Até o Java 7, tínhamos as classes `Date` e `Calendar` para representar datas. Para convertê-las para `String`s, a forma mais fácil era com o uso do `SimpleDateFormat`.
O Java 8 introduziu uma nova API de datas. Como é que ela pode ser usada? Como faço para integrar ela com as `Date` e `Calendar` que são usadas em código já existente e legados? Qual é a relação dele com o Joda-Time?

Os problemas da API legada

As classes java.util.Date e java.util.Calendar, bem como as subclasses java.util.GregorianCalendar, java.sql.Date, java.sql.Time e java.sql.Timestamp, são notórias por serem mal-arquitetadas e por serem classes difíceis de se utilizar devido ao fato de a API delas ter sido mal-elaborada. Elas funcionam corretamente se forem usadas com o devido cuidado, mas o código delas acumula várias más práticas de programação e problemas recorrentes que atrapalham a vida dos programadores em Java.
Além disso, essas classes todas são mutáveis, o que as torna inapropriadas de serem utilizadas em alguns casos. Por exemplo:

```
Date a = ...;
Date d = new Date();
pessoa.setAtualizacao(d); // Define a data de atualização.

// Em algum lugar bem longe do código acima:
d.setTime(1234); // A data de atualização muda magicamente de forma misteriosa.
```

Um outro problema nessas classes é que elas não são *thread-safe*. Como essas classes são mutáveis, isso até que é esperado. Nos casos onde elas não sofram mutações enquanto estiverem sendo usadas, isso não deverá causar problemas referentes a *thread-safety* para a maioria dessas classes. Porém com a classe `SimpleDateFormat`, a situação é diferente. Compartilhar uma instância de `SimpleDateFormat` entre diversas threads causará resultados imprevisíveis mesmo se a instância de `SimpleDateFormat` não sofrer alterações/mutações externas. Isso ocorre porque durante o processo de *parse* ou de formatação de uma data, a classe `SimpleDateFormat` altera o estado interno de si mesma.
Por isso que no Java 8, novas classes foram elaboradas para substituí-las.

A nova API

Primeiramente, na nova API **todas as classes são imutáveis e *thread-safe***. Somente isso já as torna bem mais fáceis de se utilizar. Além disso, a API delas foi bastante planejada, discutida e exercitada para ficar coerente.
As classes mais utilizadas são as seguintes:

- LocalDate - Representa uma data sem informação de hora e nem de fuso horário.
- LocalTime - Representa uma hora sem informação de data e nem de fuso horário.

- `OffsetTime` - Representa uma hora sem informação de data, mas com um fuso horário fixo (não leva em conta horário de verão).
- `LocalDateTime` - Representa uma data e hora, mas sem fuso horário.
- `ZonedDateTime` - Representa uma data e hora com fuso horário que leva em conta horário de verão.
- `OffsetDateTime` - Representa uma data e hora com fuso horário fixo (não leva em conta horário de verão).
- `YearMonth` - Representa uma data contendo apenas um mês e um ano.
- `Year` - Representa uma data correspondendo apenas a um ano.
- `Instant` - Representa um ponto no tempo, com precisão de nanossegundos.

Todas elas são implementações da interface `Temporal`, que especifica o comportamento comum a todas elas. E observe que a API delas é bem mais fácil de se usar do que `Date` ou `Calendar`, tem um monte de métodos para se somar datas, verificar quem está antes ou depois, extrair determinados campo (dia, mês, hora, segundo, etc), converter entre um tipo e outro, etc.
Também há implementações de `Temporal` mais específicas para diferentes calendários. A saber: `JapaneseDate`, `ThaiBuddhistDate`, `HijrahDate` e `MinguoDate`. Eles são análogos ao `LocalDate`, mas em calendários específicos, e portanto não têm informação de hora ou fuso horário.
Nota-se também que todas elas tem um método estático `now()` que constrói o objeto da classe correspondente de acordo com a hora do sistema. Por exemplo:

```
LocalDate hoje = LocalDate.now();
LocalDateTime horaRelogio = LocalDateTime.now();
Instant agora = Instant.now();
```

Fusos horários são representados pela classe `ZoneId`. Uma instância que corresponde ao `ZoneId`da máquina local pode ser obtido com o método `ZoneId.systemDefault()`. Uma outra forma de obter instâncias de `ZoneId` é por meio do método `ZoneId.of(String)`. Por exemplo:

```
ZoneId fusoHorarioDaqui = ZoneId.systemDefault();
ZoneId utc = ZoneId.of("Z");
ZoneId utcMais3 = ZoneId.of("+03:00");
ZoneId fusoDeSaoPaulo = ZoneId.of("America/Sao_Paulo");
```

Observe-se que alguns fusos horários são fixos, ou seja não são afetados por regras de horário de verão, enquanto que outros, tal como `ZoneId.of("America/Sao_Paulo")`, são afetados por horário de verão.

Conversão entre `Date` e as novas classes

Para converter um `Date` para uma instância de uma das classes do pacote `java.time`, podemos fazer assim:

```
Date d = ...;
Instant i = d.toInstant();
ZonedDateTime zdt = i.atZone(ZoneId.systemDefault());
OffsetDateTime odt = zdt.toOffsetDateTime();
LocalDateTime ldt = zdt.toLocalDateTime();
```

```
LocalTime lt = zdt.toLocalTime();
LocalDate ld = zdt.toLocalDate();
```

No código acima, o fuso horário usado é importante. Normalmente você irá usar o `ZoneId.systemDefault()` ou o `ZoneId.of("Z")`, dependendo do que você estiver fazendo. Em alguns casos, você pode querer usar algum outro fuso horário diferente. Se você quiser armazenar o fuso horário em alguma variável (possivelmente estática) e sempre (re)utilizar depois, não há problema (inclusive é recomendado em muitos casos).

Obviamente, há várias outras formas de se obter instâncias das classes definidas acima.

Para converter de volta para `Date`:

```
ZonedDateTime zdt = ...;
Instant i2 = zdt.toInstant();
Date.from(i2);
```

Parse e formatação com `String`

Para converter qualquer um deles para `String`, você usa a classe java.time.format.DateTimeFormatter. Ela é a substituta do `SimpleDateFormat`. Por exemplo:

```
DateTimeFormatter fmt = DateTimeFormatter
            .ofPattern("dd/MM/uuuu")
            .withResolverStyle(ResolverStyle.STRICT);
LocalDate ld = ...;
String formatado = ld.format(fmt);
```

O processo inverso é feito com os métodos estáticos `parse(String, DateTimeFormatter)` que cada uma dessas classes tem. Por exemplo:

```
DateTimeFormatter fmt = ...;
String texto = ...;
LocalDate ld = LocalDate.parse(texto, fmt);
```

Um detalhe a se atentar é o uso de `uuuu` ao invés de `yyyy` no método `ofPattern`. O motivo disto é que `yyyy` não funciona em caso de datas antes de Cristo. Raramente isso iria importar, mas onde isso não importa, os dois funcionam iguais, e onde isso importa, o `uuuu` deve ser usado. Assim sendo, não tem muito sentido em usar o `yyyy` em detrimento do `uuuu`. Mais detalhes nesta resposta (em inglês).

Um outro detalhe é o withResolverStyle(ResolverStyle) que diz o que fazer com datas mal-formadas. Há três possibilidades: STRICT, SMART e LENIENT. O `STRICT` não permite nada que não esteja rigorosamente no padrão. O modo `LENIENT` permite que ele interprete `31/06/2017`como `01/07/2017`, por exemplo. O `SMART` tenta adivinhar qual é a melhor forma, interpretando `31/06/2017` como `30/06/2017`. O padrão é o `SMART`, mas recomendo usar o `STRICT` sempre, pois ele não tolera datas mal-formadas e não tenta adivinhar o que uma data mal-formada poderia ser. Veja alguns testes acerca disso no ideone.

Conversão de Calendar

A classe legada `GregorianCalendar` é para todos os efeitos equivalente a nova classe `ZonedDateTime`. Os métodos `GregorianCalendar.from(ZonedDateTime)` e `GregorianCalendar.toZonedDateTime()` servem para fazer a conversão direta:

```
ZonedDateTime zdt1 = ...;
GregorianCalendar gc = GregorianCalendar.from(zdt1);
ZonedDateTime zdt2 = gc.toZonedDateTime();
```

Tendo então a conversão de `Calendar` para `ZonedDateTime` e vice-versa, use os métods já descritos acima caso queira obter algum dos objetos da nova API tal como `LocalDate` ou `LocalTime`.

Se o que você tiver for uma instância de `Calendar` ao invés de `GregorianCalendar`, quase sempre poderá fazer um *cast* para `GregorianCalendar` para usar o método `toZonedDateTime()`. Caso não queira usar o *cast*, é possível converter-se o `Calendar` para `Instant`:

```
Calendar c2 = ...;
Date d2 = c2.getTime();
Instant i2 = d2.toInstant();
```

Também é possível construir-se um `Calendar` a partir de um `Instant` usando o `Date` como intermediário:

```
Instant i = ...;
Date d1 = Date.from(i);
Calendar c = new GregorianCalendar();
c.setTime(d1);
```

Sobre o Joda-Time

Quanto ao Joda-Time, ela é uma API que foi desenvolvida por alguns anos exatamente com a intenção de substituir o `Date` e o `Calendar`. E ela conseguiu! O pacote `java.time` e todas as classes de lá são fortemente inspiradas no Joda-Time, embora existam algumas diferenças importantes que têm como objetivo não repetir alguns dos erros do Joda-Time.

compartilharmelhorar esta resposta
editada 22/08/18 às 19:24
respondida 12/01/17 às 21:45
Victor Stafusa
55,6mil886151

- +1 é muito bom ter toda essa informação sobre o pacote time num lugar só, facilita quando surgir uma duvida. Falando em duvida, não entendi muito bem o que quis dizer com "fuso horario fixo" do offsettime e offsetdatetime. – user28595 12/01/17 às 21:55
- 1
  @diegofm Resposta editada. – Victor Stafusa 13/01/17 às 2:19
- 1
  Ótima resposta, porém o `java.time` **não** é o "*Joda-Time com nome de pacote renomeado*". Apesar de ter várias classes com os mesmos nomes, na verdade há várias diferenças entre as 2 APIs, algumas inclusive conceituais, e há um bom resumo feito pelo autor em seu blog: blog.joda.org/2009/11/why-jsr-310-isn-joda-time_4941.html - destaque para o final "*I took the decision that I didn't want to add an API to the JDK that had known design flaws. And the changes required weren't just minor. As a result,*

JSR-310 ***started from scratch****, but with an API *'inspired by Joda-Time*".* – hkotsubo 7/05/18 às 18:23

- 1

@hkotsubo Curioso. Na época que escrevi essa resposta, vi essa informação em algum blog que vi em algum lugar. Entretanto, seja como for, reescrevi esse último pedaço. Obrigado. – Victor Stafusa 7/05/18 às 19:46

- 1

De fato, na época que a JSR310 começou a ganhar mais destaque, eu também achei que as APIs seriam iguais, mas o autor aproveitou a oportunidade para melhorar e corrigir pontos que ele não achava ideais, o que na minha opinião foi ótimo. Neste artigo ele lista as principais diferenças entre as APIs: blog.joda.org/2014/11/... – hkotsubo 8/05/18 às 13:22

comentar

10

+150

Complementando a resposta do Victor, seguem mais alguns pontos a se atentar quando for migrar de uma API para outra. No texto abaixo às vezes me refiro ao `java.time` como "API nova" (apesar de ter sido lançado em 2014) e a `Date`, `Calendar` e demais classes como "API legada" (pois é esse o termo usado no tutorial da Oracle).

Precisão

`java.util.Date` e `java.util.Calendar` possuem precisão de milissegundos (3 casas decimais na fração de segundos), enquanto as classes do pacote `java.time` possuem precisão de nanossegundos (9 casas decimais).

Isso significa que a conversão da API nova para a API legada implica em perda de precisão. Ex:

```
// Instant com 9 casas decimais (123456789 nanossegundos)
Instant instant = Instant.parse("2019-03-21T10:20:40.123456789Z");
// converte para Date (mantém apenas 3 casas decimais)
Date date = Date.from(instant);
System.out.println(date.getTime()); // 1553163640123

// ao converter de volta para Instant, as casas decimais originais são perdidas
Instant instant2 = date.toInstant();
System.out.println(instant2); // 2019-03-21T10:20:40.123Z
System.out.println(instant2.getNano()); // 123000000
```

Ao converter de `java.time.Instant` para `java.util.Date`, somente as 3 primeiras casas decimais são mantidas (as demais são simplesmente descartadas). Por isso, ao converter este `Date` de volta para `Instant`, ele não tem mais essas casas decimais. Mas repare que no final, `getNano()` retorna `123000000`. Mesmo que o `Date` só tenha precisão de milissegundos, internamente um `Instant` sempre guarda o valor em nanossegundos.

Se quiser restaurar o valor original das frações de segundo, este deve ser guardado separadamente. Para restaurá-lo, basta usar um `java.time.temporal.ChronoField`:

```
// Instant com 9 casas decimais (123456789 nanossegundos)
Instant instant = Instant.parse("2019-03-21T10:20:40.123456789Z");
// converte para Date (mantém apenas 3 casas decimais)
```

```
Date date = Date.from(instant);
// guardar o valor da fração de segundos
int nano = instant.getNano();

.....
// converter de volta para Instant e restaurar o valor dos nanossegundos
Instant instant2 = date.toInstant().with(ChronoField.NANO_OF_SECOND, nano);
System.out.println(instant2); // 2019-03-21T10:20:40.123456789Z
System.out.println(instant2.getNano()); // 123456789
```

Parsing com mais de 3 casas decimais

Esta limitação de 3 casas decimais também se aplica ao *parsing*. Por exemplo, se tentarmos fazer o *parsing* de uma `String` contendo 6 casas decimais na fração de segundos:

```
SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss.SSSSSS");
Date date = sdf.parse("2019-03-21T10:20:40.123456");
System.out.println(date); // Thu Mar 21 10:22:43 BRT 2019
```

Repare que na `String` o horário é "10:20:40", mas a saída foi "10:**22:43**". Isso acontece porque, segundo a documentação, a letra `S` corresponde aos milissegundos. Colocando 6 letras `S`, como fizemos, não faz com que o trecho `123456` seja interpretado como **microssegundos** (que é o que de fato esse valor representa). Em vez disso, o `SimpleDateFormat` interpreta como 123456 **milissegundos**, que por sua vez corresponde a "2 minutos, 3 segundos e 456 milissegundos" - e este valor é somado ao horário obtido. Por isso o resultado é 10:22:43 (*se esse algoritmo faz sentido ou não, é outra história, mas o fato é que `SimpleDateFormat` faz muitas outras coisas estranhas além dessa*).

No caso acima, ao imprimir o `Date`, internamente é chamado o seu método `toString()`, que omite os milissegundos. Então vamos usar o mesmo `SimpleDateFormat` acima para tentar imprimir os milissegundos:

```
SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss.SSSSSS");
Date date = sdf.parse("2019-03-21T10:20:40.123456");
System.out.println(sdf.format(date)); // 2019-03-21T10:22:43.000456
```

Repare que o resultado tem `.000456` (ou seja, 456 **microssegundos**), sendo que na verdade `456`é o valor dos milissegundos (já que `Date` não tem precisão de microssegundos), então deveria ser mostrado como `.456` (ou `456000`, já que o formato indica 6 dígitos). Mas ao colocar 6 letras `S`, a documentação diz que valores numéricos são preenchidos com zeros à esquerda caso o valor tenha menos dígitos do que a quantidade de letras. Por isso que 456 foi mostrado como 000456.

Ou seja, se estiver lidando com mais de 3 casas decimais na fração de segundos, `Date`, `Calendar`e `SimpleDateFormat` simplesmente não funcionam. Uma maneira de resolver é simplesmente tratar as casas decimais separadamente, por exemplo:

```
String s = "2019-03-21T10:20:40.123456";
String[] partes = s.split("\\.");
SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss");
Date date = sdf.parse(partes[0]);
```

```
// completar com zeros à direita, para sempre ter o valor em nanossegundos
int nanossegundos = Integer.parseInt(String.format("%-9s", partes[1]).replaceAll(" ", "0"));
System.out.println(sdf.format(date)); // 2019-03-21T10:20:40
System.out.println(nanossegundos); // 123456000

// o Date foi gerado sem os milissegundos, já que o parsing foi feito sem eles
// se quiser ser preciso mesmo, devemos somar os milissegundos ao Date
date.setTime(date.getTime() + (nanossegundos / 1000000));
```

Já na API `java.time` é possível fazer o *parsing* das 6 casas decimais sem problemas:

```
DateTimeFormatter parser = DateTimeFormatter.ofPattern("uuuu-MM-dd'T'HH:mm:ss.SSSSSS");
LocalDateTime dt = LocalDateTime.parse("2019-03-21T10:20:40.123456", parser);
System.out.println(dt); // 2019-03-21T10:20:40.123456
```

Agora sim as frações de segundo foram interpretadas corretamente. Isso acontece porque na nova API a letra `S` significa "frações de segundo" (e não mais milissegundos), podendo interpretar até 9 casas decimais. Isso nos leva a outro ponto importante.

Formatação e *Parsing*

Como já vimos acima, o parâmetro que passamos para `SimpleDateFormat` (`yyyy-MM-dd'T'HH:mm:ss.SSSSSS`) não funciona exatamente da mesma maneira que no `java.time`:

- o `S` tem um significado e funcionamento ligeiramente diferente: na API legada dá resultados errados quando tem mais que 3 casas decimais
- no `java.time` usei `u` para o ano, em vez de `y` (e a resposta do Victor já explica muito bem a diferença)

Esse é um ponto importante: **só porque um formato funcionava com `SimpleDateFormat`, não quer dizer que vai funcionar do mesmo jeito com `DateTimeFormatter`**. A letra `u`, por exemplo, significa "ano" no `java.time`, mas na API legada significa "dia da semana". E existem letras novas que foram adicionadas no Java 8, como o `Q` para o trimestre, `e` para o "dia da semana localizado" (ou seja, baseado no `Locale`), entre outras. Sempre consulte a documentação para mais detalhes (e mesmo na API legada há algumas diferenças, como a letra `X`, que só foi adicionada no Java 7 - veja que na documentação do Java 6 ela não existe).

Além disso, há mais opções para formatação e *parsing*. Por exemplo, o formato que estamos usando nos exemplos acima (que é definido pela norma ISO 8601) pode ser interpretado diretamente:

```
LocalDateTime dt = LocalDateTime.parse("2019-03-21T10:20:40.123456");
```

Internamente este método usa a constante predefinida `DateTimeFormatter.ISO_LOCAL_DATE_TIME`. A diferença para o exemplo anterior é que usando `.SSSSSS`, ele só consegue interpretar `Strings`que tenham exatamente 6 casas decimais. Já o `ISO_LOCAL_DATE_TIME` é mais flexível, pois permite de zero a 9 casas decimais.

Podemos simular este comportamento (ter um campo com uma quantidade variável de dígitos), usando um `java.time.format.DateTimeFormatterBuilder`:

```
DateTimeFormatter parser = new DateTimeFormatterBuilder()
```

```
        .appendPattern("uuuu-MM-dd'T'HH:mm:ss")
        .optionalStart() // frações de segundo opcionais
        .appendFraction(ChronoField.NANO_OF_SECOND, 0, 9, true) // de 0 a 9 dígitos
        .toFormatter();

LocalDateTime dt = LocalDateTime.parse("2019-03-21T10:20:40", parser);
System.out.println(dt); // 2019-03-21T10:20:40

LocalDate date = LocalDate.parse("2019-03-21T10:20:40.123456789", parser);
System.out.println(date); // 2019-03-21
```

Repare no exemplo acima o trecho final com `LocalDate`. Esta classe só possui o dia, mês e ano, mas para obtê-la a partir de uma `String` que contém data e hora, eu tive que usar o mesmo *parser*. Isso acontece porque o *parser* deve ser capaz de interpretar a `String` inteira, mesmo que depois alguns campos não sejam usados. Ou seja, o `parser` interpreta a `String` e o `LocalDate` pega só o que precisa (dia, mês e ano), descartando o restante.

Veja também que o *parser* é capaz de interpretar tanto `Strings` sem fração de segundos quanto com 9 casas decimais. `DateTimeFormatterBuilder` possui muitas opções que não são possíveis de se fazer com `SimpleDateFormat`, por isso a migração de uma para outra não é tão direta (não basta copiar e colar o mesmo formato e achar que tudo funcionará da mesma maneira, e a nova API ainda te dá mais opções e alternativas melhores para obter os mesmos resultados).

Modos de *parsing*

Detalhando um pouco mais os modos de *parsing* (que a resposta do Victor menciona), no `java.time` existem três:

O modo `LENIENT` permite datas inválidas e faz ajustes automáticos. Por exemplo, `31/06/2017` é ajustado para `01/07/2017`. Além disso, este modo aceita valores fora dos limites definidos para cada campo, como o dia 32, mês 15, etc. Por exemplo, `32/15/2017` é ajustado para `01/04/2018`.

O modo `SMART` também faz alguns ajustes quando a data é inválida, então `31/06/2017` é interpretado como `30/06/2017`. A diferença para `LENIENT` é que este modo não aceita valores fora dos limites dos campos (mês 15, dia 32, etc), então `32/15/2017` dá erro (lança um `DateTimeParseException`). É o modo *default* quando você cria um `DateTimeFormatter`.

O modo `STRICT` é o mais restrito: não aceita valores fora dos limites e nem faz ajustes quando a data é inválida, portanto `31/06/2017` e `32/15/2017` dão erro (lançam um `DateTimeParseException`).

Já `SimpleDateFormat` só possui dois modos: leniente e não-leniente (que pode ser configurado usando-se o método `setLenient`). O *default* é ser leniente, o que causa os comportamentos "estranhos" já citados (como a bagunça que é feita no *parsing* de 6 casas decimais nas frações de segundo, que poderia ser evitado setando-o para não-leniente).

Datas e timezones

Um `Date`, apesar do nome, não representa uma data - no sentido de representar **apenas um único valor** de dia, mês, ano, hora, minuto e segundo. Na verdade esta classe representa um instante, um ponto na linha do tempo. O único valor que ela guarda é um `long` contendo o timestamp: a quantidade de milissegundos desde o Unix Epoch (que por sua vez é "1 de janeiro de 1970, à meia noite em UTC").

O detalhe do timestamp é que ele é o mesmo no mundo todo, mas a data e hora correspondente pode mudar de acordo com o lugar em que você está. Por exemplo, o timestamp `1553163640000`corresponde a:

- 21 de março de 2019 às 07:20:40 em São Paulo
- 21 de março de 2019 às **11**:20:40 em Berlim
- **22** de março de 2019 às **00**:20:40 em Samoa

Em todos estes lugares, o timestamp é o mesmo: qualquer computador, em qualquer parte do mundo, que rodasse `System.currentTimeMillis()` (ou qualquer outro código que obtém o timestamp atual) naquele exato instante obteria o mesmo resultado (1553163640000). Porém, a data e hora correspondente a este timestamp são diferentes, conforme o timezone sendo utilizado.

`Date` representa o timestamp, não as datas e horas correspondentes a um timezone. O problema é que quando você imprime um `Date`, ele usa o timezone *default* que está setado na JVM para saber qual data e hora exibir:

```
// Date correspondente ao timestamp 1553163640000
Date date = new Date(1553163640000L);
TimeZone.setDefault(TimeZone.getTimeZone("America/Sao_Paulo"));
System.out.println(date.getTime() + " - " + date);
TimeZone.setDefault(TimeZone.getTimeZone("Europe/Berlin"));
System.out.println(date.getTime() + " - " + date);
TimeZone.setDefault(TimeZone.getTimeZone("Pacific/Samoa"));
System.out.println(date.getTime() + " - " + date);
```

Eu uso `TimeZone.setDefault` para mudar o timezone *default* da JVM, e em seguida uso `getTime()` para mostrar o valor do timestamp e também imprimo o próprio `Date`. A saída é:

```
1553163640000 - Thu Mar 21 07:20:40 BRT 2019
1553163640000 - Thu Mar 21 11:20:40 CET 2019
1553163640000 - Wed Mar 20 23:20:40 SST 2019
```

Repare que o valor do timestamp não mudou, mas os valores de data e hora foram ajustados para o timezone *default* que está setado no momento. Mas não se engane: estes valores de data, hora e timezone são apenas uma representação da data, mas o `Date` em si **não possui** esses valores (o único valor que ele possui é o timestamp). Outra concepção errada é achar que o `Date` "está em um timezone", mas ele não tem nenhuma informação sobre isso. Quando a data é impressa, o timezone *default* é usado somente para converter o timestamp para uma data e hora. Mas o `Date`em si não está naquele timezone.

Dito isso, é preciso atenção para converter `Date` de/para o `java.time`. A única conversão direta que não envolve timezones é entre `Date` e `Instant`, já que ambos representam o mesmo conceito: as duas classes só possuem o valor do timestamp (a diferença, claro, é a precisão, já explicada acima).
Já a conversão para as demais classes sempre exigirá um timezone. É claro que você pode usar o timezone *default* se quiser:

```
// Date correspondente ao timestamp 1553163640000
Date date = new Date(1553163640000L);
// usar timezone default
ZonedDateTime zdt = date.toInstant().atZone(ZoneId.systemDefault());
// converte para LocalDate
LocalDate dt = zdt.toLocalDate();
```

Mas é importante lembrar que qualquer aplicação pode rodar `TimeZone.setDefault` e mudar o timezone *default*, afetando todas as aplicações que estiverem rodando na mesma JVM. Se quiser usar um timezone específico, seja explícito na conversão:

```
// usar timezone específico
ZonedDateTime zdt = date.toInstant().atZone(ZoneId.of("America/Sao_Paulo"));
```

Você pode obter a lista de timezones disponíveis usando o método `getAvailableZoneIds()`. A lista pode variar porque essa informação fica embutida na JVM, mas ela pode ser atualizada sem precisar mudar a versão do Java. A atualização é importante pois a IANA (órgão responsável por manter o banco de informações de timezones que o Java e muitas outras linguagens, sistemas e aplicações usam) sempre está lançando novas versões. Isso acontece porque as regras dos fusos horários são definidas por governos e mudam o tempo todo.
Muitas linguagens e APIs possuem métodos/funções para converter uma data (somente dia, mês e ano) para um timestamp e vice-versa, mas no fundo elas estão apenas usando algum horário e timezone arbitrários (geralmente usam "meia-noite" no timezone *default* da sua respectiva configuração) e "escondendo esta complexidade" de você (algumas até permitem que se mude o timezone, mas nem sempre é algo trivial, enquanto outras nem permitem tal alteração).
O `java.time`, por sua vez, é mais explícito e exige que você sempre indique algum timezone. Por um lado pode parecer uma "burocracia" desnecessária, mas por outro lado permite que você use timezones diferentes, garantindo mais flexibilidade, controle e resultados mais corretos. Esconder esta complexidade tornaria a API mais "simples", por outro lado passaria a ideia errada (que muitas APIs passam) de que uma data (somente dia, mês e ano) pode ser "magicamente" convertida para um timestamp (sendo que, sem saber o horário e o timezone, tal conversão não é possível).

Um detalhe importante é que a classe `TimeZone` não valida o nome do timezone:

```
System.out.println(TimeZone.getTimeZone("nome que não existe"));
```

Quando o nome não existe, é retornado uma instância que corresponde a UTC:

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

Repare que o offset é zero e não há horário de verão (`dstSavings=0`). Ou seja, é o mesmo que UTC. Por isso, erros de digitação podem passar batidos e só será percebido quando começar a aparecer datas erradas. Já `ZoneId` não aceita nomes que não existem:

```
ZoneId.of("nome que não existe");
```

Este código lança uma exceção:

> java.time.DateTimeException: Invalid ID for region-based ZoneId, invalid format: nome que não existe

Outro detalhe é que `TimeZone` aceita abreviações:

```
System.out.println(TimeZone.getTimeZone("IST"));
```

O problema é que abreviações são ambíguas e não representam timezones de fato (veja mais detalhes na wiki da tag timezone, na seção "Abreviações"). "IST", por exemplo, é usada na Índia, Irlanda e Israel, então qual desses é retornado?

> sun.util.calendar.ZoneInfo[id="IST",offset=19800000,dstSavings=0,useDaylight=false,transitions=7,lastRule=null]

Nesse caso o offset é 19800000 milissegundos, que corresponde a 5 horas e meia. Portanto, corresponde ao timezone da Índia (pois atualmente eles usam o offset `+05:30`).

`ZoneId`, por sua vez, não aceita abreviações, então `ZoneId.of("IST")` lança um `java.time.zone.ZoneRulesException: Unknown time-zone ID: IST`.

Esses detalhes são importantes na hora de migrar de uma API para outra, pois não basta passar os mesmos nomes/abreviações como parâmetro. Caso o código use abreviações, você terá que tomar uma decisão quanto às mesmas e usar um nome de timezone específico (`Asia/Kolkata` para Índia, `Asia/Jerusalem` para Israel ou `Europe/Dublin` para Irlanda, por exemplo).

---

java.sql

As classes do pacote `java.sql` (Date, Time e Timestamp) herdam de `java.util.Date`, e por isso também possuem sua principal característica: não representam um único valor de data e hora, e sim um timestamp. Por isso elas também são afetadas pelo timezone *default* da JVM:

```
TimeZone.setDefault(TimeZone.getTimeZone("America/Sao_Paulo"));
LocalDate date = LocalDate.of(2018, 1, 1); // 1 de janeiro de 2018
java.sql.Date sqlDate = java.sql.Date.valueOf(date);
System.out.println("LocalDate=" + date + ", sqlDate=" + sqlDate);

// mudar o timezone default
TimeZone.setDefault(TimeZone.getTimeZone("America/Los_Angeles"));
System.out.println("LocalDate=" + date + ", sqlDate=" + sqlDate);
```

A saída é:

> LocalDate=2018-01-01, sqlDate=2018-01-01
> LocalDate=2018-01-01, sqlDate=2017-12-31

Repare que depois que mudei o timezone *default* o valor do `sqlDate` aparentemente mudou.

Isso acontece porque `java.sql.Date.valueOf` pega o dia, mês e ano do `LocalDate`, junta com "meia-noite no timezone *default* da JVM" e obtém o timestamp correspondente. No exemplo acima, o timezone *default* é `America/Sao_Paulo`, então o timestamp (obtido com `sqlDate.getTime()`) é `1514772000000`, que de fato corresponde a meia-noite do dia 01/01/2018 em São Paulo. Só que esse mesmo timestamp corresponde a 31/12/2018 às 18h em Los Angeles. Por isso que ao mudar o timezone *default* para `America/Los_Angeles` o `sqlDate` é mostrado com o valor "errado".
É o mesmo que ocorre com `java.util.Date`: o valor interno do timestamp não muda, mas ao imprimir a data, o método `toString()` usa o timezone *default* para saber quais os valores de data/hora serão mostrados.
As classes `java.sql.Time` e `java.sql.Timestamp` também sofrem desses mesmos problemas, pois ambas são subclasses de `java.util.Date`.

O método `valueOf` também é afetado pelo timezone *default*:

```
TimeZone.setDefault(TimeZone.getTimeZone("America/Sao_Paulo"));
LocalDate date = LocalDate.of(2018, 1, 1); // 1 de janeiro de 2018
java.sql.Date sqlDate = java.sql.Date.valueOf(date);
System.out.println("LocalDate=" + date + ", sqlDate=" + sqlDate);
System.out.println(sqlDate.getTime());

TimeZone.setDefault(TimeZone.getTimeZone("America/Los_Angeles"));
sqlDate = java.sql.Date.valueOf(date); // recriar o sqlDate, com o mesmo LocalDate
System.out.println("LocalDate=" + date + ", sqlDate=" + sqlDate);
System.out.println(sqlDate.getTime());
```

Repare que agora estou recriando o `sqlDate` com `valueOf`, com um timezone *default* diferente. Agora a saída é:

```
LocalDate=2018-01-01, sqlDate=2018-01-01
1514772000000
LocalDate=2018-01-01, sqlDate=2018-01-01
1514793600000
```

A data agora parece "correta", mas repare que o timestamp criado foi diferente. Isso acontece porque o método `valueOf` sempre usa meia-noite no timezone *default* que está setado no momento em que ele é chamado. Se qualquer outra aplicação rodando na mesma JVM chamar `TimeZone.setDefault`, ou se alguém desconfigurar o fuso horário da JVM ou do servidor, este código será afetado.
Mas veja que o `LocalDate` sempre mantém o mesmo valor, pois esta classe possui apenas os valores numéricos do dia, mês e ano, sem qualquer informação sobre horários ou timezones. Por isso, seu valor permanece inalterado, independente de qual for o timezone *default*.

Caso o banco de dados que você está usando tenha um driver compatível com o **JDBC 4.2**, é possível trabalhar diretamente com as classes do `java.time`, usando os métodos `setObject` da classe `java.sql.PreparedStatement` e `getObject` da classe `java.sql.ResultSet`. Um exemplo com `Instant` seria:

```
Instant instant = ...
PreparedStatement ps = ...
// seta o java.time.Instant
ps.setObject(1, instant);

// obter o Instant do banco
ResultSet rs = ...
Instant instant = rs.getObject(1, Instant.class);
// converter o instant para um timezone
ZonedDateTime zdt = instant.atZone(ZoneId.of("America/Sao_Paulo"));
...
```

Só lembrando que nem todos os bancos de dados suportam todos os tipos do `java.time`. Consulte a documentação e veja quais classes são mapeadas para quais tipos no banco de dados.

Alternativas para Java < 8

Para o Java 6 e 7 existe o ThreeTen Backport, um excelente *backport* do `java.time`, criado por Stephen Colebourne (o mesmo criador da API `java.time`, inclusive).

A maioria das funcionalidades do Java 8 está presente, com algumas diferenças:

- em vez de estarem no pacote `java.time`, as classes ficam no pacote `org.threeten.bp`
- métodos de conversão como `Date.toInstant()` e `Date.from(Instant)` só existem no Java >= 8, mas o *backport* possui a classe `org.threeten.bp.DateTimeUtils` para fazer essas conversões. Exemplos:

```
// Java >= 8, java.util.Date de/para java.time.Instant
Date date = new Date();
Instant instant = date.toInstant();
date = Date.from(instant);

// Java 6 e 7 (ThreeTen Backport), java.util.Date de/para org.threeten.bp.Instant
Date date = new Date();
Instant instant = DateTimeUtils.toInstant(date);
date = DateTimeUtils.toDate(instant);
```

A classe `DateTimeUtils` também possui métodos de conversão entre `java.sql.Date` e `java.time.LocalDate`, `java.util.TimeZone` para `java.time.ZoneId`, etc.

Basicamente, existe um equivalente para cada método de conversão que foi adicionado no Java 8. Consulte a documentação para mais detalhes.

Outra diferença é que no Java 8 pode-se usar a sintaxe de *method reference*, enquanto no *backport*foram criadas constantes que simulam este comportamento (já que o *method reference* não existe no Java <= 7):

```
// Java >= 8, usando method reference (LocalDate::from)
DateTimeFormatter parser = DateTimeFormatter.ofPattern("dd/MM/uuuu");
LocalDate date = parser.parse("20/10/2019", LocalDate::from);

// Java 6 e 7 (ThreeTen Backport), usando LocalDate.FROM para simular o method reference LocalDate::from
DateTimeFormatter parser = DateTimeFormatter.ofPattern("dd/MM/uuuu");
```

```
LocalDate date = parser.parse("20/10/2019", LocalDate.FROM);
```

Para Android, o `java.time` só está disponível a partir da API level 26 (necessário minSdkVersion>=26, não basta ter compileSdkVersion>=26), mas é possível usar o ThreeTen Backport, seguindo as configurações descritas neste link.

E para Java 5, existe o "antecessor do `java.time`" (também criado por Stephen Colebourne): o Joda-Time. Apesar de ser um projeto encerrado (em seu próprio site há um aviso sobre isso), se você ainda está preso ao Java 5 e quiser usar algo melhor do que `Date` e `Calendar`, é uma boa alternativa.

O Joda-Time não é 100% idêntico ao `java.time`, mas muitos dos seus conceitos e ideias foram aproveitados no Java 8 (inclusive algumas classes e métodos possuem os mesmos nomes). As principais semelhanças e diferenças entre as APIs são explicadas aqui e aqui.