"`, resultando em um `TemporalAccessor` com todos os valores de data e hora obtidos desta `String`. Esse resultado por sua vez é passado para o `TemporalQuery`, que executa a lógica de verificar se a data é um fim de semana. O valor retornado pelo método `parse()` é o mesmo retornado pelo `TemporalQuery`. Como a data informada (4 de maio de 2018) é uma sexta-feira, o resultado é `false`.

Verifique qual o melhor tipo a ser retornado em cada situação. Na API legada não tínhamos escolha, pois `SimpleDateFormat.parse()` sempre retorna um `Date`, mas agora podemos escolher o tipo mais adequado para cada caso, seja usando o método `parse()` que várias classes possuem, seja usando um `TemporalQuery`.

Um caso de uso comum é com um *method reference* do método `from()` das classes nativas da API, já que a assinatura deste é compatível com `TemporalQuery`. `DayOfWeek`, por exemplo, não possui o método `parse()`, mas é possível obter uma instância desta classe usando seu método `from()`:

```
DateTimeFormatter parser = DateTimeFormatter.ofPattern("dd/MM/uuuu");
DayOfWeek dow = parser.parse("04/05/2018", DayOfWeek::from); // FRIDAY
```

Com isso, não precisamos criar um `LocalDate` para só depois obter o `DayOfWeek`, podemos obtê-lo diretamente (assumindo que você não precisará do `LocalDate`, claro).

18.2 Pattern opcional e valores predefinidos

Assim como os patterns opcionais podem ser usados para formatação, também podem ser usados para parsing. Mas como o padrão do `java.time` é não usar valores predefinidos para os campos faltantes (ao contrário de `SimpleDateFormat`), precisamos definir quais são esses valores. Para isso, usamos um `DateTimeFormatterBuilder` .

No próximo exemplo, eu uso um pattern com a data obrigatória e hora opcional. Em seguida, defino qual valor será usado para o horário quando este não estiver presente, usando o método `parseDefaulting()` . Por fim, também uso este método para definir o valor do offset:

```
DateTimeFormatter parser = new DateTimeFormatterBuilder()
    // data obrigatória, hora opcional
    .appendPattern("dd/MM/uuuu[ HH:mm]")
    // valor predefinido para hora
    .parseDefaulting(ChronoField.HOUR_OF_DAY, 10)
    // valor predefinido para minuto
    .parseDefaulting(ChronoField.MINUTE_OF_HOUR, 30)
    // valor predefinido para offset (-03:00)
    .parseDefaulting(ChronoField.OFFSET_SECONDS,
                     ZoneOffset.ofHours(-3).getTotalSeconds())
    // criar DateTimeFormatter
    .toFormatter();
// String sem horário e offset, usa os valores predefinidos
// 2018-05-04T10:30-03:00
OffsetDateTime odt = OffsetDateTime.parse("04/05/2018", parser);
// String com horário, ignora o valor predefinido
// 2018-05-04T17:20-03:00
odt = OffsetDateTime.parse("04/05/2018 17:20", parser);
```

O pattern possui os campos hora e minuto como opcionais: quando eles não forem fornecidos, serão usados os valores definidos pelas chamadas do método `parseDefaulting()` . No caso, a hora terá o valor `10` e os minutos, o valor `30` . Se quiséssemos, também poderíamos ter definido valores para os segundos e nanossegundos, mas neste caso eles serão automaticamente setados para zero. Um detalhe importante é que, ao definir valores para o horário, **pelo menos as**

horas devem ter algum valor definido, caso contrário será lançada uma exceção durante o parsing.

Para o offset, eu usei o campo `ChronoField.OFFSET_SECONDS` , que corresponde ao valor do offset em segundos. Para obter este valor, basta usar o método `getTotalSeconds()` da classe `ZoneOffset` . No caso, usamos `ZoneOffset.ofHours(-3)` , que retorna o offset `-03:00` .

No primeiro caso, a `String` só possui a data, por isso o horário foi setado de acordo com os valores definidos por `parseDefaulting()` e o resultado é `2018-05-04T10:30-03:00` . No segundo caso, a `String` possui data e hora, por isso os valores definidos para estes campos são ignorados e o resultado é `2018-05-04T17:20-03:00` . Note que em ambos os casos o offset é `-03:00` : como este campo não está no pattern (e por isso não está na `String` de entrada), sempre será usado o valor predefinido.

O fato de termos que definir os valores dos campos faltantes pode parecer uma desvantagem com relação a `SimpleDateFormat` , já que esta usa valores predefinidos para todos os campos. De fato, o `java.time` é um pouco mais trabalhoso, mas em compensação podemos usar os valores que quisermos. Além disso, o fato de deixar os valores explícitos no código faz com que fique mais claro o que está acontecendo. Este comportamento é o oposto do que `SimpleDateFormat` faz e foi uma decisão de *design* da API, conforme o próprio Stephen Colebourne comenta nesta lista de discussão (https://sourceforge.net/p/threeten/mailman/message/32911367/).

Um detalhe importante é que, devido à forma como foi implementado, a documentação recomenda que o método `parseDefaulting()` seja chamado somente no final da definição do *builder* (https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatterBuilder.html#parseDefaulting-java.time.temporal.TemporalField-long-/). Portanto, devem ser os últimos métodos a serem chamados, imediatamente antes de `toFormatter()` .

Cuidados ao usar patterns opcionais

O uso de patterns opcionais torna possível o parsing de vários formatos diferentes com um único `DateTimeFormatter` , como mostra o próximo exemplo:

```
// 2 formatos possíveis para data
DateTimeFormatter parser = DateTimeFormatter.ofPattern("[dd/MM/uuuu][uuuu-MM-dd]");
// ambas as linhas abaixo resultam no mesmo LocalDate
LocalDate data1 = LocalDate.parse("2018-05-04", parser);
LocalDate data2 = LocalDate.parse("04/05/2018", parser);
```

Neste exemplo, o `DateTimeFormatter` aceita tanto `Strings` no formato `dd/MM/uuuu` quanto no formato `uuuu-MM-dd` , por isso as duas chamadas ao método `parse()` funcionam.

Porém, esse `DateTimeFormatter` não é bom para formatar, pois para cada seção opcional é verificado se o objeto contém todos os campos da seção. Como ambas usam os campos dia, mês e ano (exatamente o que `LocalDate` possui), as duas seções opcionais são usadas e a data é impressa nos dois formatos. Ou seja, se fizermos `data.format(parser)` o resultado será `04/05/20182018-05-04` . Para evitar esse problema, o ideal é usar outro `DateTimeFormatter` para formatar a data.

Outro ponto de atenção é que nem sempre patterns opcionais adjacentes com os mesmos campos funcionam. Por exemplo, para termos um `DateTimeFormatter` que aceita offsets em 3 formatos diferentes (`-03` , `-0300` ou `-03:00`), poderíamos tentar o seguinte:

```
// data e hora, com 3 formatos possíveis para offset
DateTimeFormatter parser = DateTimeFormatter.ofPattern("dd/MM/uuuu HH:mm[X][XX][XXX]");
OffsetDateTime odt = OffsetDateTime.parse("04/05/2018 17:30-03:00", parser);
```

Só que este código lança uma exceção:

```
java.time.format.DateTimeParseException: Text '04/05/2018 17:30-03:00'
could not be parsed, unparsed text found at index 19
```

Provavelmente devido a algum detalhe de implementação, este problema é facilmente resolvido mudando-se a ordem das seções opcionais, colocando as maiores (com mais caracteres) primeiro:

```
// data e hora, com 3 formatos possíveis para offset
DateTimeFormatter parser = DateTimeFormatter.ofPattern("dd/MM/uuuu
HH:mm[XXX][XX][X]");
OffsetDateTime odt = OffsetDateTime.parse("04/05/2018 17:30-03:00",
parser);
```

Com isso, todos os 3 formatos de offset são aceitos e o parsing é feito corretamente. Vale lembrar que este `DateTimeFormatter` não é bom para formatar, pois imprimirá o offset 3 vezes (uma em cada formato).

Enfim, patterns opcionais são uma excelente funcionalidade da API, mas use com moderação. Para casos em que há muitos formatos possíveis, é melhor criar várias instâncias de `DateTimeFormatter` e ir tentando uma a uma, em vez de tentar um único pattern com várias seções opcionais.

18.3 Locales, maiúsculas e textos customizados

Nos EUA é comum escrever datas como `May 4th, 2018` (nome do mês, dia com sufixo – "st", "nd", "rd" ou "th" – e ano). Como fazer parsing de uma `String` neste formato? Para este exemplo, só para complicar um pouco mais, vamos supor também que o nome do mês pode estar escrito em minúsculas, ou seja, o parsing deve aceitar tanto `May` como `may` .

Com um `DateTimeFormatterBuilder` , tal tarefa é perfeitamente possível. Para fazer o parsing do nome do mês, basta usar o locale correspondente ao idioma inglês (`Locale.ENGLISH`). Também devemos

usar o método `parseCaseInsensitive()` para ignorar a diferença entre letras maiúsculas e minúsculas (por padrão, o parsing é *case sensitive*, e no caso do mês somente `May` seria aceito).

Para o dia com sufixo, podemos usar um `Map` com textos customizados, como fizemos no capítulo anterior. No caso, o texto será o valor numérico mais o sufixo correspondente: `1st` , `2nd` etc.

```
Map<Long, String> diasComSufixo = new HashMap<>();
// preencher o map com todos os valores válidos para o dia
for (int dia = 1; dia <= 31; dia++) {
    String texto = Integer.toString(dia);
    switch (dia) {
        case 1:
        case 21:
        case 31:
            texto += "st";
            break;
        case 2:
        case 22:
            texto += "nd";
            break;
        case 3:
        case 23:
            texto += "rd";
            break;
        default:
            texto += "th";
    }
    diasComSufixo.put((long) dia, texto);
}
```

Em seguida, basta criar o `DateTimeFormatter` :

```
DateTimeFormatter parser = new DateTimeFormatterBuilder()
    // ignorar maiúsculas e minúsculas
    .parseCaseInsensitive()
    // nome do mês seguido de espaço
    .appendPattern("MMMM ")
    // dia com sufixo
    .appendText(ChronoField.DAY_OF_MONTH, diasComSufixo)
```

```
    // vírgula, espaço, ano
    .appendPattern(", uuuu")
    // locale do idioma inglês (para o nome do mês)
    .toFormatter(Locale.ENGLISH);
// 2018-05-04
LocalDate data = LocalDate.parse("may 4th, 2018", parser);
```

Com isso, o parsing da data é feito corretamente e o resultado é `2018-05-04`.

Se usarmos este mesmo `DateTimeFormatter` para formatar a data (ou seja, se fizermos `data.format(parser)`) o resultado será `May 4th, 2018` – repare que o nome do mês começa com letra maiúscula, pois o modo *case insensitive* só afeta o parsing. Para formatação, são usadas as `Strings` correspondentes ao locale, que são predefinidas na JVM (no caso do nome do mês em inglês, o resultado é `May`).

Um detalhe importante é que o método `parseCaseInsensitive()` deve ser chamado antes de `appendPattern("MMMM ")`, caso contrário não funcionará. Somente os campos que aparecem depois da chamada de `parseCaseInsensitive()` ignorarão a diferença entre maiúsculas e minúsculas.

E para fazer com que os campos voltem a considerar esta diferença, basta chamar o método `parseCaseSensitive()`. Isso é interessante porque permite que o *parser* seja bem flexível, com diferentes campos tendo comportamentos distintos quanto a ignorar ou não a diferença entre maiúsculas e minúsculas.

```
DateTimeFormatter parser = new DateTimeFormatterBuilder()
    .parseCaseInsensitive()
    // ... campos que ignoram maiúsculas e minúsculas
    .parseCaseSensitive()
    // ... campos que não ignoram maiúsculas e minúsculas
    .parseCaseInsensitive()
    // ... mais campos que ignoram maiúsculas e minúsculas
    .parseCaseSensitive()
    // ... mais campos que não ignoram maiúsculas e minúsculas
    .toFormatter();
```

18.4 Validação e modos de parsing

A API legada é leniente por padrão, então `SimpleDateFormat` aceita `Strings` como `33/01/2018` (33 de janeiro de 2018) e faz todos os ajustes possíveis para obter uma data válida. Por outro lado, o comportamento padrão de `DateTimeFormatter` é não ser leniente. Além disso, essa classe possui vários modos diferentes de parsing, conforme veremos a seguir.

Por padrão, não é leniente

Quando criamos um `DateTimeFormatter` , por padrão ele não é leniente, então datas como 33 de janeiro não serão aceitas no parsing, como mostra o próximo exemplo:

```
DateTimeFormatter parser = DateTimeFormatter.ofPattern("dd/MM/uuuu");
LocalDate data = LocalDate.parse("33/01/2018", parser);
```

Esse código lança uma exceção:

```
java.time.format.DateTimeParseException: Text '33/01/2018' could not be
parsed: Invalid value for DayOfMonth (valid values 1 - 28/31): 33
```

A mensagem diz que o valor `33` é inválido para o campo `DayOfMonth` (dia do mês).

Para que o `DateTimeFormatter` seja leniente, temos que usar um `java.time.format.ResolverStyle` : um `enum` que define o modo como os valores dos campos são "resolvidos" (ou ajustados). Para isso, basta usar o método `withResolverStyle()` . Como as classes do `java.time` são imutáveis, este método retorna **outro** `DateTimeFormatter` (seguindo a mesma lógica das demais classes, na qual os métodos `withXXX` retornam outra instância com algum valor modificado).

```
DateTimeFormatter parser = DateTimeFormatter.ofPattern("dd/MM/uuuu")
    // mudar para leniente
    .withResolverStyle(ResolverStyle.LENIENT);
LocalDate data = LocalDate.parse("33/01/2018", parser); // 2018-02-02
```

No modo leniente, o dia 33 é ajustado: como janeiro tem 31 dias, "33" seriam 2 dias depois de 31 de janeiro. Por isso, a data é ajustada para 2 de fevereiro.

Por padrão, também não é "não leniente"

`SimpleDateFormat` só possui 2 modos de parsing: leniente e não leniente. Tanto que o método para mudar esta configuração (`setLenient()`) recebe um parâmetro booleano, indicando se é leniente ou não.

`ResolverStyle` , por sua vez, possui 3 valores diferentes: `LENIENT` , `STRICT` e `SMART` . No exemplo anterior usamos `LENIENT` , que faz com que o `DateTimeFormatter` aceite valores acima dos limites de um campo (como o dia 33) e faça os devidos ajustes. Os outros dois modos veremos a seguir.

Mas antes, vamos ver como a API se comporta ao fazer o parsing de 31 de abril:

```
DateTimeFormatter parser = DateTimeFormatter.ofPattern("dd/MM/uuuu");
LocalDate data = LocalDate.parse("31/04/2018", parser); // 2018-04-30
```

O resultado é 30 de abril de 2018 (`2018-04-30`).

Isso acontece porque, quando criamos um `DateTimeFormatter` sem indicar um `ResolverStyle` , por padrão é usado o modo `SMART` , cujo comportamento é "leniente mas nem tanto". Ou seja, ele só aceita valores dentro dos limites permitidos pelo campo (dia entre 1 e 31, mês entre 1 e 12 etc.) mas faz alguns ajustes quando julga necessário (no caso, o dia 31 não existe em abril, então é ajustado para o último dia do mês).

Para que este ajuste não seja feito e o parsing aceite somente datas válidas, basta mudar o `ResolverStyle` para `STRICT` , que é o modo mais rigoroso de todos:

```
DateTimeFormatter parser = DateTimeFormatter.ofPattern("dd/MM/uuuu")
    .withResolverStyle(ResolverStyle.STRICT);
```

```
LocalDate data = LocalDate.parse("31/04/2018", parser);
```

Com isso, 31 de abril não é mais aceito, pois abril só tem 30 dias, e o código lança uma exceção:

```
java.time.format.DateTimeParseException: Text '31/04/2018' could not be
parsed: Invalid date 'APRIL 31'
```

Resumindo, os 3 valores de `ResolverStyle` são:

- `STRICT` : só aceita datas válidas. Por exemplo, datas como 31 de abril ou 29 de fevereiro em anos que não são bissextos lançam uma exceção.
- `LENIENT` : aceita qualquer valor, inclusive fora dos limites de um campo, como dia 33, e faz os ajustes necessários. Por exemplo, 31 de abril é ajustado para 1 de maio, 33 de janeiro vira 2 de fevereiro etc.
- `SMART` : é o padrão quando nenhum `ResolverStyle` é especificado, e funciona como um modo intermediário entre `STRICT` e `LENIENT` . Não aceita valores fora dos limites de um campo, mas faz ajustes quando necessário. Por exemplo, 31 de abril ou 29 de fevereiro em anos não bissextos são aceitos e ajustados para o último dia do mês, mas dia 33 lança exceção.

É leniente, mas não é bagunça

Anteriormente vimos que `SimpleDateFormat` é tão leniente que, quando o pattern é `yyyyMMdd` e é feito o parsing de `2018-02-01` , o resultado é 2 de dezembro de 2017. Com `DateTimeFormatter` isso não acontece pois, mesmo em modo leniente, ele espera que a `String` corresponda ao pattern. Exemplo:

```
DateTimeFormatter parser = DateTimeFormatter.ofPattern("uuuuMMdd")
    .withResolverStyle(ResolverStyle.LENIENT);
System.out.println(parser.parse("2018-02-01"));
```

Mesmo estando em modo leniente, `DateTimeFormatter` percebe que a `String` está em um formato diferente do pattern, por isso esse código lança uma exceção:

```
java.time.format.DateTimeParseException: Text '2018-02-01' could not be
parsed at index 4
```

Usar "u" ou "y" para o ano?

No capítulo anterior vimos os patterns existentes para representar o ano: `u` e `y` . Um caso em que faz diferença usar um ou outro é quando temos um parser em modo `STRICT` . Vamos criar um `DateTimeFormatter` com dia, mês e ano e ver o que acontece:

```
DateTimeFormatter parser = DateTimeFormatter.ofPattern("dd/MM/yyyy")
    .withResolverStyle(ResolverStyle.STRICT);
System.out.println(LocalDate.parse("10/02/2018", parser));
```

Este código lança uma exceção:

```
java.time.format.DateTimeParseException: Text '10/02/2018' could not be
parsed: Unable to obtain LocalDate from TemporalAccessor: {DayOfMonth=10,
YearOfEra=2018, MonthOfYear=2},ISO of type java.time.format.Parsed
```

A mensagem da exceção informa que o ano foi corretamente setado (`YearOfEra=2018`), porém o modo `STRICT` precisa saber também qual é a era (AC ou DC), conforme explicado pelo próprio Stephen Colebourne nesta resposta do Stack Overflow (https://stackoverflow.com/a/41104034/). Como a era não foi informada, o parser não tem como saber se o ano é 2018 AC ou 2018 DC e não consegue criar a data.

Uma maneira de resolver é mudar o `ResolverStyle` para `SMART` ou `LENIENT` , o que faz com que a era atual (DC) seja usada para criar a data (e é por isso que `y` funciona nesses casos). Mas, como já vimos, esses dois modos também fazem ajustes automáticos em datas inválidas. Se eu não quero que estes ajustes sejam feitos (ou seja, eu preciso do modo `STRICT`), basta mudar o pattern para `dd/MM/uuuu` .

Outra alternativa é usar um `DateTimeFormatterBuilder` e definir um valor para a era com `parseDefaulting(ChronoField.ERA, valor)` , usando o valor 1 para DC e 0 para AC.

Ignorando o ResolverStyle

Os 3 modos de parsing definidos por `ResolverStyle` possuem um ponto em comum: eles sempre tentam "resolver" a data. Ou seja, com base nos valores obtidos, estes podem ser ajustados (ou não, dependendo do `ResolverStyle` escolhido) para que no final tenhamos alguma data válida. Mas e se quisermos implementar nossa própria lógica, sem depender destes ajustes predefinidos pela API?

Por exemplo, se tivermos uma `String` como `"99/00/2018"` (que seria o "dia 99 do mês zero"). Nos modos `STRICT` e `SMART` ela lançará uma exceção, pois os valores `99` e `00` estão fora dos limites aceitos pelos respectivos campos.

Já no modo `LENIENT` ela será ajustada: o mês zero é ajustado para dezembro do ano anterior (2017), que tem 31 dias. E 99 excede 31 em 68 dias, portanto a data é ajustada para 68 dias depois de 31 de dezembro de 2017, resultando em 9 de março de 2018 (se esse algoritmo faz sentido ou não, é outra história, o fato é que o modo leniente faz isso).

Mas vamos supor que um requisito do sistema (que por algum motivo aceita tais `Strings`) diz que o mês zero deve ser automaticamente mudado para janeiro, e qualquer dia maior do que o máximo permitido no mês deve ser ajustado para o último dia do mês. Ou seja, 31 de abril vira 30 de abril, 99 de março vira 31 de março etc.

Usando o método `parse()` não temos como obter os valores `99` e `00` , pois nenhum `ResolverStyle` permite isso: ou uma exceção é lançada, ou os valores são automaticamente ajustados.

Felizmente, há uma alternativa: o método `parseUnresolved()` , que, como o próprio nome diz, não "resolve" a data (não faz ajustes nos campos, nem valida seus valores). Este método recebe a `String` com a data e um `java.text.ParsePosition` , que indica a posição da `String` onde o parsing começará.

Após o parsing, o `ParsePosition` é atualizado com a posição da `String` onde deu erro (ou `-1` caso nenhum tenha ocorrido). Também é atualizada a posição final do parsing, e podemos compará-la com o tamanho da entrada, para saber se toda a `String` foi processada:

```
String input = "99/00/2018";
DateTimeFormatter parser = DateTimeFormatter.ofPattern("dd/MM/uuuu");
// começar o parsing da posição zero (ou seja, desde o início da String)
ParsePosition position = new ParsePosition(0);
TemporalAccessor parsed = parser.parseUnresolved(input, position);
// verifica se houve erro no parsing
if (position.getErrorIndex() >= 0) {
    System.out.println("Erro na posição " + position.getErrorIndex());
} else if (position.getIndex() < input.length()) {
    System.out.println("Não fez parsing da String toda, parou na posição "
                       + position.getIndex());
} else {
    // parsing feito com sucesso
    System.out.println(parsed);
}
```

Nesse caso, o parsing é feito com sucesso e a variável `parsed` é impressa:

```
{Year=2018, DayOfMonth=99, MonthOfYear=0},null
```

Repare que o parsing obteve os valores de cada campo (ano 2018, mês zero e dia 99). Como a variável `parsed` é um `TemporalAccessor` , podemos usar o método `getLong()` para obter os valores dos campos e aplicar nossa lógica customizada para criar a data.

Na verdade, o método `parseUnresolved()` retorna uma instância de `java.time.format.Parsed` (uma classe não pública que implementa `TemporalAcessor`). Nesta classe, o método `getLong()` retorna o valor de um campo sem validá-lo, que é exatamente o que precisamos. Já o método `get()` verifica o valor antes de retorná-lo, por isso não é possível usá-lo para este caso, já que o dia e mês possuem valores inválidos e uma exceção seria lançada.

No código a seguir é usada a classe `java.time.YearMonth` , que representa um ano e um mês, sem nenhum valor para o dia. Esta classe possui alguns métodos úteis para este caso, que serão explicados em seguida:

```
// supondo que o parsing foi feito com sucesso
TemporalAccessor parsed = parser.parseUnresolved(input, position);
// obter o ano
int ano = (int) parsed.getLong(ChronoField.YEAR);
// obter o mês
int mes = (int) parsed.getLong(ChronoField.MONTH_OF_YEAR);
// se mês for zero, mudar para janeiro
if (mes == 0) {
    mes = 1;
}
// YearMonth representa o mês e o ano
YearMonth ym = YearMonth.of(ano, mes);
LocalDate data;
// obter o dia
int dia = (int) parsed.getLong(ChronoField.DAY_OF_MONTH);
// verifica se o dia ultrapassa o máximo permitido no mês
if (dia > ym.lengthOfMonth()) {
    // dia maior que o permitido, ajustar para o último dia do mês
    data = ym.atEndOfMonth();
} else {
    // valor do dia OK, usá-lo para construir o LocalDate
    data = ym.atDay(dia);
}
```

O uso da classe `YearMonth` facilita este algoritmo, pois ela possui o método `lengthOfMonth()` , que retorna o número de dias naquele mês e ano (ambos são necessários para verificar os casos de anos bissextos). Além disso, há também os métodos `atEndOfMonth()` (que retorna um `LocalDate` correspondente ao último dia daquele mês e ano) e `atDay()` (que retorna um `LocalDate` correspondente ao dia informado, com os mesmos valores do mês e ano do `YearMonth`).

Como o valor do mês é zero, este é ajustado para janeiro. E como o dia é 99 e este valor ultrapassa o `lengthOfMonth()` de janeiro (que é `31`

), o dia é ajustado para o final do mês. Assim, a data resultante é `2018-01-31`.

Vale lembrar que este exemplo não cobre casos em que o mês é maior que 12 ou o dia é menor que 1. Mas estes casos não são difíceis de serem adicionados e ficam como exercício para o leitor.

18.5 Parsing pode retornar mais de um tipo diferente

Dado este `DateTimeFormatter`:

```
// data obrigatória, hora e offset opcionais
DateTimeFormatter parser = DateTimeFormatter.ofPattern("dd/MM/uuuu[ HH:mm][XXX]");
```

Conforme já visto anteriormente, se tivermos uma `String` com somente a data e quisermos fazer o parsing dela para obter um `OffsetDateTime`, teremos que configurar valores predefinidos para as horas e o offset, usando um `DateTimeFormatterBuilder` e o método `parseDefaulting()`.

Mas vamos supor que eu **não quero** necessariamente um `OffsetDateTime`, e sim o tipo mais específico que eu puder, dependendo da entrada: se a `String` só contém a data, quero que o resultado seja um `LocalDate`, se tiver data e hora, o resultado deve ser um `LocalDateTime`, e somente se tiver data, hora e offset, o resultado deve ser um `OffsetDateTime`.

Nesse caso, a solução é usar o método `parseBest()`, que recebe uma lista de vários `TemporalQuery` e tenta usá-los com o resultado do parsing. Exemplo:

```
// data obrigatória, hora e offset opcionais
DateTimeFormatter parser = DateTimeFormatter.ofPattern("dd/MM/uuuu[ HH:mm][XXX]");
```

```
TemporalAccessor parsed = parser.parseBest("04/05/2018",
    // lista de TemporalQuery a serem usados no parsing
    OffsetDateTime::from, LocalDateTime::from, LocalDate::from);
// verifica se o resultado é OffsetDateTime
if (parsed instanceof OffsetDateTime) {
    OffsetDateTime odt = (OffsetDateTime) parsed;
    // fazer o que quiser com o OffsetDateTime
}
// verifica se o resultado é LocalDateTime
if (parsed instanceof LocalDateTime) {
    LocalDateTime ldt = (LocalDateTime) parsed;
    // fazer o que quiser com o LocalDateTime
}
// verifica se o resultado é LocalDate
if (parsed instanceof LocalDate) {
    LocalDate data = (LocalDate) parsed;
    // fazer o que quiser com o LocalDate
}
```

Foram usados os *method references* dos métodos `from()` presentes nas classes `OffsetDateTime` , `LocalDateTime` e `LocalDate` , mas você poderia passar qualquer instância de `TemporalQuery` . A única restrição é que, para serem usadas com `parseBest()` , o retorno do `TemporalQuery` deve ser uma classe que implemente `TemporalAccessor` .

Primeiro o método `parseBest()` faz o parsing da `String` , que no caso é `04/05/2018` (ou seja, somente a data). Depois o resultado do parsing é passado para `OffsetDateTime.from()` , já que este é o primeiro da lista. Como a `String` não possui todos os campos necessários para criar um `OffsetDateTime` (data, hora e offset), o parsing prossegue para o próximo da lista: `LocalDateTime.from()` .

`LocalDateTime` precisa de data e hora para ser criado, mas a `String` só possui data, então também não é possível criar uma instância desta classe. Com isso, o parsing prossegue para o próximo da lista: `LocalDate.from()` . Neste caso, o `LocalDate` pode ser criado, pois ele só precisa da data (dia, mês e ano). Com isso o método `parseBest()` retorna uma instância de `LocalDate` e o código entra no terceiro `if` .

E se a `String` tiver data e hora?

```
TemporalAccessor parsed = parser.parseBest("04/05/2018 17:30",
    OffsetDateTime::from, LocalDateTime::from, LocalDate::from);
```

Nesse caso, `OffsetDateTime.from()` ainda não conseguirá criar um `OffsetDateTime` , pois precisa do offset. Mas o `LocalDateTime` conseguirá ser criado e o código entra no segundo `if` . O método `parseBest()` é útil para estes casos em que mais de um tipo pode ser retornado, e dependendo do retorno, uma ação diferente pode ser tomada.

18.6 Parsing de ISO 8601

Por padrão, as classes de data e hora podem fazer o parsing de `Strings` no formato ISO 8601, sem precisar de um `DateTimeFormatter` . Exemplos:

```
// somente data
LocalDate data = LocalDate.parse("2018-05-04");
// data e hora (com segundos e nanossegundos)
LocalDateTime dataHora = LocalDateTime.parse("2018-05-
04T17:30:45.123456789");
// data, hora e offset
OffsetDateTime odt = OffsetDateTime.parse("2018-05-04T17:30:45-03:00");
// data, hora, offset e timezone
ZonedDateTime zdt = ZonedDateTime.parse("2018-05-04T17:30-
03:00[America/Sao_Paulo]");
```

A "exceção" é `ZonedDateTime` , pois como a norma ISO 8601 não aceita identificadores da IANA, essa classe estendeu o formato adicionando o nome do timezone entre colchetes. Mas os demais campos (data, hora e offset) estão de acordo com a ISO 8601.

Internamente, cada classe usa uma das constantes predefinidas na classe `DateTimeFormatter` . `LocalDate` , por exemplo, usa `DateTimeFormatter.ISO_LOCAL_DATE` , ou seja, o seu método `parse()` só

aceitará `Strings` no formato `uuuu-MM-dd` . Veja a documentação de cada classe para saber qual formato cada uma aceita.

Caso seja preciso fazer parsing de um formato diferente, crie um `DateTimeFormatter` específico e passe como parâmetro no método `parse()` , conforme já vimos no início do capítulo. Um caso comum é aceitar offsets com dois pontos (`-03:00`), sem os dois pontos (`-0300`) ou somente com as horas (`-03`).

Nesse caso, podemos usar um `DateTimeFormatterBuilder` . Primeiro adicionamos a data e hora, e em seguida adicionamos os offsets em todos os formatos possíveis, cada um em uma seção opcional:

```
DateTimeFormatter parser = new DateTimeFormatterBuilder()
    // data e hora no formato ISO 8601
    .append(DateTimeFormatter.ISO_LOCAL_DATE_TIME)
    // offset: -03:00, -0300 ou -03
    .appendPattern("[XXX][XX][X]")
    // criar o DateTimeFormatter
    .toFormatter();
System.out.println(OffsetDateTime.parse("2018-05-04T10:00-03", parser));
```

Note que usei a constante `ISO_LOCAL_DATE_TIME` . Eu poderia até ter usado `uuuu-MM-dd'T'HH:mm:ss.SSS` , por exemplo, mas este pattern só faz parsing se os segundos e frações de segundo estiverem presentes. Além disso, usando este pattern, as frações de segundo devem ter exatamente 3 casas decimais (`SSS`).

A constante `ISO_LOCAL_DATE_TIME` é mais flexível e permite que os segundos e nanossegundos sejam setados para zero caso não estejam presentes, além de aceitar uma quantidade variável de casas decimais para os nanossegundos (de zero a 9). Aliás, todas as constantes de `DateTimeFormatter` (cujos nomes começam com `ISO_`) possuem essa flexibilidade.

18.7 Parsing de abreviação de timezone

Como já vimos várias vezes ao longo do livro, as abreviações de timezones (como "BRT", "IST", "EST" etc.) não são consideradas timezones de fato. Além disso, muitas são ambíguas, como "IST", que é usada na Índia, Israel e Irlanda. Por isso, ao fazer parsing de uma abreviação, devemos nos atentar para este fato.

O problema é que `DateTimeFormatter` usa algum timezone arbitrário ao fazer parsing de uma abreviação e nem sempre será o que você precisa. Por exemplo:

```
DateTimeFormatter parser = DateTimeFormatter.ofPattern("EEE MMM dd
HH:mm:ss z uuuu", Locale.ENGLISH);
System.out.println(ZonedDateTime.parse("Sun Jan 07 10:00:00 IST 2018",
parser));
```

A `String` de entrada possui a abreviação "IST", mas qual timezone foi usado, o de Israel, Irlanda ou Índia? A saída é `2018-01-07T10:00+02:00[Asia/Jerusalem]` , ou seja, "IST" resultou em uma data no timezone de Israel.

Se quisermos que a abreviação corresponda a outro timezone, basta configurar um conjunto de timezones a serem usados em caso de ambiguidade, usando um `java.util.Set` com os `ZoneIds` e passando-o para um `DateTimeFormatterBuilder` . No exemplo a seguir, eu configuro o `Set` para usar um timezone da Índia (um dos que usam a abreviação "IST") e outro da China, cuja abreviação é "CST" (que também é ambígua, pois é usada em Cuba e na região central dos EUA).

```
// timezones a serem usados em caso de ambiguidade
Set<ZoneId> zones = new HashSet<>();
zones.add(ZoneId.of("Asia/Kolkata")); // Índia
zones.add(ZoneId.of("Asia/Shanghai")); // China
DateTimeFormatter parser = new DateTimeFormatterBuilder()
    .appendPattern("EEE MMM dd HH:mm:ss ")
    // usar java.time.format.TextStyle para abreviação do timezone
    .appendZoneText(TextStyle.SHORT, zones) // usar o Set de timezones
    .appendPattern(" uuuu").toFormatter(Locale.ENGLISH);
System.out.println(ZonedDateTime.parse("Sun Jan 07 10:00:00 IST 2018",
```

```
parser));
System.out.println(ZonedDateTime.parse("Sun Jan 07 10:00:00 CST 2018",
parser));
```

Sempre que alguma abreviação é encontrada, os timezones que estão no `Set` têm "preferência" sobre os demais que usam a mesma abreviação. Com isso, o resultado será:

```
2018-01-07T10:00+05:30[Asia/Kolkata]
2018-01-07T10:00+08:00[Asia/Shanghai]
```

Se eu simplesmente usar o pattern `EEE MMM dd HH:mm:ss zzz uuuu`, os resultados terão `Asia/Jerusalem` e `America/Chicago`. Para não depender do comportamento arbitrário da JVM e usar os timezones que você precisa, use o `DateTimeFormatterBuilder` com o `Set` de timezones.

Com isso, encerramos os principais casos de parsing. No próximo capítulo veremos como usar a API para testes que dependem da data e hora atual, além de outros casos de uso.

Capítulo 19
Testes e outros casos de uso

Várias classes do `java.time` possuem o método `now()` , que retorna a data/hora atual. Este método obtém o valor do timestamp correspondente ao instante atual e usa algum timezone para convertê-lo para uma data, hora e offset específicos.

Quando `now()` é chamado sem parâmetros, ele usa o timezone padrão da JVM, mas também é possível especificar qual timezone deve ser usado, passando um `ZoneId` como parâmetro. E em ambos os casos será usado o relógio do sistema operacional para obter o valor do timestamp atual.

Mas muitos sistemas possuem condições específicas que dependem da data e hora atual. Por exemplo, o sistema deve realizar alguma ação somente se a data atual for antes ou depois de determinado dia, ou somente se for horário comercial etc. Como fazer testes deste tipo sem alterar o relógio do servidor ou mudar as chamadas de `now()` ? No `java.time` podemos usar a classe `java.time.Clock` .

19.1 Funcionamento básico de java.time.Clock

Um `Clock` é responsável por determinar qual é o timestamp atual. Esta informação é retornada como um `Instant` , através do método `instant()` . Para converter este `Instant` para uma data, hora e offset específicos, precisamos de um timezone, que é obtido pelo método `getZone()` , que retorna um `ZoneId` .

Quando o método `now()` é chamado sem parâmetros, internamente é usado o método estático `Clock.systemDefaultZone()` . O `Clock` retornado por este método usa o timezone padrão da JVM como seu `ZoneId` . E

para retornar o `Instant` correspondente ao timestamp atual, ele usa, segundo a documentação, "o melhor relógio disponível no sistema".

Conforme já explicado anteriormente, no Java 8 "*o melhor relógio disponível no sistema*" é na verdade o método `System.currentTimeMillis()` , que retorna o timestamp em milissegundos. Ou seja, mesmo que o Sistema Operacional no qual a JVM está rodando tenha um relógio com precisão maior (como microssegundos ou nanossegundos), o `Clock` está limitado aos milissegundos. A partir do Java 9, o `Clock` de fato usa todas as casas decimais que o relógio do sistema possui (limitado aos nanossegundos, que é o máximo que a API suporta).

Quando passamos um `ZoneId` para o método `now()` , internamente é chamado `Clock.system(zoneId)` . Este método retorna um `Clock` que usa o relógio do sistema, mas em vez de usar o timezone padrão da JVM, usa o `ZoneId` que foi passado.

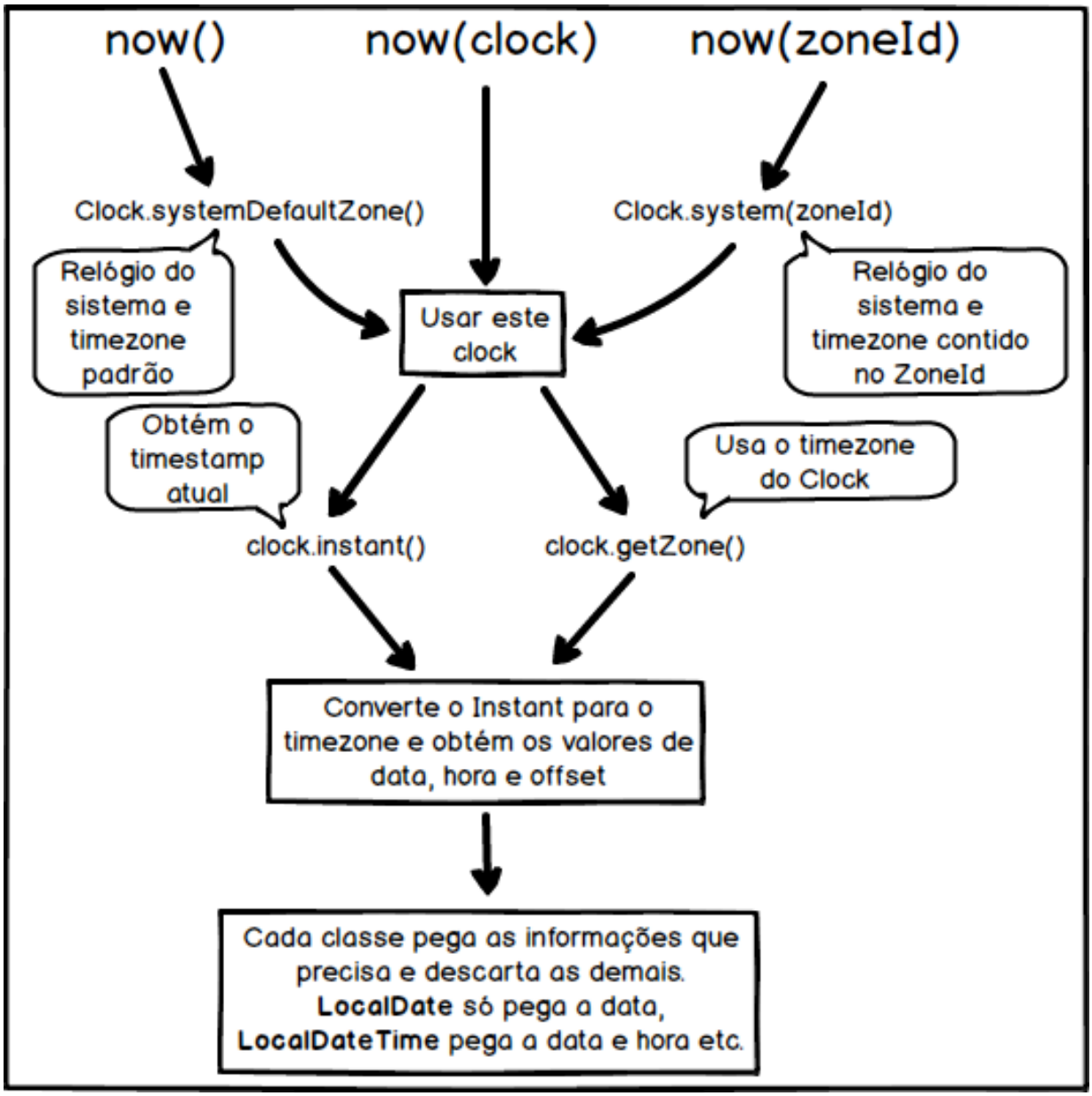

Figura 19.1: No fim, todos os métodos now() acabam usando um Clock

Mas para que precisamos nos preocupar com o `Clock` se podemos simplesmente usar o método `now()` sem parâmetros ou passando um `ZoneId` ? A vantagem é tornar o código mais testável, conforme veremos a seguir.

19.2 Manipulando o tempo

Existem várias situações nas quais o comportamento do código depende da data atual. É muito comum casos de uso como: se hoje for dia útil (ou for depois de alguma data específica, ou qualquer outro critério envolvendo a data/hora atual), o sistema faz X, se não, faz Y. Como testar esses casos sem ter que mudar as chamadas de `now()` para algum valor fixo ou alterar relógio do sistema?

No `java.time` podemos criar um `Clock` fixo, que sempre retorna **o mesmo valor** para o timestamp atual (em vez de usar o relógio do sistema). Por exemplo, se quisermos criar um `Clock` que sempre retorna "4 de maio de 2018, às 17:00 em São Paulo" como o instante atual, basta criar o `Instant` e o `ZoneId` correspondentes e usar o método `fixed()` :

```
// 4 de maio de 2018, às 17:00 em São Paulo
ZonedDateTime agoraFixo = ZonedDateTime.parse("2018-05-04T17:00-
03:00[America/Sao_Paulo]");
// criar o Clock fixo
Clock clock = Clock.fixed(agoraFixo.toInstant(), agoraFixo.getZone());
```

Primeiro, foi criado um `ZonedDateTime` com a data, hora e timezone que queremos (4 de maio de 2018, às 17:00 em São Paulo). Em seguida, passei o instante e timezone correspondentes para o método `Clock.fixed()` , que cria um `Clock` fixo (vale lembrar que tanto o `Instant` quanto o `ZoneId` podem ser criados de qualquer outra maneira, eu apenas usei `ZonedDateTime` como um exemplo).

Com isso, este `Clock` sempre retornará o mesmo `Instant` (que no caso é `2018-05-04T20:00:00Z`) como sendo o instante atual, e este `Instant` será convertido para o timezone `America/Sao_Paulo` , para que sejam obtidos os valores de data, hora e offset. Para usar o `Clock` , basta passá-lo para o método `now()` :

```
// 2018-05-04
LocalDate data = LocalDate.now(clock);
// 2018-05-04T17:00-03:00
```

```
OffsetDateTime dataHoraOffset = OffsetDateTime.now(clock);
// 2018-05-04T20:00:00Z
Instant instant = Instant.now(clock);
```

E se quisermos obter a data/hora correspondente a esse mesmo instante, mas em outro timezone? Neste caso, basta criar outro `Clock` com o método `withZone(ZoneId)` , que mantém o valor do `Instant` fixo e muda somente o timezone:

```
// Mudar o timezone do Clock
clock = clock.withZone(ZoneId.of("Asia/Tokyo"));
LocalDate data = LocalDate.now(clock); // 2018-05-05
```

Como o `ZoneId` mudou para `Asia/Tokyo` , o `Instant` é convertido para este timezone e o resultado é 5 de maio de 2018. Vale lembrar que `Clock` , assim como as demais classes da API, é imutável e o método `withZone()` retorna outra instância com o `ZoneId` modificado.

Para que seja possível usar um `Clock` fixo nos testes, é preciso que todas as chamadas de `now()` do seu código recebam um `Clock` como parâmetro. Depois basta configurar sua aplicação de forma que ela carregue a instância correta de acordo com o ambiente: em produção, você usaria `Clock.systemDefaultZone()` para usar o relógio do sistema no timezone padrão, ou `Clock.system(zoneId)` para usar um timezone específico; no ambiente de testes, seria usado o `Clock` fixo com o valor que você precisar para cada caso de teste.

Ou seja, o `Clock` passa a ser apenas mais uma dependência do seu código, podendo ser tratada como qualquer outra. Suas instâncias podem ser criadas por *factories* e passadas para as classes que as usam, gerenciadas pela injeção de dependências do seu framework preferido e por aí vai.

Exemplo prático de teste com Clock

Vamos supor que temos uma classe que calcula a idade, a partir da data de nascimento:

```
public class CalculaIdade {
    // Clock a ser usado para obter a data atual
    private Clock clock;
    public CalculaIdade(Clock clock) {
        this.clock = clock;
    }
    public long getIdade(LocalDate dataNasc) {
        return ChronoUnit.YEARS.between(dataNasc, LocalDate.now(clock));
    }
}
```

A classe recebe um `Clock` no seu construtor, que será usado para calcular a data atual. O método `getIdade()` recebe a data de nascimento e verifica quantos anos há entre ela e a data atual. Mantive o código bem simples para fins didáticos, por isso não incluí verificações como "dar erro se a data de nascimento estiver no futuro", entre outras coisas que um sistema real teria.

Em produção, a classe seria criada desta maneira:

```
CalculaIdade calc = new CalculaIdade(Clock.systemDefaultZone());
```

`Clock.systemDefaultZone()` também poderia ser substituído por `Clock.system(zoneId)` , para usar um `ZoneId` específico em vez do timezone padrão da JVM. Outra alternativa é configurar seu framework de injeção de dependências preferido (caso você use) para retornar o `Clock` correto.

Já nos testes, basta trocar por um `Clock` fixo para cada caso de teste. Por exemplo, em JUnit poderíamos fazer assim:

```
// Clock fixo: 1 dia antes da data de nascimento
ZonedDateTime z = ZonedDateTime.parse("2010-01-09T00:00-
03:00[America/Sao_Paulo]");
CalculaIdade calc = new CalculaIdade(Clock.fixed(z.toInstant(),
z.getZone()));
LocalDate dataNasc = LocalDate.of(2000, 1, 10);
Assert.assertEquals(calc.getIdade(dataNasc), 9); // Assert do JUnit
```

Neste teste, a data de nascimento é 10 de janeiro de 2000 e o `Clock` foi configurado para retornar 9 de janeiro de 2010 como a data atual (1 dia antes do décimo aniversário). Por isso a idade calculada é `9`. Para testar todos os casos que deseja, basta trocar o `Clock` para os valores necessários para cada teste.

Agora vamos supor que a classe `CalculaIdade` não usa um `Clock` e simplesmente chama `LocalDate.now()`:

```
public long getIdade(LocalDate dataNasc) {
    return ChronoUnit.YEARS.between(dataNasc, LocalDate.now());
}
```

Neste caso, teríamos que usar *mocks* nos testes. Uma opção é usar o PowerMock (https://github.com/powermock/), uma das bibliotecas mais conhecidas para fazer *mocks* de métodos estáticos. Com isso é possível fazer `LocalDate.now()` retornar um valor específico.

19.3 Obtendo informações do timezone

A partir de um `ZoneId` é possível obter várias informações sobre o histórico de offsets do timezone. Estas informações são encapsuladas pela classe `java.time.zone.ZoneRules`, que pode ser obtida pelo método `getRules()`. Tendo o `ZoneRules`, podemos verificar o histórico de todas as transições do timezone (todas as vezes em que houve alguma mudança de offset), usando o método `getTransitions()`:

```
ZoneRules rules = ZoneId.of("America/Sao_Paulo").getRules();
rules.getTransitions().forEach(System.out::println);
```

O método `getTransitions()` retorna uma lista de `java.time.zone.ZoneOffsetTransition`, uma classe que representa uma transição (uma mudança de offset). O código anterior imprime várias linhas, uma para cada vez que o offset mudou no timezone

`America/Sao_Paulo` . Escolhi duas delas para vermos com mais detalhes:

```
Transition[Overlap at 2017-02-19T00:00-02:00 to -03:00]
Transition[Gap at 2017-10-15T00:00-03:00 to -02:00]
```

A primeira linha indica um *overlap* (quando um horário existe duas vezes, uma em cada offset), informando a data e hora em que ocorreu, além dos offsets usados antes e depois da transição. Neste caso, ela indica o fim do horário de verão em fevereiro de 2017: à meia-noite do dia 19 o offset era `-02:00` e mudou para `-03:00` (fazendo com que a hora local voltasse para 23:00, conforme já explicado no capítulo sobre timezones).

Em seguida, na segunda linha, temos um *gap* que corresponde ao início do horário de verão, e ela também informa a data e hora em que acontece, além dos offsets antes e depois da transição.

A classe `ZoneOffsetTransition` possui métodos para obter estas informações separadamente, como `getInstant()` , que retorna o `Instant` correspondente ao instante exato em que a transição ocorre, além de `getDateTimeBefore()` , `getOffsetBefore()` , `getDateTimeAfter()` e `getOffsetAfter()` , que podem ser usados para obter a data, hora e offset antes e depois da transição. Há também os métodos `isGap()` e `isOverlap()` para verificar se a transição é um *gap* ou *overlap*.

Quando é a próxima mudança de offset?

É possível saber quando será a próxima transição, a partir de um instante qualquer, usando o método `nextTransition()` :

```
// próxima mudança de offset
ZoneOffsetTransition proxima = rules.nextTransition(Instant.now());
```

Com isso podemos saber quando será a próxima mudança de offset, a partir de qualquer instante. No exemplo anterior foi usado o instante atual (`Instant.now()`), o que pode ser útil para alertar aos

usuários sobre a próxima mudança de horário de verão, por exemplo.

Para obter a transição anterior, há o método `previousTransition()`, que também recebe um `Instant` como parâmetro.

Outros métodos úteis da classe `ZoneRules` são:

- `getOffset(Instant)` : retorna o offset (`ZoneOffset`) usado pelo timezone em determinado instante. Lembre-se de que um timezone possui o histórico de offsets de uma região, então a cada instante podemos ter um offset diferente. Por isso este método recebe um `Instant` como parâmetro.
- `isDaylightSavings(Instant)` : verifica se em determinado instante o timezone está em horário de verão.
- `getValidOffsets(LocalDateTime)` : retorna uma lista de `ZoneOffset` com os offsets válidos para determinada data e hora. Na maioria dos casos a lista terá apenas um offset. Em casos de *gap* a lista será vazia (nenhum offset válido para esta data e hora), e em casos de *overlap* a lista terá dois offsets (a data e hora ocorre duas vezes, uma em cada offset).

Há ainda outros. Como sempre, consulte a documentação para mais detalhes (https://docs.oracle.com/javase/8/docs/api/java/time/zone/ZoneRules.html/).

Saber se dois timezones são iguais

Cada identificador da IANA (como `America/Sao_Paulo` ou `Europe/London`) corresponde a uma região específica do mundo, na qual o histórico de offsets é o mesmo. Se há dois nomes diferentes, significa que, em algum momento da história, há alguma diferença. Por exemplo, muitos países europeus atualmente usam as mesmas regras (mesmo offset, horário de verão começa e termina no mesmo dia etc.) mas cada um possui o seu próprio timezone, já que nem todos

aderiram às regras atuais ao mesmo tempo, portanto seus históricos são diferentes.

Porém, alguns timezones tiveram seus nomes mudados no passado (somente o nome, pois as regras se mantiveram). Isso é mapeado pelo arquivo de *backward*, que faz parte do TZDB e pode ser consultado no GitHub (https://github.com/eggert/tz/blob/2018e/backward/). Neste arquivo podemos ver que `America/Sao_Paulo` era chamado de `Brazil/East` (https://github.com/eggert/tz/blob/2018e/backward#L60/).

Por questões de retrocompatibilidade, `ZoneId` aceita ambos os identificadores. Porém, o método `equals()` não considera que eles são iguais:

```
ZoneId sp = ZoneId.of("America/Sao_Paulo");
ZoneId br = ZoneId.of("Brazil/East");
System.out.println(sp.equals(br)); // false
```

Isso acontece porque o método `equals()` compara o valor de `getId()`, ou seja, a `String` contendo o identificador da IANA. Para saber se dois timezones possuem as mesmas regras (o mesmo histórico de offsets), devemos comparar seus respectivos `ZoneRules`:

```
System.out.println(sp.getRules().equals(br.getRules())); // true
```

O método `equals()` da classe `ZoneRules` compara todo o histórico dos timezones: o instante exato em que cada transição ocorre, os offsets antes e depois etc. Se houver qualquer diferença, o retorno é `false`.

Encontrar os timezones que usam determinado offset ou abreviação

Na segunda parte do livro vimos como usar a API legada para encontrar os timezones que usam determinado offset ou abreviação, em determinado instante. Vamos ver o código equivalente usando `java.time`.

Cada timezone possui uma lista de offsets diferentes ao longo da história, por isso temos que escolher um `Instant` como referência. O mesmo vale para abreviações, pois muitos lugares usam uma abreviação diferente no horário de verão (além de outros que simplesmente mudaram suas abreviações em algum momento da história). Para obter o offset basta usar o `ZoneRules` , e para encontrar a abreviação, basta usar um `DateTimeFormatter` com o pattern `z` , não se esquecendo de percorrer todos os locales, já que a abreviação é *locale sensitive*.

No exemplo a seguir, vamos procurar os timezones que usam o offset `+02:00` ou a abreviação "EST", usando como referência o instante atual:

```
Instant referencia = Instant.now(); // 2018-05-04T20:00:00Z
// procurar timezones que usam offset +02:00
ZoneOffset offset = ZoneOffset.ofHours(2);
// pattern "z" retorna a abreviação do timezone
DateTimeFormatter fmt = DateTimeFormatter.ofPattern("z");
// percorrer todos os timezones
ZoneId.getAvailableZoneIds().stream().map(ZoneId::of).forEach(zone -> {
    if (offset.equals(zone.getRules().getOffset(referencia))) {
        // timezone usa o offset +02:00 no instante de referência
    }
    // verificar todos os locales, pois a abreviação é locale sensitive
    for (Locale locale : Locale.getAvailableLocales()) {
        String abrev =
fmt.withLocale(locale).format(referencia.atZone(zone));
        if ("EST".equals(abrev)) {
            // timezone usa a abreviação EST no instante de referência
        }
    }
});
```

Dependendo de quando você rodar o código (e da versão do TZDB da sua JVM), o resultado será diferente. No meu caso, obtive mais de 50 timezones que usam o offset `+02:00` , além de 8 que usam a abreviação "EST", usando como referência o instante equivalente a `2018-05-04T20:00:00Z` .

Lembre-se novamente: dado um offset ou uma abreviação, **não é possível obter um único timezone que os use**. O melhor que pode ser obtido é uma lista de timezones que usam aquele offset ou abreviação, em determinado instante. A partir desta lista você pode escolher um dos timezones, dependendo dos seus casos de uso.

É claro que em alguns casos essa lista pode ter apenas um timezone (no caso das abreviações, é mais comum, no caso dos offsets, nem tanto). Mesmo assim, é mais seguro seguir a abordagem de obter uma lista de timezones em vez de assumir que sempre será um só.

19.4 Criar um TemporalAdjuster

Nos capítulos anteriores vimos como usar a classe `TemporalAdjusters`, que possui vários métodos que retornam um `TemporalAdjuster` específico: um para obter o último dia do mês, outro para a primeira terça-feira do mês etc. Mas também é possível criar nossa própria implementação.

A interface `TemporalAdjuster` possui o método `adjustInto()`, que recebe como parâmetro um `Temporal`. Ou seja, além dos métodos `get()` e `getLong()`, é possível usar `plus()`, `minus()` e `with()`, para que possamos fazer quaisquer ajustes que precisarmos. O retorno do método também é um `Temporal`.

Por exemplo, para criar um `TemporalAdjuster` que retorna uma data 3 meses no futuro, no primeiro dia do mês, o código ficaria assim:

```
TemporalAdjuster adjuster = temporal -> {
    return temporal
        // 3 meses no futuro
        .plus(3, ChronoUnit.MONTHS)
        // no dia 1
        .with(ChronoField.DAY_OF_MONTH, 1);
};
```

```
// 2018-08-01
LocalDate data = LocalDate.of(2018, 5, 4).with(adjuster);
```

`TemporalAdjuster` é uma interface funcional, por isso é possível usar a sintaxe de *lambda*. O parâmetro é um `Temporal` , então o ajuste funcionará com qualquer classe de data e hora da API (pois todas implementam `Temporal`), desde que ela suporte as unidades e campos usados, claro.

O exemplo anterior, por exemplo, só funciona com classes que suportam `ChronoUnit.MONTHS` e `ChronoField.DAY_OF_MONTH` . Portanto podemos usá-lo também com `LocalDateTime` , `OffsetDateTime` e `ZonedDateTime` , mas uma exceção seria lançada se usássemos em um `LocalTime` ou `YearMonth` (este último suporta `ChronoUnit.MONTHS` mas não suporta `ChronoField.DAY_OF_MONTH`).

O `TemporalAdjuster` é muito útil para encapsular a lógica de manipulação de uma data, e deixa seu código mais flexível (pois é possível escolher a implementação em tempo de execução, por exemplo) e testável (já que pode ser facilmente substituído por *mocks* nos testes). Além disso, é um ótimo substituto para aquela classe utilitária (geralmente chamada de `DateUtils` , `CalendarUtils` etc.) que todo projeto tem, cheia de métodos estáticos que manipulam `Date` ou `Calendar` . E por falar nestas classes, no próximo capítulo veremos alguns detalhes importantes para migrar seu código da API legada para o `java.time` .

Capítulo 20
Migração entre java.time e API legada

Agora que já temos um bom entendimento de ambas as APIs, podemos ver como trabalhar com elas simultaneamente. Como ainda existe muito código legado usando `Date` e `Calendar` , inclusive em bibliotecas e frameworks "famosos", nem sempre será possível ter um código usando somente o `java.time` .

20.1 Relação entre Date e as novas classes

`Date` representa um timestamp: um ponto específico na linha do tempo. Esta classe não representa um único valor de dia, mês, ano, hora, minuto e segundo, pois em cada timezone ela pode corresponder a uma data e hora diferente. A classe do `java.time` que representa esse mesmo conceito é `Instant` , por isso no Java 8 foram adicionados dois métodos novos em `Date` para converter de/para `Instant` , conforme mostra o próximo exemplo:

```
Date date = new Date();
// converter para Instant
Instant instant = date.toInstant();
// converter de volta para Date
date = Date.from(instant);
```

Os métodos `toInstant()` e `Date.from()` retornam um `Instant` e um `Date` que correspondem ao mesmo timestamp. Só há um porém.

`Date` possui precisão de milissegundos (3 casas decimais na fração de segundos), enquanto a precisão de `Instant` é de nanossegundos (9 casas decimais). Ao converter um `Instant` para `Date` , o valor é truncado para milissegundos e os demais dígitos são perdidos. Exemplo:

```
// valor com nanossegundos (9 casas decimais)
Instant instant = Instant.parse("2018-01-01T10:00:00.123456789Z");
// converter para Date (os 6 últimos dígitos são perdidos)
Date date = Date.from(instant);
// converter de volta para Instant
instant = date.toInstant(); // 2018-01-01T10:00:00.123Z
```

O `Instant` possui 9 dígitos nas frações de segundo (123456789 nanossegundos). Ao converter para `Date` , somente os 3 primeiros dígitos são mantidos (`123`), pois esta classe possui precisão de milissegundos. Portanto, os demais dígitos (`456789`) são perdidos. Ao converter o `Date` de volta para `Instant` , o valor da fração de segundos é `123` e o resultado é `2018-01-01T10:00:00.123Z` .

Caso seja necessário obter o valor do `Instant` original (com todas as 9 casas decimais), o valor dos nanossegundos deve ser guardado separadamente, para que possa ser restaurado depois:

```
// valor com nanossegundos (9 casas decimais)
Instant instant = Instant.parse("2018-01-01T10:00:00.123456789Z");
// converter para Date
Date date = Date.from(instant);
// guardar o valor dos nanossegundos
int nano = instant.getNano();
// converter de volta para Instant
Instant instant2 = date.toInstant()
    // restaurar o valor original dos nanossegundos
    .with(ChronoField.NANO_OF_SECOND, nano);
```

Converter Date para LocalDate

Um caso de uso comum é converter `Date` para um `LocalDate` . Não há uma forma direta, pois `Date` representa um timestamp, um valor que corresponde a uma data diferente em cada timezone. Por isso, a conversão deve ser feita considerando algum timezone:

```
Date date = new Date();
LocalDate localDate = date
    // converter para Instant
    .toInstant()
```

```
    // converter para um timezone
    .atZone(ZoneId.of("America/Sao_Paulo"))
    // obter o LocalDate
    .toLocalDate();
```

Primeiro, o `Date` é convertido para `Instant`, usando-se o método `toInstant()`. Depois este `Instant` é convertido para algum timezone, através do método `atZone()`, que retorna um `ZonedDateTime`. Por fim, o método `toLocalDate()` é chamado, retornando o `LocalDate`.

A API foi desenhada de modo a "pedir" explicitamente por um `ZoneId`. Pode parecer um obstáculo para dificultar nossa vida, mas eu vejo isso mais como uma forma de nos fazer pensar da maneira correta com relação aos conceitos de timestamp e data/hora: o timestamp corresponde a uma data e hora diferente em cada timezone, então você **precisa** dizer qual é o timezone que será usado.

É uma abordagem diferente da API legada, que usa o timezone padrão da JVM (até quando não queremos) e faz várias conversões implícitas (e nem sempre óbvias ou intuitivas), sem deixar claro para o desenvolvedor o que, de fato, está acontecendo.

Aliás, muitas APIs de data adotam esta abordagem mais "simples": tratam os conceitos de timestamp e a data/hora local como se fossem uma coisa só, usando algum timezone arbitrário para converter entre um e outro – e não se iluda, para que o timestamp se transforme em uma data e hora, é preciso usar algum timezone ou offset, **sempre**. O `java.time` foi pensado de modo a manter estes dois conceitos separados, e a conversão entre eles sempre exige um `ZoneId` ou `ZoneOffset`.

Poderíamos ter usado `ZoneId.systemDefault()` no lugar de `ZoneId.of()` para que o timezone padrão da JVM fosse aplicado, de forma a "simular" o que a API legada faz. Pode parecer redundante, mas pelo menos estará indicado que ali tem um timezone sendo utilizado, além de deixar o código mais fácil de ser alterado, caso precise mudar para um timezone específico.

Converter o `LocalDate` de volta para `Date` também não é algo tão direto. `LocalDate` só possui o dia, mês e ano, mas para termos um valor de timestamp é necessário ter também o horário e o offset. Neste caso, teremos que setar algum valor arbitrário para estes campos, para só então obter o `Instant` e passá-lo para o método `Date.from()` . Exemplo:

```
LocalDate data = LocalDate.now();
Date date = Date.from(
    // setar algum horário
    data.atTime(10, 0)
    // setar timezone
    .atZone(ZoneId.of("America/Sao_Paulo"))
    // converter para Instant
    .toInstant());
```

Ao setar o timezone com o método `atZone()` , o offset é automaticamente calculado. Se for o caso, podem ser feitos ajustes para os casos de *gap* ou *overlap*, conforme já explicado no capítulo sobre `ZonedDateTime` .

O problema é que diferentes valores para o horário e offset resultarão em timestamps diferentes. Se o objetivo é restaurar o `Date` original, então o valor do timestamp (ou do `Instant`) deve ser guardado separadamente. Caso contrário, não será possível obter o `Date` original.

20.2 Conversões envolvendo Calendar

`Calendar` possui um timestamp, mas também possui um timezone e os respectivos campos de data e hora (dia, mês, ano, horas, minutos, segundos) com os valores ajustados de acordo com o seu timezone.

Por ter um timestamp, esta classe também possui o método `toInstant()` , que retorna um `Instant` contendo o mesmo valor do

timestamp. Mas não há um método para converter o `Instant` de volta para `Calendar` . Uma alternativa é setar o valor numérico do timestamp diretamente:

```
Instant instant = // algum valor qualquer
// converter para Calendar
Calendar cal = Calendar.getInstance();
// setar o valor do timestamp
cal.setTimeInMillis(instant.toEpochMilli());
```

`Calendar.getInstance()` cria uma instância com o timezone padrão da JVM, então este é o timezone que o `Calendar` usará para obter os campos de data e hora a partir do timestamp. Como já vimos anteriormente, também é possível passar um `TimeZone` para o método `getInstance()` , para que o `Calendar` use outro timezone.

Outro detalhe é que a classe `Calendar` é abstrata e possui várias implementações diferentes (já que existem vários calendários diferentes em uso no mundo). `getInstance()` retorna uma destas implementações de acordo com o locale padrão da JVM. Para a grande maioria dos locales, o retorno é um `java.util.GregorianCalendar` , que representa o calendário gregoriano, que é usado atualmente pela maior parte do mundo (os demais tipos de calendários estão além do escopo deste livro e não serão explorados).

O equivalente de `GregorianCalendar` é a classe `ZonedDateTime` , já que ambos representam uma data e hora em determinado timezone. Por isso no Java 8 foram adicionados os respectivos métodos de conversão, conforme mostra o próximo exemplo:

```
ZonedDateTime zonedDt = // algum valor qualquer
GregorianCalendar cal = GregorianCalendar.from(zonedDt);
ZonedDateTime zdt = cal.toZonedDateTime();
```

Os métodos `from()` e `toZonedDateTime()` preservam o timezone e o timestamp ao fazer a conversão entre os tipos. Com isso, ambos correspondem ao mesmo instante e aos mesmos valores de data, hora e offset. Vale lembrar que, ao converter de `ZonedDateTime` para `GregorianCalendar` , os nanossegundos são truncados para

milissegundos. Quanto ao fato de esses métodos não estarem em `Calendar` , há uma breve discussão sobre isso na lista de emails do OpenJDK (http://mail.openjdk.java.net/pipermail/threeten-dev/2013-June/001476.html/).

20.3 Conversão entre TimeZone e ZoneId

Além de `Date` e `Calendar` , outras classes legadas também possuem métodos novos para converter de/para o `java.time` . A classe `TimeZone` , por exemplo, agora possui o método `toZoneId()` , que retorna um `ZoneId` com o mesmo identificador da IANA. Assim, um `TimeZone` cujo ID é `America/Sao_Paulo` retornará um `ZoneId` com este mesmo identificador.

Mas há um detalhe: `TimeZone.getTimeZone()` aceita algumas abreviações, como "EST" e "IST" (que já vimos anteriormente que não são timezones de fato). Ao convertê-los para `ZoneId` , os resultados podem não ser o que você espera:

```
System.out.println(TimeZone.getTimeZone("IST").toZoneId());
System.out.println(TimeZone.getTimeZone("EST").toZoneId());
```

A saída deste código é:

```
Asia/Kolkata
-05:00
```

"IST" — uma abreviação usada na Índia, Irlanda e Israel — foi convertida para um `ZoneId` com um timezone da Índia, enquanto "EST" — que é usada nos EUA, Panamá e Cancun, entre outros — foi mapeado para o **offset** `-05:00` (sequer retornou um timezone).

Se o `TimeZone` foi criado com um identificador da IANA, a conversão será feita sem problemas, mas cuidado ao usar as abreviações, pois podem acontecer essas conversões inesperadas.

Para converter o `ZoneId` para um `TimeZone` , basta passá-lo para o método `getTimeZone()` . Como `ZoneId` não aceita as abreviações, não há o problema do timezone resultante ser algo inesperado:

```
ZoneId zoneId = ZoneId.of("Asia/Tokyo");
// converter ZoneId para TimeZone
TimeZone tz = TimeZone.getTimeZone(zoneId);
```

20.4 Classes do pacote java.sql

As subclasses de `Date` no pacote `java.sql` também possuem métodos de conversão:

- `java.sql.Date` possui o método `toLocalDate()` , que retorna um `LocalDate` , e `valueOf(LocalDate)` , que converte o `LocalDate` para `java.sql.Date` .
- `java.sql.Time` possui o método `toLocalTime()` , que retorna um `LocalTime` , e `valueOf(LocalTime)` , que converte o `LocalTime` para `Time` .
- `java.sql.Timestamp` possui os métodos `from(Instant)` e `toInstant()` , para converter de e para `Instant` , respectivamente. Como `Timestamp` possui precisão de nanossegundos, a conversão é feita sem perda de precisão. Além disso, esta classe também possui métodos para converter de e para `LocalDateTime` : `toLocalDateTime()` e `valueOf(LocalDateTime)` .

As conversões envolvendo os tipos locais usam sempre o timezone padrão da JVM: lembre-se de que `java.sql.Date` , `Time` e `Timestamp` representam na verdade um timestamp (por serem subclasses de `java.util.Date`) e seus valores de data e hora correspondem a este timestamp no timezone padrão da JVM. Podemos ver este comportamento no próximo exemplo:

```
// timestamp 1525464000000 -> 2018-05-04T17:00-03:00
java.sql.Date date = new java.sql.Date(1525464000000L);
```

```
TimeZone.setDefault(TimeZone.getTimeZone("America/Sao_Paulo"));
System.out.println(date.toLocalDate());
TimeZone.setDefault(TimeZone.getTimeZone("Asia/Tokyo"));
System.out.println(date.toLocalDate());
```

Primeiro foi criado um `java.sql.Date` com um valor de timestamp equivalente a `2018-05-04T17:00-03:00` . Depois o timezone padrão é mudado duas vezes. O método `toLocalDate()` converte o valor do timestamp para o timezone padrão que estiver setado no momento, e com isso obtém o valor da data que será usada no `LocalDate` . Como o timestamp corresponde a 4 de maio de 2018 em São Paulo, mas a 5 de maio em Tóquio, a saída é:

```
2018-05-04
2018-05-05
```

O método `valueOf(LocalDate)` também sofre deste mesmo problema. Internamente, ele pega os valores do dia, mês e ano do `LocalDate` e seta o horário para meia-noite no timezone padrão da JVM. Isso acaba gerando um `java.sql.Date` com valores de timestamp completamente diferentes, dependendo do timezone padrão que estiver setado. Estes problemas também acontecem com `Time` e `Timestamp` e seus respectivos métodos de conversão de/para o `java.time` .

Conforme mencionado anteriormente no capítulo sobre o pacote `java.sql` , somente a partir do JDBC 4.2 é possível usar as classes do `java.time` para gravar e ler campos de data em um banco de dados. Se o banco de dados que você está usando disponibiliza um driver compatível, é possível usar `java.sql.PreparedStatement` e `java.sql.ResultSet` :

```
PreparedStatement ps = ...
// seta o java.time.Instant
ps.setObject(1, Instant.now());

// obter o Instant do banco
ResultSet rs = ...
Instant instant = rs.getObject(1, Instant.class);
```

Consulte a documentação do seu banco de dados para saber como cada classe do `java.time` é mapeada para os tipos de data do SQL.

20.5 Quando não converter entre as APIs

Creio que não exista uma regra definitiva, mas deixo aqui algumas recomendações.

Não misture as APIs desnecessariamente. Em código novo que não dependa da API legada, recomendo usar somente o `java.time` . Não há motivo pelo qual fazer `TimeZone.getDefault().toZoneId()` , por exemplo, sendo que `ZoneId.systemDefault()` é mais direto. Quer a data atual? Use `LocalDate.now()` . Precisa da data e hora? `LocalDateTime.now()` .

E para obter o instante atual? Já vi muito código assim:

```
Calendar cal = Calendar.getInstance();
cal.setTime(new Date());
Instant now = Instant.ofEpochMilli(cal.getTimeInMillis());
```

`getInstance()` retorna um `Calendar` com a data e hora atual, então setar `new Date()` é redundante e desnecessário. Em seguida, é usado o timestamp para criar o `Instant` , mas estas 3 linhas poderiam ser substituídas simplesmente por `Instant.now()` .

> E se você precisar *apenas* do valor numérico do timestamp em milissegundos, nem precisa criar classes à toa, basta usar `System.currentTimeMillis()` .

"*Ah, mas talvez fizeram assim porque `Instant` tem precisão de nanossegundos e eles só precisam dos milissegundos*".

Nesse caso, bastaria usar o método `truncatedTo()` , que trunca determinados valores. Se eu quero que a precisão máxima seja em

milissegundos, basta usar a `TemporalUnit` equivalente:

```
// truncar o Instant para milissegundos
Instant instant = Instant.now().truncatedTo(ChronoUnit.MILLIS);
```

Com isso, todos os valores menores que `ChronoUnit.MILLIS` (ou seja, campos cuja duração seja menor que 1 milissegundo) terão seus valores truncados. No caso, todas as casas decimais depois da terceira serão setadas para zero: se o valor do `Instant` original fosse `2018-08-17T12:15:10.123456789Z` , o resultado de `truncatedTo(ChronoUnit.MILLIS)` seria `2018-08-17T12:15:10.123Z` .

O método `truncatedTo()` é útil para setar vários valores de uma vez para zero. Para setar determinados campos individualmente e/ou com outros valores diferentes de zero, use os métodos `with` .

Outra alternativa é usar um `Clock` que ignora durações menores que 1 milissegundo:

```
Clock clock = Clock.tick(Clock.systemDefaultZone(), Duration.ofMillis(1));
Instant instant = Instant.now(clock);
```

O método `tick()` recebe um `Clock` (no caso, `systemDefaultZone()` , que usa o relógio do sistema e o timezone padrão da JVM) e um `Duration` indicando a duração mínima a ser considerada – no caso, usei 1 milissegundo, indicando que esta será a precisão máxima a ser usada pelos timestamps retornados por este `Clock` . Ou seja, mesmo que o relógio do sistema suporte mais que 3 casas decimais nas frações de segundo, esse `Clock` retornará timestamps com no máximo 3.

Praticamente tudo que era possível fazer com `Date` e `Calendar` também é com `java.time` (muitas vezes de uma maneira melhor). Consulte a documentação, pois na maioria das vezes não será necessário misturar as APIs.

20.6 Quando misturar as APIs?

Particularmente, um dos casos em que misturo a API legada com `java.time` é quando estou usando bibliotecas que ainda precisam de `Date` ou `Calendar` e tenho que fazer alguma manipulação nestes objetos. Por exemplo, suponha que estou usando uma API que retorna um `Date` , que eu preciso passar para outra API, só que o valor recebido por esta deve ter o horário setado para o fim do dia:

```
Date date = algumaAPI.getDate(); // API que retorna java.util.Date
date = ... // manipular o Date (setar horário para fim do dia)
outraAPI.fazAlgo(date); // passar o Date para outra API
```

É possível fazer estes cálculos usando `Calendar` , mas neste caso eu prefiro converter o `Date` para o `java.time` (seja para `LocalDate` , `Instant` , ou qualquer outra classe que seja mais fácil de trabalhar, dependendo do que preciso fazer), efetuar os cálculos e converter o resultado de volta para `Date` .

Para obter o final do dia, você provavelmente pensou que basta setar o horário para 23:59:59 e pronto. Mas graças aos *gaps* e *overlaps*, nem sempre essa abordagem funciona. Um exemplo é o timezone `America/Sao_Paulo` : quando termina o horário de verão, há um overlap das 23:00 às 23:59 (este intervalo de horários ocorre duas vezes) e você teria que ajustar manualmente para obter a segunda ocorrência de 23:59 (usando `withLaterOffsetAtOverlap()` , como já vimos anteriormente).

Já no timezone `Asia/Dhaka` (Bangladesh), até 2009, havia um *gap* que fazia com que o dia terminasse às 22:59 (pois às 23:00 o relógio pulava diretamente para meia-noite do dia seguinte). Veja no próximo exemplo para entender melhor:

```
ZonedDateTime fimDoDiaErrado = LocalDate.of(2009, 6, 19)
    // setar horário para 23:59:59.999999999
    .atTime(LocalTime.MAX)
    // converter para o timezone
    .atZone(ZoneId.of("Asia/Dhaka"));
```

O horário é setado para 23:59:59.999999999 (usei a constante `LocalTime.MAX` para facilitar) e, em seguida, o valor é convertido para o timezone `Asia/Dhaka` . Como este horário faz parte de um *gap*, ele é corrigido para o próximo horário válido — no caso, 00:59 do dia seguinte (já vimos em detalhes este comportamento no capítulo sobre `ZonedDateTime`). Por isso o resultado é `2009-06-20T00:59:59.999999999+07:00[Asia/Dhaka]` (00:59 do dia 20, que com certeza não é "o final do dia 19").

O modo correto de se obter o último instante do dia é: primeiro obter o **início do dia seguinte** e depois subtrair 1 nanossegundo. Com isso eu garanto que peguei o último instante daquele dia:

```
ZonedDateTime fimDoDia = LocalDate.of(2009, 6, 19)
    // início do dia seguinte no timezone
    .plusDays(1).atStartOfDay(ZoneId.of("Asia/Dhaka"))
    // subtrair 1 nanossegundo
    .minusNanos(1);
```

O resultado é `2009-06-19T22:59:59.999999999+06:00` . Apenas para conferir que este é, de fato, o último instante do dia no timezone `Asia/Dhaka` , vamos somar 1 nanossegundo a ele (como ele é "o final do dia", o resultado deve ser o início do dia seguinte):

```
System.out.println(fimDoDia);
// somar 1 nanossegundo - deve ir para o início do dia seguinte
System.out.println(fimDoDia.plusNanos(1));
```

Ao somar 1 nanossegundo, o resultado é o início do dia seguinte. A saída é:

```
2009-06-19T22:59:59.999999999+06:00[Asia/Dhaka]
2009-06-20T00:00+07:00[Asia/Dhaka]
```

Para converter de volta para `Date` , basta fazer `Date.from(fimDoDia.toInstant())` .

Qualquer cálculo que possa ser feito mais facilmente e de forma mais precisa no `java.time` (ou seja, praticamente todos) pode seguir

esta recomendação: converta as classes legadas para a nova API, faça os cálculos e converta de volta para a API legada.

20.7 Diferenças entre SimpleDateFormat e DateTimeFormatter

Já falamos um pouco sobre as diferenças entre `SimpleDateFormat` e `DateTimeFormatter`. A principal delas, especialmente quando for converter seu código de uma para outra, é prestar atenção no pattern. Nem todas as letras funcionam exatamente da mesma maneira. Já vimos o exemplo de `u`, que na API legada corresponde ao dia da semana, e no `java.time` corresponde ao ano. Há também letras que não existem na API legada, como o `e` e `c` para o dia da semana, o `Q` para o trimestre, o `V` para o nome do timezone, entre outros.

Outra diferença importante é que o `java.time` é mais rigoroso que a API legada. Um exemplo é a forma com que é tratado o pattern `h`. Ele corresponde ao campo *hour-of-am-pm* (algo como "hora AM/PM"), com valores de 1 a 12. Este campo é ambíguo, pois qualquer valor só faz sentido se tivermos o indicador AM ou PM. Por exemplo, 3 AM corresponde a 3 da manhã e 3 PM corresponde a 3 da tarde (ou 15:00).

Apesar disso, `SimpleDateFormat` aceita normalmente valores maiores que 12, o que não é nenhuma surpresa (dado que já vimos coisas bem piores que esta classe aceita):

```
SimpleDateFormat sdf = new SimpleDateFormat("hh:mm");
// faz parsing de 17:00, mesmo que o campo "h" tenha valores de 1 a 12
System.out.println(sdf.parse("17:00"));
```

Por outro lado, `DateTimeFormatter` não aceita tal condição, mesmo se estiver em modo leniente:

```
DateTimeFormatter fmt = DateTimeFormatter.ofPattern("hh:mm")
    .withResolverStyle(ResolverStyle.LENIENT);
System.out.println(LocalTime.parse("17:00", fmt));
```

O código anterior lança uma exceção, pois o pattern `h` só aceita valores de 1 a 12. Mas mesmo se a `String` fosse `"10:00"` daria erro, pois ao usar o pattern `h` temos que definir se a hora é AM ou PM: ou a `String` de entrada tem esta informação (cujo parsing é feito com o pattern `a`), ou então deve-se setar um valor predefinido, usando `parseDefaulting()` com o campo `ChronoField.AMPM_OF_DAY` . Ou ainda, usa-se o pattern `H` (maiúsculo), que aceita horas com valores de 0 a 23.

Portanto, não adianta simplesmente copiar o pattern que funcionava em `SimpleDateFormat` e achar que funcionará da mesma maneira em `DateTimeFormatter` . Leia a documentação e veja qual é o equivalente de cada campo na nova API. Veja também se seu código não pode ser substituído por algum dos novos recursos do `java.time` , como os patterns opcionais e campos com valores predefinidos.

20.8 Considerações finais

Com isso chegamos ao final do livro. Obviamente não cobrimos 100% da API e nem vimos todos os casos de uso possíveis, já que não era esta a intenção (e de qualquer forma, não caberia tudo em um livro só). Mas acredito que vimos os principais pontos envolvendo datas, horas e timezones, que vão ajudar você a lidar melhor com estes conceitos no seu código. E isso vale não só para Java, mas para qualquer linguagem, já que muitas coisas costumam ser bem parecidas.

Sempre que você encontrar uma função que recebe um timestamp e retorna uma data (um dia, mês e ano específico) ou vice-versa, procure saber qual o timezone que esta função está usando para converter o timestamp de/para a data, e se é possível mudá-lo ou

configurá-lo (ou ao menos obtê-lo), para não ser pego de surpresa quando for manipular a data.

Ao fazer parsing e formatação, consulte a documentação, pois cada linguagem possui seus próprios patterns, nem sempre iguais aos do Java. E lembre-se do princípio básico: parsing para converter `String` para data e formatação para fazer o oposto. Veja quais são as funções que fazem cada uma destas operações e quais os tipos de data disponíveis.

Verifique também como a linguagem lida com timezones: se ela suporta os identificadores da IANA, onde ficam estas informações e como atualizá-las. Muitas linguagens atualmente usam o TZBD e possuem algum mecanismo para atualizá-lo, similar ao TZUpdater Tool. Veja se a documentação não está erroneamente chamando offsets de "timezones", e faça testes para ver o que acontece nas transições do horário de verão (se o horário e offset mudam corretamente, por exemplo).

Enfim, muitos dos conceitos vistos podem ser aplicados em outras linguagens. Sempre verifique a documentação para saber se e como tais conceitos são implementados. Nem todas as linguagens possuem uma API completa como o `java.time` , então é importante saber as limitações e, se for o caso, implementar o que falta.

Lidar com datas da maneira correta é mais complicado do que parece, mas espero ter tornado este assunto um pouco menos difícil para você.