**Parte SEIS**

**API de Data/Hora**

**Capítulo VINTE E UM**

**Classes Principais de Data/Hora**

**Objetivos do Exame**

Criar e gerenciar eventos baseados em data e hora, incluindo uma combinação de data e hora em um único objeto usando LocalDate, LocalTime, LocalDateTime, Instant, Period e Duration.

Definir, criar e gerenciar eventos baseados em data e hora usando Instant, Period, Duration e TemporalUnit.

**Uma nova API de Data/Hora**

Desde o início do Java, java.util.Date e java.util.Calendar (introduzido posteriormente) têm sido as classes usadas ao se trabalhar com datas e horas.

No entanto, essas classes estão longe de serem perfeitas. Alguns de seus problemas são:

* java.util.Date representa o tempo com "apenas" precisão de milissegundos (o que pode não ser suficiente em algumas aplicações). Os anos começam em 1900 e os meses começam no 0.
* O fuso horário da data é o fuso horário padrão da JVM.
* Tanto java.util.Date quanto java.util.Calendar são classes mutáveis, o que significa que, quando elas mudam, não criam outra instância com os novos valores (o que não é ideal agora que você pode programar em um estilo funcional com Java).

Por essas razões, o Java 8 introduziu uma nova API de Data/Hora baseada na popular biblioteca Joda-Time e contida também no novo pacote java.time.

Este capítulo tratará das classes principais da nova API, que não fornecem informações sobre fuso horário. As classes que fornecem informações de fuso horário serão o tópico do próximo capítulo.

Vamos começar com uma visão geral de alto nível das classes principais.

Todas essas classes são imutáveis, seguras para threads e, com exceção de Instant, não armazenam ou representam um fuso horário.

Por um lado, temos:

**LocalDate**  
Representa uma data com as informações de ano, mês e dia do mês. Por exemplo, 2015-08-25.

**LocalTime**  
Representa uma hora com informações de hora, minutos, segundos e nanossegundos. Por exemplo, 13:21.05.123456789.

**LocalDateTime**  
Uma combinação dos anteriores. Por exemplo, 2015-08-25 13:21.05.12345.

Por outro lado:

**Instant**  
Representa um ponto único no tempo em segundos e nanossegundos. Por exemplo, 923.456.789 segundos e 186.054.812 nanossegundos.

**Period**  
Representa uma quantidade de tempo em termos de anos, meses e dias. Por exemplo, 5 anos, 2 meses e 9 dias.

**Duration**  
Representa uma quantidade de tempo em termos de segundos e nanossegundos. Por exemplo, 12,87656 segundos.

LocalDate, LocalTime, LocalDateTime e Instant implementam a interface java.time.temporal.Temporal, portanto todos têm métodos semelhantes.

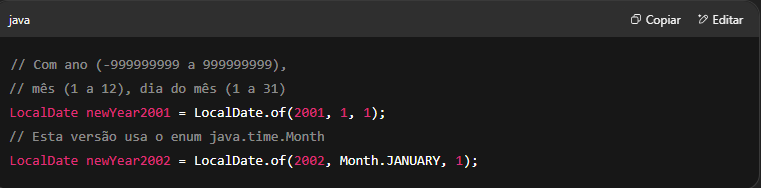
Enquanto Period e Duration implementam a interface java.time.temporal.TemporalAmount, o que também os torna muito semelhantes.

**Classe LocalDate**

A chave para aprender como usar esta classe é ter em mente que ela contém o ano, mês, dia e informações derivadas de uma data. Todos os seus métodos usam essas informações ou têm uma versão para trabalhar com cada uma delas.

Os seguintes são os métodos mais importantes (mais usados) desta classe.

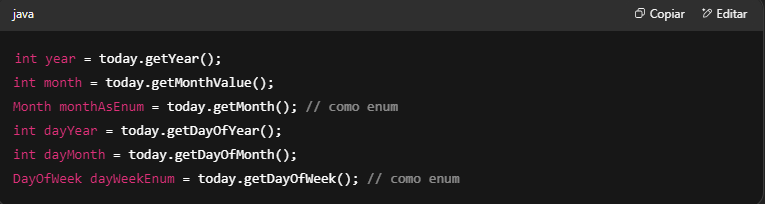
Para criar uma instância, podemos usar o método estático of:

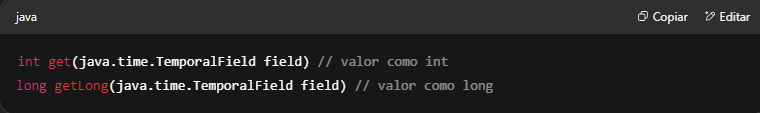


Note que, ao contrário de java.util.Date, os meses começam em um. Se você tentar criar uma data com valores inválidos (como 29 de fevereiro), uma exceção será lançada. Para a data de hoje use now():

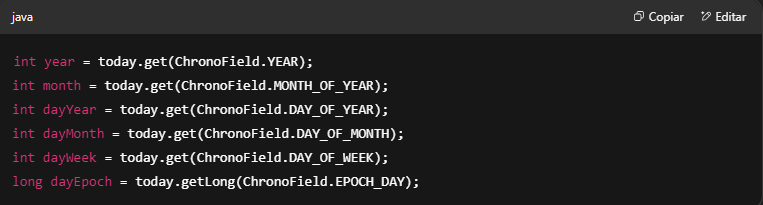


Uma vez que temos uma instância de LocalDate, podemos obter o ano, o mês e o dia com métodos como os seguintes:

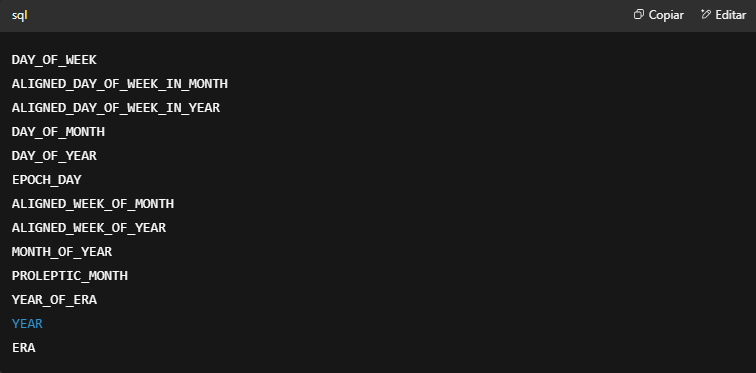
Também podemos usar o método get:



Que recebe uma implementação da interface java.time.TemporalField para acessar um campo específico de uma data. java.time.ChronoField é uma enumeração que implementa esta interface, então podemos ter, por exemplo:

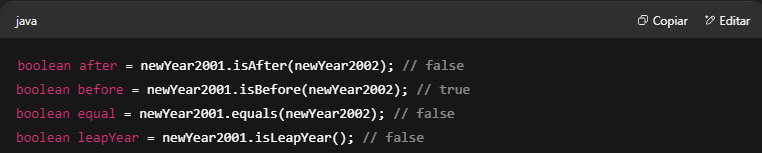


Os valores suportados para ChronoField são:



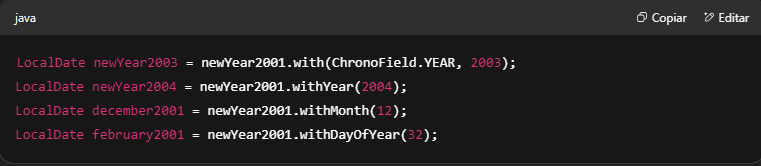
Usar um valor diferente lançará uma exceção. O mesmo ocorre ao obter um valor que não cabe em um int com get(TemporalField).

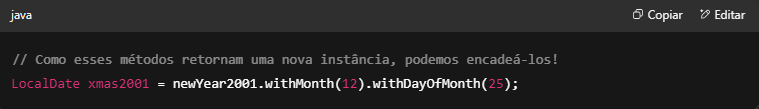
Para verificar um LocalDate contra outra instância, temos três métodos, além de um para anos bissextos:



Uma vez que uma instância desta classe é criada, não podemos modificá-la, mas podemos criar outra instância a partir de uma existente.

Uma forma é através do método with() e suas versões:



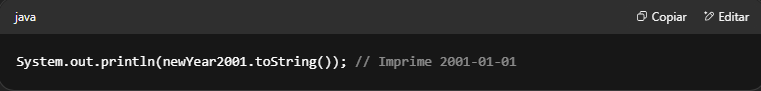


Outra forma é adicionando ou subtraindo — adivinhe — anos, meses, dias ou até semanas:



Note que as versões plus e minus usam a enumeração java.time.temporal.ChronoUnit, diferente de java.time.ChronoField. Os valores suportados são:



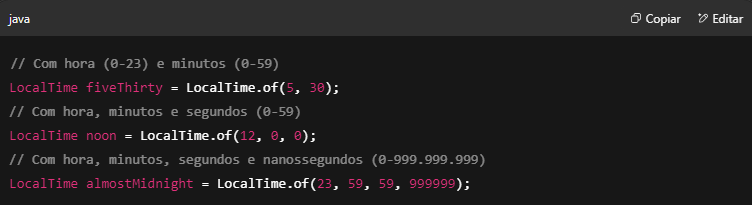
Por fim, o método toString() retorna a data no formato uuuu-MM-dd: 

**Classe LocalTime**

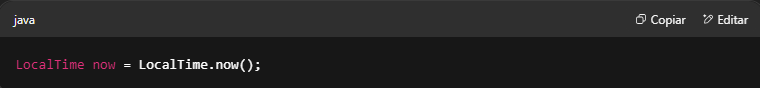
A chave para aprender a usar esta classe é ter em mente que ela contém a hora, minutos, segundos e nanossegundos. Todos os seus métodos usam essas informações ou têm uma versão para trabalhar com cada uma delas.

A seguir estão os métodos mais importantes (mais usados) desta classe. Como você pode ver, são os mesmos (ou muito semelhantes) métodos de LocalDate, adaptados para funcionar com hora em vez de data.

Para criar uma instância, podemos usar o método estático of:



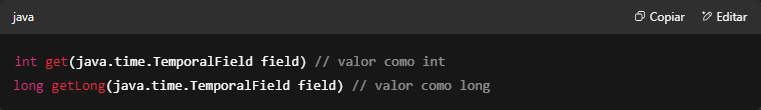
Se você tentar criar um horário com um valor inválido (como LocalTime.of(24, 0)), uma exceção será lançada. Para obter o horário atual, use now():



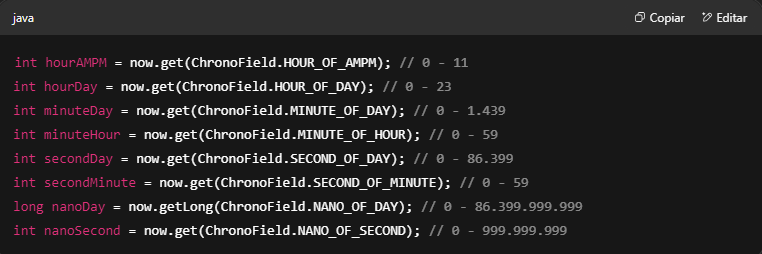
Uma vez que temos uma instância de LocalTime, podemos obter a hora, os minutos e outras informações com métodos como os seguintes:



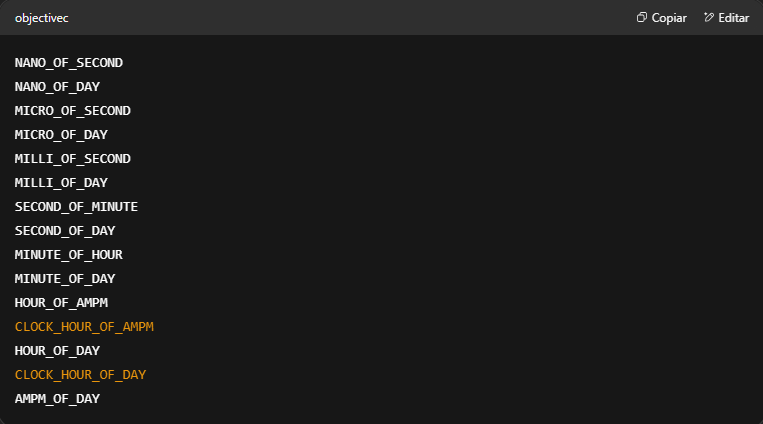
Também podemos usar o método get():



Assim como no caso de LocalDate, podemos ter, por exemplo:

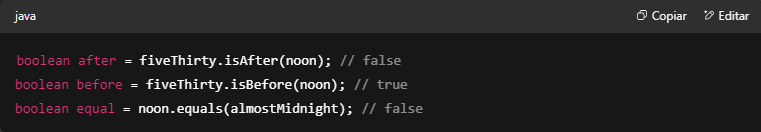


Os valores suportados para ChronoField são:



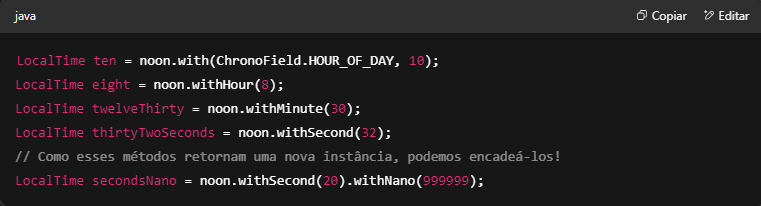
Usar um valor diferente lançará uma exceção. O mesmo ocorre ao obter um valor que não cabe em um int com get(TemporalField).

Para comparar um objeto LocalTime com outro, temos três métodos:



Assim como LocalDate, uma vez que uma instância de LocalTime é criada, não podemos modificá-la, mas podemos criar outra instância a partir de uma existente.

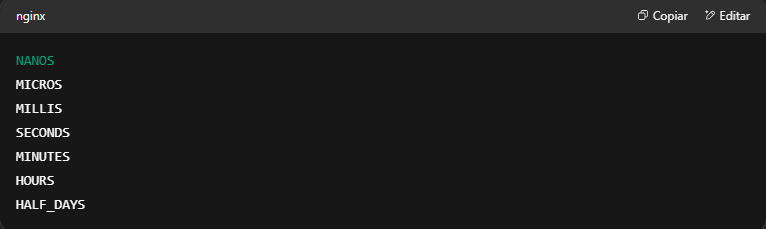
Uma forma é através do método with e suas versões:



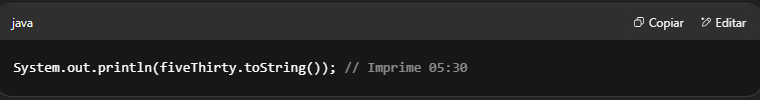
Claro, outra maneira é adicionando ou subtraindo horas, minutos, segundos ou nanossegundos:



Note que as versões plus e minus usam a enumeração java.time.temporal.ChronoUnit, diferente de java.time.ChronoField. Os valores suportados são:



Por fim, o método toString() retorna o horário no formato HH:mm:ss.SSSSSSSSS, omitindo as partes com valor zero (por exemplo, retornando apenas HH:mm se os segundos/nanossegundos forem zero):

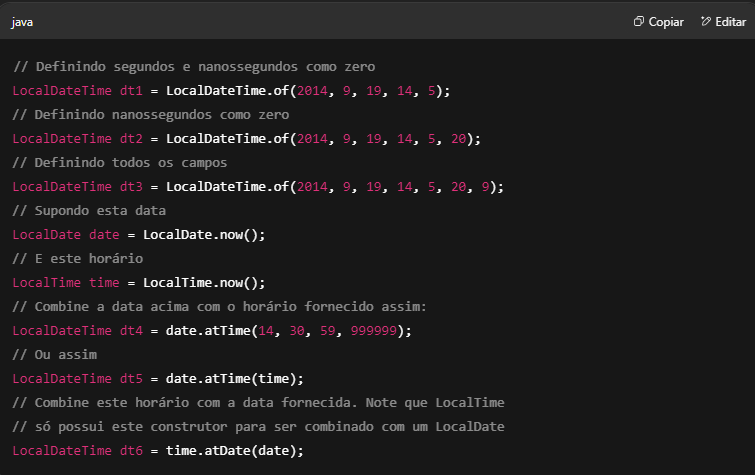


**Classe LocalDateTime**

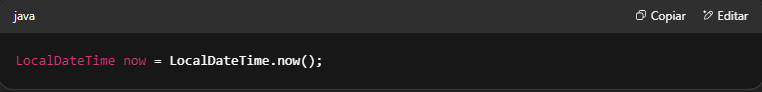
A chave para aprender a usar esta classe é lembrar que ela combina as classes LocalDate e LocalTime.

Ela representa tanto uma data quanto uma hora, com informações como ano, mês, dia, horas, minutos, segundos e nanossegundos. Outros campos, como dia do ano, dia da semana e semana do ano também podem ser acessados.

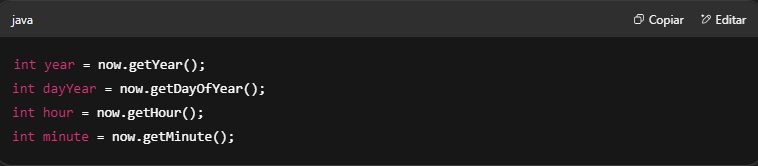
Para criar uma instância, podemos usar o método estático of() ou a partir de uma instância de LocalDate ou LocalTime:



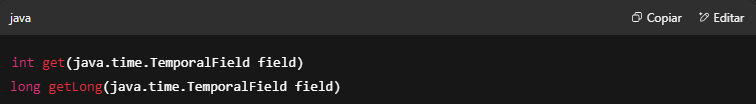
Se você tentar criar uma instância com um valor ou data inválida, uma exceção será lançada. Para obter a data/hora atual, use now():



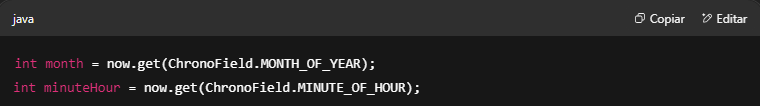
Uma vez que temos uma instância de LocalDateTime, podemos obter as informações com os métodos que já conhecemos de LocalDate e LocalTime, como:



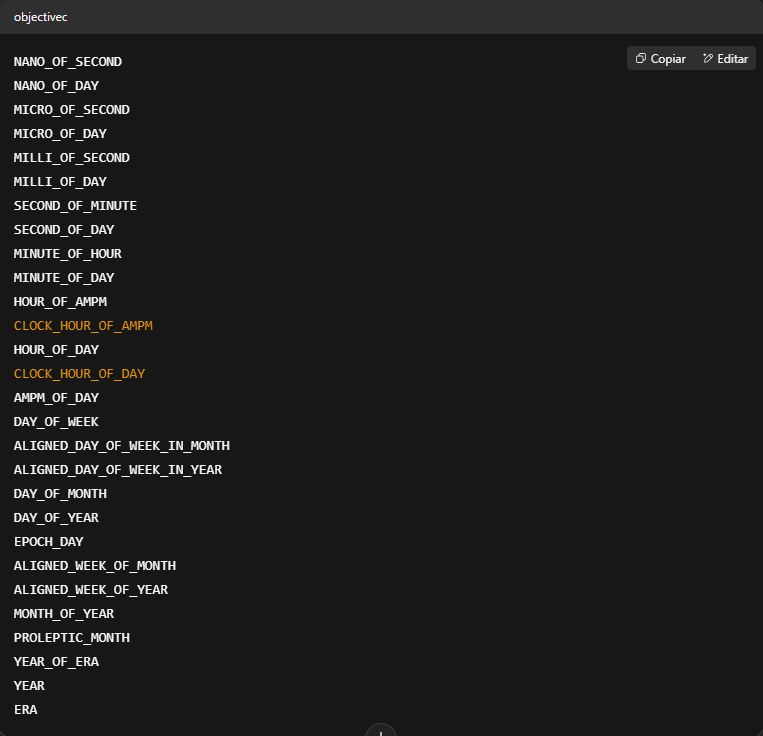
Também podemos usar o método get():



Por exemplo:

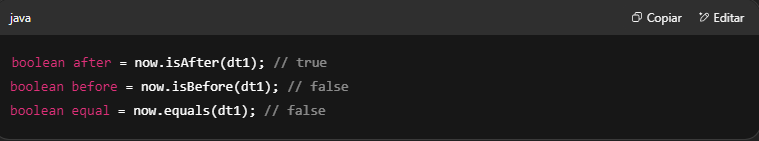


Os valores suportados para ChronoField são:



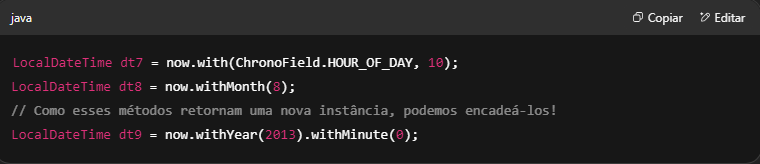
Usar um valor diferente lançará uma exceção. O mesmo ocorre ao obter um valor que não cabe em um int com get(TemporalField).

Para verificar um objeto LocalDateTime em relação a outro, temos três métodos:

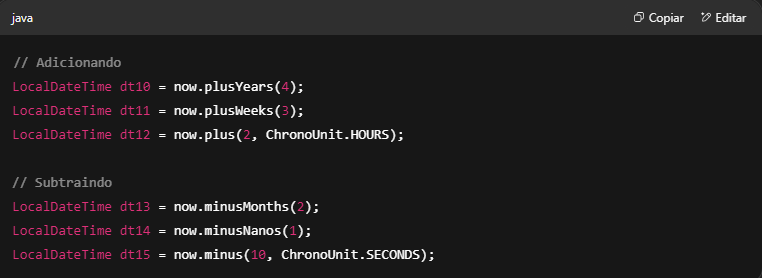


Uma vez que uma instância de LocalDateTime é criada, não podemos modificá-la, mas podemos criar outra instância a partir de uma existente.

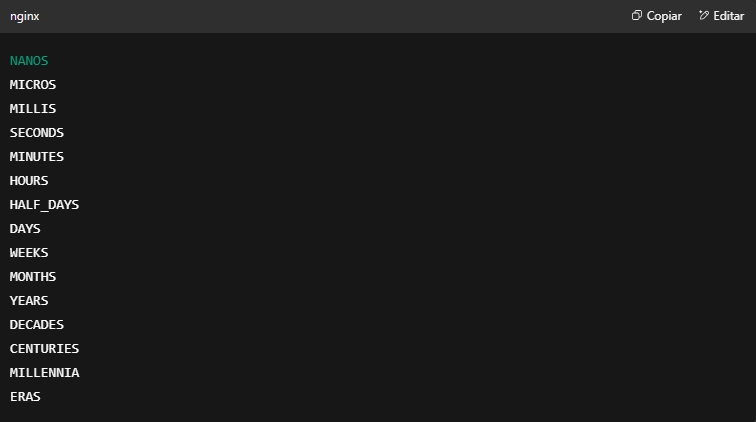
Uma forma é através do método with e suas versões:



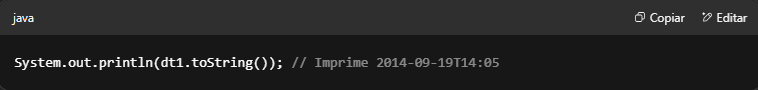
Outra maneira é adicionando ou subtraindo anos, meses, dias, semanas, horas, minutos, segundos ou nanossegundos:



Nesse caso, os valores suportados para ChronoUnit são:



Por fim, o método toString() retorna a data-hora no formato uuuu-MM-dd'T'HH:mm:ss.SSSSSSSSS, omitindo as partes com valor zero. Por exemplo:



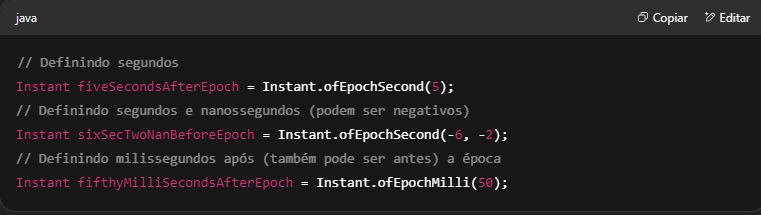
**Classe Instant**

Embora, em termos práticos, uma instância de LocalDateTime represente um instante na linha do tempo, há outra classe que pode ser mais apropriada.

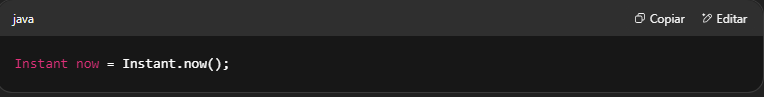
No Java 8, a classe java.time.Instant representa um instante no número de segundos que se passaram desde a época (epoch), uma convenção usada em sistemas UNIX/POSIX e definida à meia-noite de 1º de janeiro de 1970 no horário UTC.

A partir dessa data, o tempo é medido em 86.400 segundos por dia. Essa informação é armazenada como um long. A classe também oferece suporte a precisão de nanossegundos, armazenada como um int.

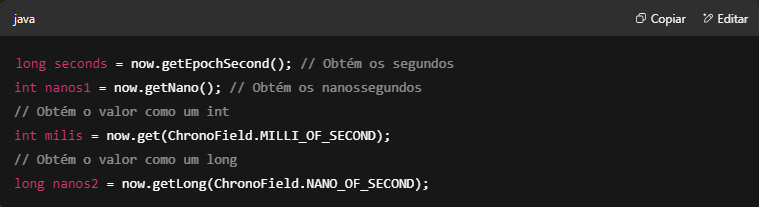
Você pode criar uma instância desta classe com os seguintes métodos:



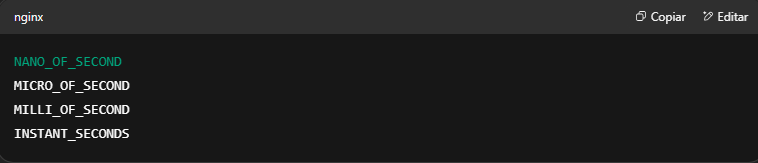
Para obter a instância atual do relógio do sistema, use:



Uma vez que temos uma instância de Instant, podemos obter as informações com os seguintes métodos:

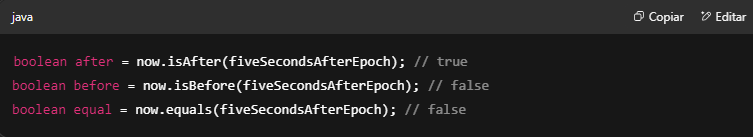


Os valores de ChronoField suportados são:



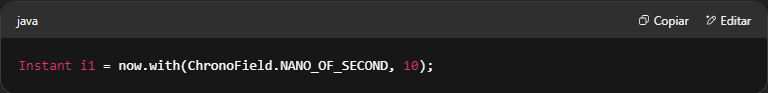
Usar qualquer outro valor lançará uma exceção. O mesmo ocorre ao obter um valor que não cabe em um int com get(TemporalField).

Para verificar um objeto Instant em relação a outro, temos três métodos:

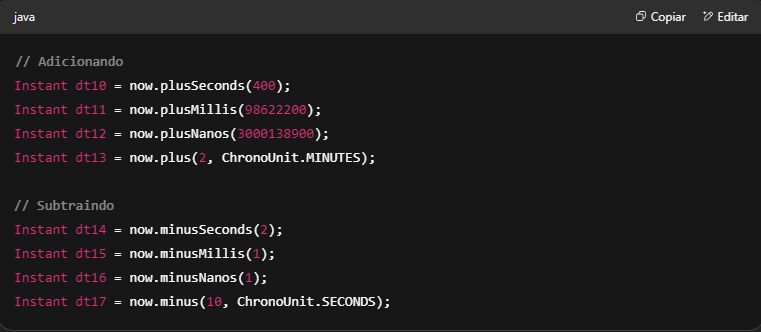


Uma vez que uma instância deste objeto é criada, não podemos modificá-la, mas podemos criar outra instância a partir de uma existente.

Uma forma é através do método with:



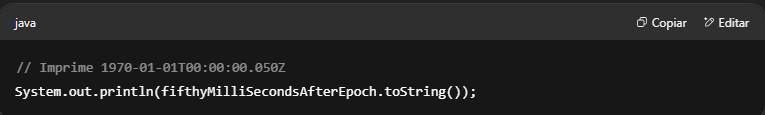
Outra forma é adicionando ou subtraindo segundos, milissegundos ou nanossegundos:



Os valores suportados de ChronoUnit são:



Por fim, o método toString() retorna a instância no formato uuuu-MM-dd'T'HH:mm:ss.SSSSSSSSS, por exemplo:

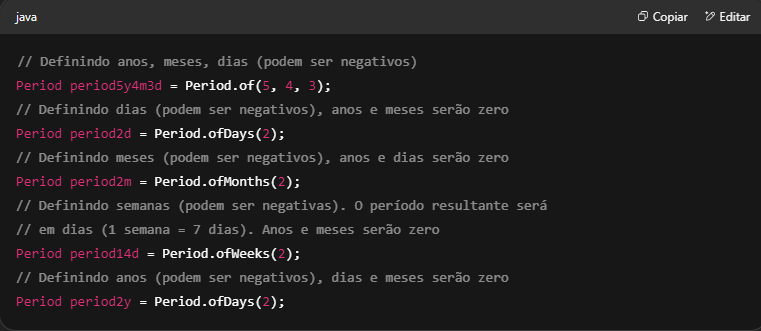


Note que contém informação de zona de tempo (Z). Isso ocorre porque Instant representa um ponto no tempo a partir da época de 1970-01-01Z no fuso horário UTC.

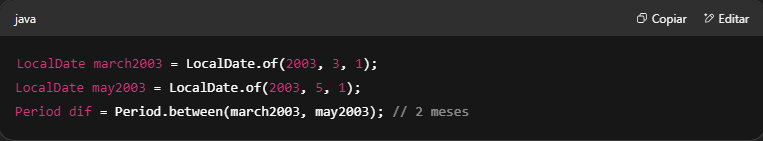
**Classe Period**

A classe java.time.Period representa uma quantidade de tempo em termos de anos, meses e dias.

Você pode criar uma instância dessa classe com os seguintes métodos of:



Um Period também pode ser considerado como a diferença entre dois LocalDate. Felizmente, há um método que suporta esse conceito:



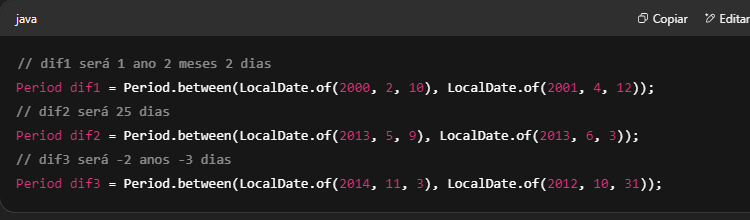
A data inicial é INCLUÍDA, mas a data final NÃO.

Cuidado com a forma como a data é calculada.

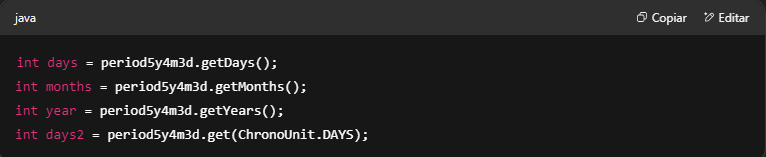
Primeiro, os meses completos são contados, e então o número restante de dias é calculado. O número de meses é então dividido em anos (1 ano equivale a 12 meses). Um mês é considerado se o dia final do mês for maior ou igual ao dia inicial do mês.

O resultado desse método pode ser um período negativo se a data final for anterior à data inicial (ano, mês e dia terão sinal negativo).

Aqui estão alguns exemplos:



Uma vez que temos uma instância de Period, podemos obter as informações com os seguintes métodos:



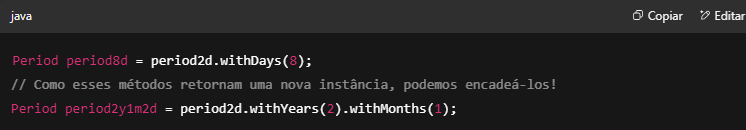
Os valores suportados de ChronoUnit são:



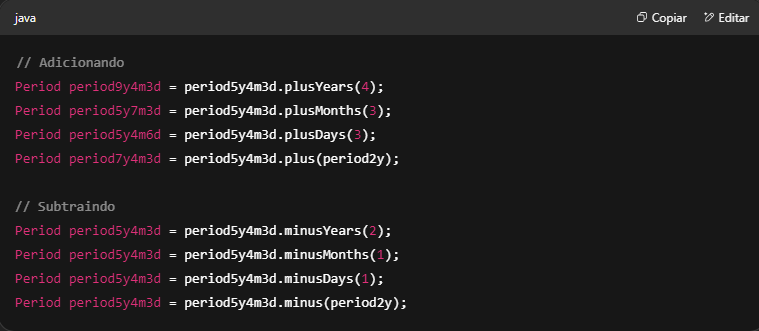
Usar qualquer outro valor lançará uma exceção.

Uma vez que uma instância de Period é criada, não podemos modificá-la, mas podemos criar outra instância a partir de uma existente.

Uma forma é através do método with e suas versões:

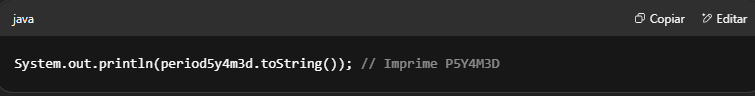


Outra maneira é adicionando ou subtraindo anos, meses ou dias:



Os métodos plus e minus aceitam uma implementação da interface java.time.temporal.TemporalAmount (ou seja, outra instância de Period ou uma instância de Duration).

Por fim, o método toString() retorna o período no formato PNYNMND, por exemplo:

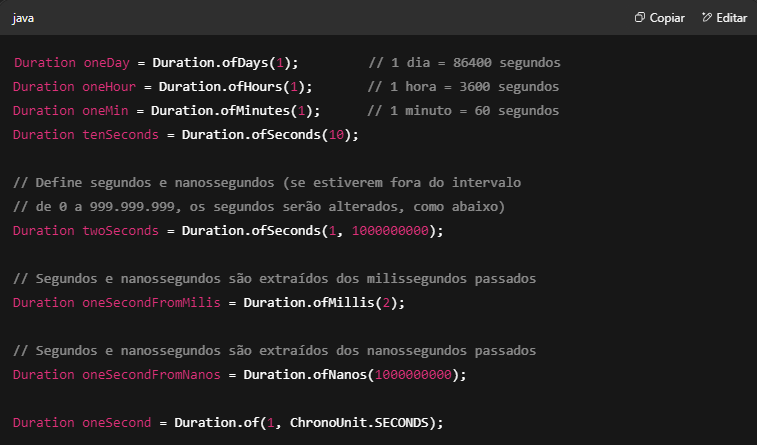


Um período zero será representado como zero dias: P0D.

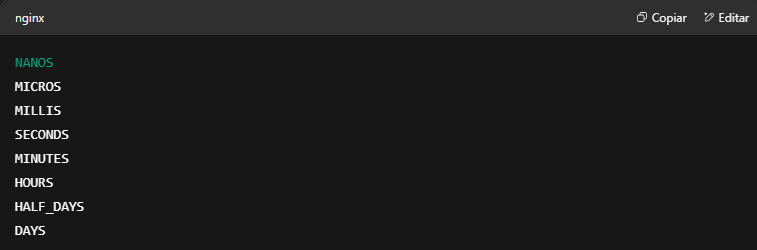
**Classe Duration**

A classe java.time.Duration é como a classe Period, com a única diferença de que ela representa uma quantidade de tempo em termos de segundos e nanossegundos.

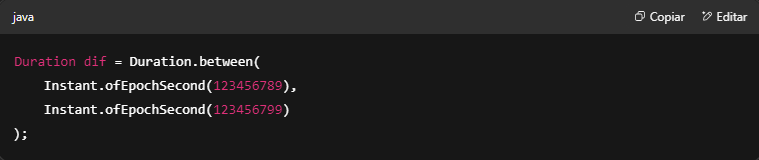
Você pode criar uma instância desta classe com os seguintes métodos of:



Os valores válidos de ChronoUnit são:



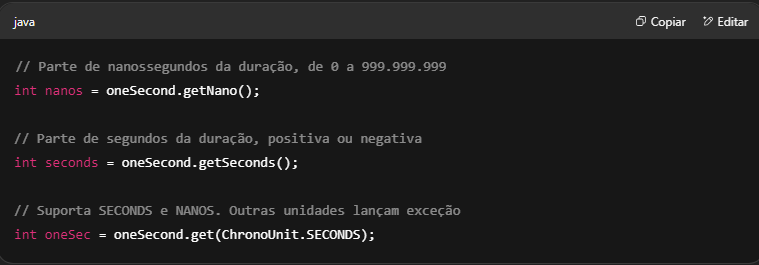
Uma Duration também pode ser criada como a diferença entre duas implementações da interface java.time.temporal.Temporal, desde que suportem segundos (e, para mais precisão, nanossegundos), como LocalTime, LocalDateTime e Instant. Assim, podemos ter algo como:



O resultado pode ser negativo se o fim for antes do início. Uma duração negativa é expressa com sinal negativo na parte dos segundos. Por exemplo, uma duração de -100 nanossegundos é armazenada como -1 segundo mais 999.999.900 nanossegundos.

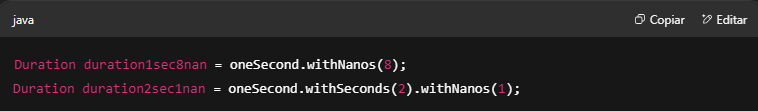
Se os objetos forem de tipos diferentes, então a duração é calculada com base no tipo do primeiro objeto. Mas isso só funciona se o primeiro argumento for um LocalTime e o segundo um LocalDateTime (porque pode ser convertido para LocalTime). Caso contrário, uma exceção é lançada.

Uma vez que temos uma instância de Duration, podemos obter as informações com os seguintes métodos:

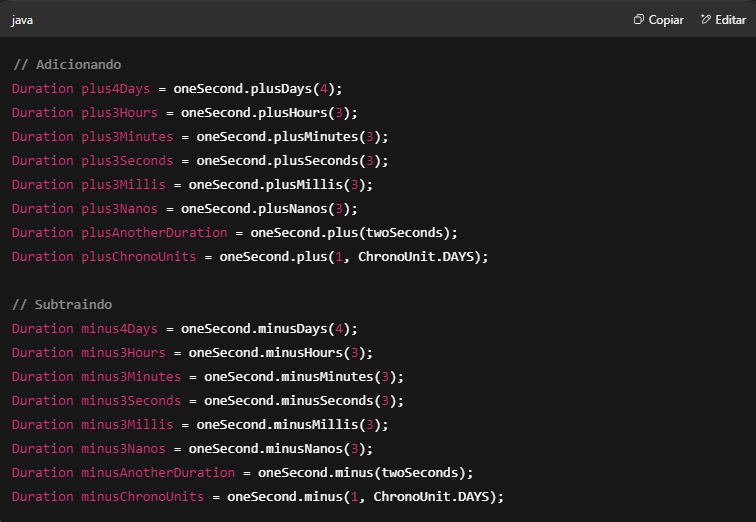


Uma vez que uma instância de Duration é criada, não podemos modificá-la, mas podemos criar outra instância a partir de uma existente.

Uma forma é através do método with e suas versões:



Outra forma é adicionando ou subtraindo dias, horas, minutos, segundos, milissegundos ou nanossegundos:



Os métodos plus e minus aceitam outra Duration ou um valor válido de ChronoUnit (os mesmos usados para criar uma instância).

Por fim, o método toString() retorna a duração com o formato PTnHnMnS. Quaisquer segundos fracionários são colocados após um ponto decimal na seção dos segundos. Se uma seção tiver valor zero, ela é omitida. Por exemplo:

* 2 dias e 4 minutos → PT48H4M
* 45 segundos e 99 milissegundos → PT45.099S

**Pontos-chave**

LocalDate, LocalTime, LocalDateTime, Instant, Period, Duration são as classes principais da nova API de Data/Hora do Java, localizada no pacote java.time.  
Elas são imutáveis, seguras para threads e, com exceção de Instant, não armazenam nem representam um fuso horário.

LocalDate, LocalTime, LocalDateTime e Instant implementam a interface java.time.temporal.Temporal, portanto todos têm métodos semelhantes. Enquanto Period e Duration implementam a interface java.time.temporal.TemporalAmount, o que também os torna muito semelhantes.

* LocalDate representa uma data com informações de ano, mês e dia do mês.  
  Você pode criar uma instância usando:  
  LocalDate.of(2015, 8, 1);

Valores válidos de ChronoField para usar com o método get() são:  
DAY\_OF\_WEEK, ALIGNED\_DAY\_OF\_WEEK\_IN\_MONTH, ALIGNED\_DAY\_OF\_WEEK\_IN\_YEAR,  
DAY\_OF\_MONTH, DAY\_OF\_YEAR, EPOCH\_DAY, ALIGNED\_WEEK\_OF\_MONTH,  
ALIGNED\_WEEK\_OF\_YEAR, MONTH\_OF\_YEAR, PROLEPTIC\_MONTH, YEAR\_OF\_ERA, YEAR, e ERA.

Valores válidos de ChronoUnit para usar com os métodos plus() e minus() são:  
DAYS, WEEKS, MONTHS, YEARS, DECADES, CENTURIES, MILLENNIA e ERAS.

* LocalTime representa um horário com informações de hora, minutos, segundos e nanossegundos.  
  Você pode criar uma instância usando:  
  LocalTime.of(14, 20, 50, 99999);

Valores válidos de ChronoField para usar com o método get() são:  
NANO\_OF\_SECOND, NANO\_OF\_DAY, MICRO\_OF\_SECOND, MICRO\_OF\_DAY, MILLI\_OF\_SECOND,  
MILLI\_OF\_DAY, SECOND\_OF\_MINUTE, SECOND\_OF\_DAY, MINUTE\_OF\_HOUR, MINUTE\_OF\_DAY,  
HOUR\_OF\_AMPM, CLOCK\_HOUR\_OF\_AMPM, HOUR\_OF\_DAY, CLOCK\_HOUR\_OF\_DAY, e AMPM\_OF\_DAY.

Valores válidos de ChronoUnit para usar com os métodos plus() e minus() são:  
NANOS, MICROS, MILLIS, SECONDS, MINUTES, HOURS e HALF\_DAYS.

* LocalDateTime é uma combinação de LocalDate e LocalTime.  
  Você pode criar uma instância usando:  
  LocalDateTime.of(2015, 8, 1, 14, 20, 50, 99999);

Valores válidos de ChronoField e ChronoUnit são uma combinação dos usados para LocalDate e LocalTime.

* Instant representa um ponto único no tempo em segundos e nanossegundos.  
  Você pode criar uma instância usando:  
  Instant.ofEpochSecond(134556767, 999999999);

Valores válidos de ChronoField para usar com o método get() são:  
NANO\_OF\_SECOND, MICRO\_OF\_SECOND, MILLI\_OF\_SECOND, e INSTANT\_SECONDS.

Valores válidos de ChronoUnit para usar com os métodos plus() e minus() são:  
NANOS, MICROS, MILLIS, SECONDS, MINUTES, HOURS, HALF\_DAYS, e DAYS.

* Period representa uma quantidade de tempo em termos de anos, meses e dias.  
  Você pode criar uma instância usando:  
  Period.of(3, 12, 30);

Valores válidos de ChronoUnit para usar com o método get() são:  
DAYS, MONTHS, YEARS.

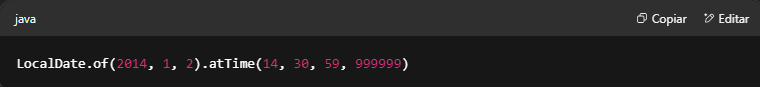
* Duration representa uma quantidade de tempo em termos de segundos e nanossegundos.  
  Você pode criar uma instância usando:  
  Duration.ofSeconds(50, 999999);

Valores válidos de ChronoUnit para usar com o construtor e os métodos get(), plus() e minus() são:  
NANOS, MICROS, MILLIS, SECONDS, MINUTES, HOURS, HALF\_DAYS, e DAYS.

**Autoavaliação**

**1. Quais das seguintes são formas válidas de criar um objeto LocalDate?**  
A. LocalDate.of(2014);  
B. LocalDate.with(2014, 1, 30);  
C. LocalDate.of(2014, 0, 30);  
D. LocalDate.now().plusDays(5);

**2. Dado:**



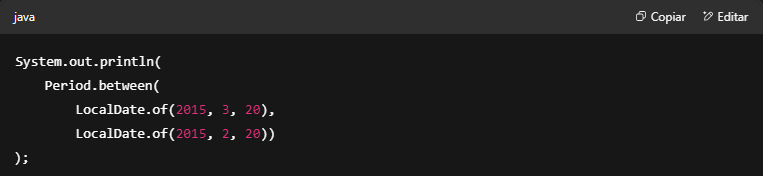
**Qual dos seguintes é o resultado da execução da linha acima?**  
A. Um LocalDate de 2014-01-02  
B. Um LocalTime de 14:30:59:999999  
C. Um LocalDateTime de 2014-01-02 14:30:59:999999  
D. Uma exceção é lançada

**3. Quais das seguintes são valores válidos de ChronoUnit para LocalTime?**  
A. YEAR  
B. NANOS  
C. DAY  
D. HALF\_DAYS

**4. Quais das seguintes afirmações são verdadeiras?**  
A. java.time.Period implementa java.time.temporal.Temporal  
B. java.time.Instant implementa java.time.temporal.Temporal  
C. LocalDate e LocalTime são seguros para threads  
D. LocalDateTime.now() retornará a hora atual no fuso horário UTC

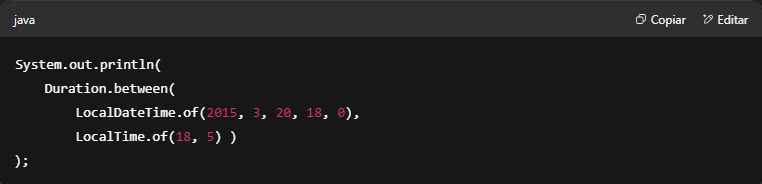
**5. Qual das seguintes é uma forma válida de obter a parte de nanossegundos de um objeto Instant referenciado por i?**  
A. int nanos = i.getNano();  
B. long nanos = i.get(ChronoField.NANOS);  
C. long nanos = i.get(ChronoUnit.NANOS);  
D. int nanos = i.getEpochNano();

**6. Dado:**



**Qual dos seguintes é o resultado da execução da linha acima?**  
A. P29D  
B. P-29D  
C. P1M  
D. P-1M

**7. Dado:**



**Qual dos seguintes é o resultado da execução da linha acima?**  
A. PT5M  
B. PT-5M  
C. PT300S  
D. Uma exceção é lançada

**8. Quais das seguintes são valores válidos de ChronoField para LocalDate?**  
A. DAY\_OF\_WEEK  
B. HOUR\_OF\_DAY  
C. DAY\_OF\_MONTH  
D. MILLI\_OF\_SECOND