

Novos tempos: javax.time

Conheça a nova especificação de Data e Hora

Você irá conhecer a nova API de data e hora da plataforma Java. Construída como uma opção ao uso de `java.util.Date` e `java.util.Calendar`, vem com duas premissas: facilidade de uso e correção de bugs e falhas arquiteturais de suas antecessoras.

DANIEL CICERO AMADEI E MICHAEL NASCIMENTO SANTOS

Neste artigo conheceremos a nova API de data e hora que está sendo elaborada para a plataforma Java SE 7. O foco da nova API é resolver vários problemas que afetam os desenvolvedores Java há anos, presentes nas classes `java.util.Date` e `java.util.Calendar`.

Por que uma nova API?

A arquitetura elaborada para as classes `Date` e `Calendar`, ambas do pacote `java.util` é bastante questionável. Veremos algumas dessas decisões arquiteturais dúbias, além de “bugs” da API atual, no decorrer do artigo.

Outro grande problema é a falta de tipos para representar unidades comuns no dia-a-dia de qualquer desenvolvedor como, por exemplo, períodos, instantes, durações, entre outros. Sem classes que representem essas unidades do “mundo real”, é necessária a criação de soluções paliativas por conta e risco dos desenvolvedores. Isso resulta em mais codificação e código legado para manutenção.

Esses pontos claramente abrem espaço a uma nova abordagem para o tratamento de datas e horas dentro da plataforma Java.

Alguns dos problemas apresentados por `Date` e `Calendar`

Um dos principais problemas das classes atuais é a **inconsistência**. A classe `java.util.Date` não representa uma data, mas sim um instante na *linha do tempo*. Desta forma, não temos como representar somente uma data ou somente o tempo. Outro exemplo de inconsistência é o fato da classe `Date` utilizar anos com base em 1900, enquanto a `Calendar` requerer os anos completos, incluindo o século. O código apresentado a seguir, apesar do uso de um construtor descontinuado (*deprecated*), ilustra a criação de uma data cujo ano será 3909, ao contrário do que parece ao ler o código.

```
Date date = new Date(2009, 2, 1);
```

O início dos meses em 0 (Zero) na classe `Calendar` também é algo que tem atrapalhado a vida dos desenvolvedores Java de forma considerável. Por exemplo, para criar uma instância de `Calendar` que represente a data 01/01/2009, temos que escrever o seguinte código:

```
Calendar calendar = new GregorianCalendar(2009, 0, 1);
```

Além disso, temos uma série de outros problemas arquiteturais ou de implementação:

- Os objetos são mutáveis, inseguros em ambiente multi-thread, requerendo sincronização;
- Não há como formatar uma instância de `Calendar` usando as classes do pacote `java.text`;
- Não existe opção para representar períodos ou durações de forma padronizada
- [As APIs atuais exigem muito código para tarefas rotineiras de manipulação de datas e horas. Sem mencionar ainda que muitas dessas “tarefas rotineiras” constituem casos de uso comuns e poderiam ser implementados pelas próprias APIs.-](#)

Esses são apenas alguns dos problemas que a JSR 310¹ irá resolver. Os já velhos conhecidos problemas devido às alterações no horário de verão também estão sendo endereçados. Além de tudo isso, você verá que a API é muito mais intuitiva e irá lhe poupar muitas e muitas linhas de codificação.

Princípios de Design da JSR 310

A JSR 310 tem sido elaborada com alguns princípios para facilitar seu uso e entendimento e tornar seu código mais robusto e de fácil leitura.

Baseada em padrões

A nova API está sendo construída totalmente alinhada com o ISO-8601, um padrão internacional para representação de datas e horas.

Imutável

Os principais objetos da nova API serão imutáveis. Não será possível alterá-los após a construção. Também serão *thread-safe*, podendo até mesmo ser *singletons*.

A seguir, um primeiro exemplo, da criação de uma data (sem a hora!):

```
LocalDate date = LocalDate.date(2009, 1, 1);
```

Nesse caso, criamos uma data com a classe `javax.time.calendar.LocalDate`. Essa classe representa uma data sem fuso horário (time zone²) no padrão ISO-8601, resultando em 2009-01-01. Caso desejemos alterar o ano dessa data, por exemplo, para 2010 não iremos utilizar um método `set()`. Neste caso, devemos utilizar um método `with()`, que cria uma nova instância da classe com o ano desejado. O código ficaria conforme apresentado a seguir:

```
date = date.withYear(2010);
```

Repare na atribuição ao objeto `date` com o resultado da invocação do método. Isso é necessário, pois é criado um novo objeto, mantendo o original inalterado, pois os objetos envolvidos são imutáveis.

Interfaces Fluêntes

As classes e interfaces seguem o padrão de fluência para permitir uma melhor legibilidade do código. Isso transforma as invocações a mais do que um método em sentenças de fácil leitura. O código apresentado a seguir, além do ano, altera o mês e o dia.

```
date = date.withYear(2010).withMonthOfYear(3).withDayOfMonth(2);
```

Clareza

Os métodos são bem definidos e indicam claramente seu propósito. Por exemplo, para subtrair dois anos de uma data qualquer, utilizamos o seguinte código:

```
date = date.withYear(2010).withMonthOfYear(3).withDayOfMonth(2);
date = date.minusYears(2);
```

Isso irá resultar na data 2008-03-02. Muito mais claro do que o método `roll(int field, int amount)` da classe `Calendar`.

¹ A nova especificação para representação de datas e horas dentro da plataforma Java, denominada `javax.time` neste artigo, está sendo criada pela JSR 310. O link para acesso à JSR pode ser obtido na seção de links.

² Utilizaremos os termos em inglês para a zona de fuso horário, *time zone* e fuso horário como *offset*, como veremos no decorrer de todo o artigo para que não precisemos usar os termos em português, muito mais incomuns até mesmo aqui no Brasil.

Extensível

Através de pontos de extensão, utilizando conceitos do design pattern *Strategy*, é possível controlar e customizar como a API se comporta em relação às datas e horas. Mesmo assim, não é necessário ser um especialista para usar a API. São fornecidas implementações padrão para a maioria dos cenários.

Um exemplo desses “ganchos” para extensibilidade é a interface **javax.time.calendar.DateAdjuster**, capaz de realizar ajustes em uma data de forma padronizada. Você simplesmente implementa essa interface e a utiliza para ajustar suas datas de forma padronizada e bem definida.

Além disso, a **javax.time** traz consigo uma classe utilitária, a **javax.time.calendar.DateAdjusters**, contendo implementações dos casos de uso mais comuns para o ajuste de datas. Essas implementações devem satisfazer a maioria das necessidades de desenvolvimento. O código apresentado a seguir ilustra o ajuste sendo realizado em uma data:

```
LocalDate date = Clock.systemDefaultZone().today();
date = date.with(DateAdjusters.next(DayOfWeek.MONDAY));
```

O código que apresentamos, além do ajuste da data, apresenta uma nova classe: **javax.time.calendar.Clock**. Essa classe é um *façade* para o acesso à data e hora correntes. O método **today()**, que invocamos na classe **Clock**, retorna a data corrente como uma instância de **LocalDate**.

Voltando a falar sobre o ajuste da data, ele ocorre na invocação ao método **with()**. Informamos **DateAdjusters.next(DayOfWeek.MONDAY)** como parâmetro do método. O objeto retornado pelo método **next()** é responsável por ajustar a data para a segunda-feira subsequente à data informada. Avaliando as datas, caso executemos tal código no dia 05/02/2009, após o ajuste, a nova data (lembre-se da imutabilidade!) será dia 09/02/2009.

Ainda no âmbito dos *Adjusters*, um outro exemplo muito interessante de sua aplicabilidade seria o ajuste das datas para dias úteis. Você pode, ainda, implementar seu próprio *adjuster* – digamos, para saltar feriados obtidos de um cadastro de feriados do seu sistema.

Duas escalas para lidar com tempo

A **javax.time** apresenta duas formas distintas para lidar com o tempo. Uma escala voltada para máquinas, denominada **Continuous** e a outra com foco em datas para seres humanos, denominada **Human**.

Continuous

Essa abordagem da **javax.time** é voltada para máquinas e representa o tempo na forma de um número incremental, sem muito significado para seres humanos, porém com grande valor para uso em processamentos que requerem cálculos envolvendo *timestamps*.

Instant

A classe **javax.time.Instant** representa um ponto instantâneo na linha do tempo, um instante de tempo, conhecido também como *timestamp*. Possui precisão de nanosegundos e 96 bits para armazenamento. Através dessa abordagem, é possível armazenar até algumas centenas de vezes uma data equivalente ao tempo de existência do universo. (Para os curiosos, o universo tem em torno de 13,8 bilhões de anos.) A seguir, um exemplo onde validamos se um instante é superior a outro utilizando o método **isAfter()**.

```
Instant instante1 = Clock.systemDefaultZone().instant();
//qualquer código aqui
Instant instante2 = Clock.systemDefaultZone().instant();
boolean avaliacao = instante1.isAfter(instante2);
```

Duration

A classe **javax.time.Duration** representa uma duração de tempo. Dentro da **javax.time**, ela representa a duração entre dois instantes. [O instante inicial que forma uma instância de Duration é inclusivo e o final é exclusivo](#). Apesar de armazenar dados provenientes de lá, a classe **Duration** é desconectada e independente da linha do tempo. O código apresentado a seguir ilustra o uso da classe **Duration**:

```
Instant agora = Clock.systemDefaultZone().instant();
Instant umMinutoMais = agora.plusSeconds(60);

Duration duration = Duration.durationBetween(agora, umMinutoMais);
System.out.println(duration);
```

No trecho de código apresentado, vemos primeiramente a criação de dois instantes. O primeiro representa o momento exato da execução, enquanto que o segundo derivamos a partir do primeiro com a soma de 60 segundos. Repare no método **plusSeconds()**. Ele é conciso, bem definido e indica exatamente seu propósito e a unidade manipulada. Após a criação dos instantes, criamos uma duração entre eles, que deverá conter exatamente 60 segundos. Ao imprimir a duração no console, temos a impressão da duração utilizando o padrão ISO-8601.

InstantInterval

A classe **javax.time.InstantInterval** representa um intervalo de instantes na linha do tempo. A classe pode possuir intervalos inclusivos, exclusivos ou algum dos intervalos pode não estar associado.

O código apresentado³ a seguir ilustra a criação de um intervalo a partir de dois instantes e a posterior verificação se um terceiro intervalo está contido entre eles:

```
Instant agora = Clock.systemDefaultZone().instant();
Instant umMinutoMais = agora.plusSeconds(60);

InstantInterval intervalo = InstantInterval.intervalBetween(agora, umMinutoMais);
boolean contido = intervalo.contains(Clock.systemDefaultZone().instant());
```

No exemplo, por utilizarmos o método “padrão”, o intervalo inicial é inclusivo e o final é exclusivo. Existe outro método que recebe valores booleanos indicando se desejamos que cada um dos intervalos seja inclusivo (**true**) ou exclusivo (**false**). Além disso, podemos construir a instância de **InstantInterval** a partir dos métodos “builder”: **intervalFrom()** e **intervalTo()**.

Human

Essa abordagem da **javax.time** é voltada para seres humanos. Ela representa os valores de datas e horas utilizando campos com classes específicas para representar cada um dos dados do calendário: ano, mês, dia, hora, minuto e segundo. Além desses campos mais comuns, temos algumas outras classes ou enumerações para representar, por exemplo, o dia do ano, a semana do ano, os nanossegundos de cada segundo, entre outras.

Através da abordagem para seres humanos, temos formas (classes!) para representar datas e horas, datas sem hora, horas sem data, offsets e time zones.

Datas e Horas locais (sem time zone ou offset)

Os tipos mais simples presentes dentro da **javax.time** são os tipos chamados de “locais”. Esses tipos podem representar data ou hora de forma isolada ou as duas em conjunto. São chamadas de *locais* por não estarem associados a um offset ou time zone.

³ Até a finalização deste artigo, a classe **InstantInterval** ainda não estava finalizada e funcional na implementação de referência da **javax.time** e o código apresentado como exemplo ainda não estava funcionando corretamente.

LocalDate

A primeira classe que mereceria nossa atenção seria a **javax.time.calendar.LocalDate**. Como já estamos cansados de ver exemplos envolvendo essa classe, pois é a que estamos acompanhando no decorrer do artigo, veremos algo a mais.

A representação dos campos de datas e horas dentro das classes é efetuada através de classes com este propósito. Com isso, essas classes contêm métodos implementando ações comuns, requeridas de cada um desses campos que compõem nossas datas e horas. A **javax.time** utiliza como padrão os métodos **get()** retornando valores numéricos para tais campos, por exemplo, **getYear()** retorna o ano como um inteiro e os métodos **to()** retornam as classes específicas, no caso do ano, seria a classe **javax.time.calendar.field.Year**. O exemplo apresentado a seguir ilustra o uso do método **toYear()**.

```

LocaleDate hoje = Clock.systemDefaultZone().today();
Year ano = hoje.toYear();
boolean bissexto = ano.isLeap();

```

No exemplo, obtemos o ano e, a partir do objeto criado, verificamos se ele é bissexto. Repare como isso seria trabalhoso caso o ano fosse representado somente pelo tipo primitivo (**int**). Teríamos que fazer esse cálculo manualmente, cada um em seu projeto, testar esse código e mantê-lo.

No exemplo apresentado a seguir, temos a obtenção do mês, representado pela enumeração **javax.time.calendar.field.MonthOfYear** e, a partir dela, obtemos o último dia do mês representado pela classe **javax.time.calendar.field.DayOfMonth**.

```

LocaleDate hoje = Clock.systemDefaultZone().today();
MonthOfYear mes = hoje.toMonthOfYear();
DayOfMonth ultimoDiaMes = mes.getLastDayOfMonth(hoje.toYear());

```

Vemos aqui também, com a obtenção do último dia do mês, o grande valor no uso de classes específicas em comparação com tipos primitivos. O cálculo do último dia do mês é resolvido automaticamente pela API, poupando você, desenvolvedor, desse (grande) trabalho.

LocalTime

A classe **javax.time.calendar.LocalTime** representa uma hora sem time zone ou offset. O trecho de código apresentado a seguir imprime a hora corrente e, posteriormente, subtrai uma hora e imprime o valor atualizado.

```

LocalTime agora = Clock.systemDefaultZone().currentTime();
System.out.println(agora);

agora = agora.minusHours(1);
System.out.println(agora);

```

LocalDateTime

A classe **javax.time.calendar.LocalDateTime** representa uma data e hora sem time zone ou offset. O trecho de exemplo apresentado a seguir imprime a hora corrente e posteriormente subtrai 36 horas e imprime o valor atualizado.

```

LocalDateTime agora = Clock.systemDefaultZone().currentDateTime();
System.out.println(agora);

agora = agora.minusHours(36);
System.out.println(agora);

```

No exemplo, caso a primeira data e hora fosse representada por “2009-02-07T11:50:08.093”, após a subtração de 36 horas, teríamos “2009-02-05T23:50:08.093”. Repare que o cálculo ocorreu corretamente e alterou até mesmo a data.

Datas e Horas com offset

As datas e horas com offset representam um valor relativo ao UTC (Coordinated Universal Time ou Tempo Universal Coordenado). Esses valores relativos, ou offsets, estão geralmente casados com as áreas de fuso horário (time zones), porém a **javax.time** separa os dois conceitos: offsets e time zones em classes distintas. O objetivo dessa separação é tratar trocas de offset por uma mesma time zone devido a horários de inverno ou verão.

OffsetDate

A classe **javax.time.calendar.OffsetDate** representa uma data com um offset em relação ao UTC. Por exemplo, ao criar um objeto **OffsetDate** aqui no Brasil, na região que segue o horário de Brasília, temos como offset o valor -03h00min, ou seja, se em Brasília são 16h, o horário UTC estará marcando 13h.

O trecho de código apresentado a seguir ilustra a obtenção da data atual considerando o offset:

```
OffsetDate hoje = Clock.systemDefaultZone().offsetDateTime().toOffsetDate();
```

Caso esteja sob o offset de Brasília, ao imprimir o objeto, a string “2009-02-07-03:00” teria sido impressa no console, seguindo o padrão ISO-8601 para datas e horas com offset.

OffsetTime

A classe **javax.time.calendar.OffsetTime** representa um horário com um offset em relação ao UTC, seguindo o mesmo padrão da classe **OffsetDate**.

O trecho de código apresentado a seguir ilustra a obtenção do horário atual considerando o offset. Após isso, a criação de uma nova instância de **OffsetTime** contendo o mesmo horário, porém com o offset encontrado no Japão, que é +09h00min. Após isso, fazemos uma comparação dos dois horários com o intuito de ilustrar que a comparação levará em conta o offset.

```
OffsetTime hojeBrasil = Clock.systemDefaultZone().offsetDateTime().toOffsetTime();
OffsetTime hojeJapao = hojeBrasil.withOffset(ZoneOffset.zoneOffset(9));
boolean horarioJapaoPosterior = hojeJapao.isAfter(hojeBrasil);
```

Após a execução do código apresentado, a variável **horarioJapaoPosterior** irá armazenar o valor **true** indicando que a hora no Japão é posterior.

OffsetDateTime

A classe **javax.time.calendar.OffsetDateTime** representa uma data e um horário com um offset em relação ao UTC, seguindo o padrão dos dois exemplos vistos até o momento. O código a seguir ilustra a obtenção de uma **OffsetDateTime**.

```
OffsetDateTime hojeBrasil = Clock.systemDefaultZone().offsetDateTime();
```

Ao imprimir essa variável no console, temos uma string representando a data, horário e o offset seguindo o padrão ISO: “2009-02-09T23:20:54.171-03:00”.

Time Zones

Além de objetos capazes de armazenar o offset, temos a classe **javax.time.calendar.ZonedDateTime** que armazena também o time zone. O time zone é representado pela classe **javax.time.calendar.TimeZone** que tem o propósito de tratar regras e exceções das alterações nos offsets das regiões mundiais. Essas alterações geralmente se devem a mudanças no horário de verão ou inverno.

Dentro da **javax.time**, é possível instanciar a classe **TimeZone** a partir de um identificador da base de time zones *zoneinfo*⁴ ou a partir do offset em relação a UTC.

O código apresentado a seguir ilustra a obtenção de duas instâncias de **ZonedDateTime**, a primeira com o time zone padrão da JVM, configurado como “América/Sao_Paulo” e o segundo forçando o uso do time zone de Paris.

```
ZonedDateTime agora = Clock.systemDefaultZone().zonedDateTime();
System.out.println(agora);
```

```
TimeZone timeZone = TimeZone.getTimeZone("Europe/Paris");
agora = Clock.system(timeZone).zonedDateTime();
System.out.println(agora);
```

Ao executar o código apresentado, as datas e horas correspondentes ao time zone informado são impressas no console:

```
2009-02-09T23:37:18.968-03:00 UTC-03:00
2009-02-10T03:37:18.968+01:00 Europe/Paris
```

Repare que a primeira string impressa apresenta o time zone como UTC-03:00, isso por que não há um identificador correspondente para este time zone, ao contrário do que ocorreu com o time zone de Paris, que criamos a partir do identificador, que é apresentado na impressão do objeto.

Uma forma equivalente de obter a instância de **ZonedDateTime** à que utilizamos é a apresentada a seguir, uma vez que o time zone de Paris corresponde ao offset de uma hora em relação ao UTC:

```
ZonedDateTime agora = Clock.systemDefaultZone().zonedDateTime();
System.out.println(agora);
```

```
TimeZone timeZone = TimeZone.getTimeZone(ZoneOffset.of(1));
agora = Clock.system(timeZone).zonedDateTime();
System.out.println(agora);
```

Matchers e Resolvers

Como já dissemos, a **javax.time** recorre a conceitos do design pattern *Strategy* para que você possa customizar pontos de seu comportamento. Um exemplo pelo qual já passamos foi o dos *Adjusters* que permitem a realização de ajustes em datas e horas de forma bastante flexível.

Além dos *Adjusters*, a **javax.time** nos provê o conceito de *Matchers* e *Resolvers*, que veremos agora.

Matchers

Os *Matchers* possuem a responsabilidade de realizar consultas em datas e horas de forma muito simples e flexível. Eles reduzem drasticamente a quantidade de código e a lógica empregada neste tipo de operação.

O código apresentado a seguir ilustra a consulta à data (e hora) atual. Nesse caso consultamos se o ano é 2009, valorizando uma variável booleana denominada **ano2009**. Como estamos realmente em 2009, essa variável será valorizada com **true**.

```
LocalDateTime agora = Clock.systemDefaultZone().dateTime();
boolean ano2009 = agora.matches(Year.isYear(2009));
```

A consulta envolvendo o horário é praticamente idêntica, conforme pode ser visto a seguir:

```
LocalDateTime agora = Clock.systemDefaultZone().dateTime();
```

⁴ A base de dados denominada *zoneinfo* ou *tz* é um conjunto das zonas de fuso horário de todo o mundo. Essa base de dados também é chamada de Olson Database devido ao nome de seu criador, Arthur David Olson.

```
boolean vinteUmaHoras = agora.matches(HourOfDay.hourOfDay(21));
```

É possível realizar tais operações, pois cada uma dessas classes (**Year**, **HourOfDay** e as outras classes representando os elementos do calendário) implementa as interfaces necessárias para executar o método **matches()**.

A interface que representa o *Matcher* para a consulta em datas é a **javax.time.calendar.DateMatcher**, que possui um único método **boolean matchesDate(LocalDate input)**. Já a interface para a consulta em horários é a **javax.time.calendar.TimeMatcher** e ela segue o mesmo padrão da outra que vimos, possuindo um único método **boolean matchesTime(LocalTime time)**.

Essas interfaces estão disponíveis para você, desenvolvedor, implementar sua própria estratégia de consulta nas datas ou nos horários. Com isso, você garante que terá essa lógica implementada em um objeto coeso e reutilizável.

A **Listagem 1** apresenta um exemplo de implementação de **DateMatcher** para verificar se a data possui um dia ímpar ou não.

A classe **javax.time.calendar.DateMatchers** possui alguns *matchers* pré-configurados para você utilizar. Entre os *matchers* que já vêm por padrão temos, por exemplo, um para verificar se estamos durante a semana ou em um final de semana, se é o primeiro ou último dia do mês, além de alguns outros.

O código apresentado a seguir ilustra o uso da classe **DateMatchers** para a verificação se a data corrente é um final de semana.

```
LocalDateTime agora = Clock.systemDefaultZone().dateTime();
boolean finalDeSemana = agora.matches(DateMatchers.weekendDay());
```

Como visto, a classe **DateMatcher** pode nos auxiliar com a verificação de datas e nos auxiliar muito em tarefas do mundo real como, por exemplo, verificar se determinada data está dentro do domingo de páscoa ou na sexta-feira santa.

Resolvers

Assim como os *matchers*, os *Resolvers* são pontos de extensibilidade disponíveis na **javax.time**. Através deles você pode indicar como deseja que uma data inválida seja tratada, por exemplo, ao criar uma data no dia 29 de fevereiro em um ano que não seja bissexto. A classe **javax.time.calendar.DateResolvers** possui algumas implementações dos casos de uso mais comuns onde possa vir a ser necessário o uso de um *Resolver*.

O exemplo apresentado a seguir ilustra a criação de uma data utilizando um resolver provido pela classe **DateResolvers** que resolve uma data inválida como a próxima data válida.

```
LocalDate data = DateResolvers.nextValid().resolveDate(Year.isoYear(2009),
    MonthOfYear.monthOfYear(2), DayOfMonth.dayOfMonth(29));
```

Ao imprimir tal data no console, temos como data apresentada 2009-03-01. Caso você tenha achado o código verboso demais, é possível utilizar o recurso de **static imports** para reduzir a verbosidade. A **Listagem 2** apresenta tal opção.

Períodos

Os períodos, dentro da **javax.time**, são cidadãos de primeira-classe. Isso quer dizer que existem classes capazes de representá-los, ao contrário do que você encontra nas APIs atuais, onde não há forma padrão para representar períodos.

Através da representação dos períodos, podemos expressar durações de tempo da maneira tratada pelos seres humanos como, por exemplo, a duração de uma reunião ou de suas férias.

Como dito, existe uma classe para cada parte do período:

- **javax.time.period.field.Days**

- `javax.time.period.field.Hours`
- `javax.time.period.field.Minutes`
- `javax.time.period.field.Months`
- `javax.time.period.field.Seconds`
- `javax.time.period.field.Weeks`
- `javax.time.period.field.Years`

É uma classe para representar o período como um todo: `javax.time.period.Period`. O código apresentado a seguir representa a criação de um período de duas horas:

```
Period period = Period.hours(2);
```

A partir desse momento, o objeto representa o determinado período para o qual foi criado. Ao imprimir tal período no console, é impressa sua representação de acordo com o padrão ISO-8106. Essa representação para o período de duas horas é “PT2H”.

Uma das grandes vantagens no uso de objetos especializados para os períodos é permitir que você execute operações nesse objeto. Isso resultará na criação de outro objeto para representar o novo período resultante da operação. O objeto que representa o período também é imutável. A seguir, uma subtração de quarenta minutos de nosso período de duas horas:

```
period = period.minusMinutes(40);
```

Ao executar tal operação, o período ainda permanece com duas horas e *menos* quarenta minutos e necessita ser *normalizado* para representar o período real, que seria de uma hora e vinte minutos.

Normalização

O método `normalize()` retorna uma cópia do período normalizado para os limites padrão dos campos de data e hora, levando em conta as seguintes regras:

- 12 meses em um ano
- 60 minutos em uma hora
- 60 segundos em um minuto
- 1.000.000.000 de nanossegundos em um segundo

Por exemplo, um período de 13 meses é normalizado para um ano e um mês e a criação de um período de 5000 minutos com o código `Period.minutes(5000)` resulta em um período normalizado de 83 horas e 20 minutos.

Caso você necessite ainda realizar a normalização por dias, quebrando cada 24 horas em um dia, você deve utilizar o método `normalizedWith24HourDays()`, que neste nosso exemplo resultaria em um período de três dias, onze horas e vinte minutos.

Conclusão

Você conheceu um pouco sobre a nova especificação para representar datas e horas dentro da plataforma Java. Essa nova API tenta trazer várias facilidades e corrigir problemas conhecidos das opções atuais, buscando transformar seu código em algo mais legível e simples. Passamos por todos os pontos mais importantes da nova API, fornecendo exemplos pontuais e focados, tentando ambientar você com os novos conceitos.

A JSR 310, que está definindo a nova API, é aberta e você pode contribuir para seu desenvolvimento. Acesse o site da JSR (veja seção de links) e faça parte da definição do futuro da linguagem. O código apresentado neste artigo está disponível para download no site da revista, contando com um JAR da `javax.time` compilado durante a escrita do artigo. Como a API ainda está sendo concebida, é possível que quando o artigo chegar a você, leitor, algo pode ter sido alterado. Devido a isso, deixamos aqui a sugestão de baixar os fontes da última versão da API e adequar os

exemplos às possíveis alterações. Acesse também a lista de discussões para entender o rumo que está sendo dado à API.

Links

jsr-310.dev.java.net

Site de desenvolvimento da JSR 310, onde você pode obter acesso a todos os artefatos relacionados à nova API, além de participar de seu desenvolvimento, cadastrando-se nas listas de discussão.

jcp.org/en/jsr/detail?id=310

Site oficial da JSR 310, parte do JCP.

joda-time.sourceforge.net

Site da Joda Time, API que está servindo de base para o desenvolvimento da JSR 310.

Listagem 1. Exemplo do uso de um DateMatcher customizado para realizar a consulta de dias ímpares

```
package br.com.jm.javax.time.human.matchers;

//imports...

public class ExemploMatcherDialmpar {

    public static void main(String[] args) throws Exception {
        LocalDateTime agora = Clock.systemDefaultZone().dateTime();
        boolean dialmpar = agora.matches(new DialmparMatcher());
        System.out.println(dialmpar);
    }
}

class DialmparMatcher implements DateMatcher {
    @Override
    public boolean matchesDate(LocalDate data) {
        if (data.getDayOfMonth() % 2 == 0) {
            return false;
        } else {
            return true;
        }
    }
}
```

Listagem 2. Uso de static imports para minimizar a verbosidade do código no uso de Resolvers

```
package br.com.jm.javax.time.human.resolvers;

import static javax.time.calendar.field.DayOfMonth.dayOfMonth;
import static javax.time.calendar.field.MonthOfYear.monthOfYear;
import static javax.time.calendar.field.Year.isoYear;

import javax.time.calendar.DateResolvers;
import javax.time.calendar.LocalDate;

public class ExemploResolverSI {

    public static void main(String[] args) throws Exception {
        LocalDate data = DateResolvers.nextValid().resolveDate(isoYear(2009),
            monthOfYear(2), dayOfMonth(29));

        System.out.println(data);
    }
}
```

BOX LEAD

LEAD 2

De que se trata o artigo:

O artigo aborda o resultado da JSR-310, a JSR que está definindo a nova API de data e hora que será incorporada na plataforma Java. A arquitetura e as principais classes e interfaces da API são discutidas e exemplificadas. Além disso, são abordadas as lacunas presentes nas classes atuais (Date e Calendar) e como a `javax.time` endereça tais problemas.

Para que serve:

Apresentar a nova API de datas e horas que será incorporada na plataforma Java, provendo discussões sobre a sua arquitetura e exemplos de uso da API, além de comparações com as classes atuais.

Em que situação o tema é útil:

O tema é importante para qualquer um que deseje estar atualizado sobre as novas tendências da plataforma Java. A manipulação de datas e horas é peça-chave em qualquer implementação e a criação de uma nova API tende a facilitar muito a vida do desenvolvedor.

LEAD 3: BOX resumo Devman

Novos tempos: javax.time

O artigo aborda a nova API para representação de datas e horas da plataforma Java e como ela facilita a vida do desenvolvedor Java, corrigindo bugs e endereçando problemas das classes atuais Date e Calendar. A nova API possui representações distintas para datas que serão utilizadas para cálculos computacionais e datas que serão utilizadas por seres humanos. Além disso, temos representações padrão para tipos de dados que atualmente precisamos criar como durações, períodos e intervalos.

Daniel Cicero Amadei (daniel.amadei@gmail.com) é Bacharel em Sistemas de Informação pelo Mackenzie e pós-graduado pela Fundação Vanzolini. Trabalha com Java desde 1999 e possui as certificações SCJP, SCWCD, SCBCD, SCDJWS, SCEA, BEA Certified Developer: Integration Solutions e BEA Certified SOA Architect. Já atuou como Desenvolvedor, Analista e Arquiteto de Sistemas e atualmente é Consultor em Arquitetura BEA.

Michael Nascimento Santos (michael@michaelnascimento.com.br) é um dos spec-leads da JSR-310 e expert em outras cinco JSRs. Ganhou um JavaOne Rock Star Speaker Award pela sua palestra sobre a JSR-310 em 2008 no JavaOne. Atua como Senior Technical Consultant na Summa Technologies do Brasil.