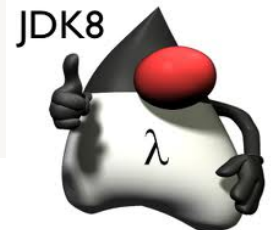




# Java 8 Date & Time API

`package java.time`



PARTE III

- ▶ A classe **Clock** é uma classe abstrata de uso opcional. Uma instância de **Clock** é um objecto temporal que dá acesso ao instante, data e tempo correntes do fuso horário do sistema ou de um outro qualquer fuso horário dado como parâmetro.
- ▶ Caso se aplique no fuso horário do sistema, é equivalente a todos os métodos **now()** implementados nas várias classes que referimos até agora, sendo portanto, nestes casos, irrelevante. É equivalente ao método **System.currentTimeMillis()**; que é o método usado quando invocamos os métodos **now()** ou **now(ZoneId)** das várias classes. Exemplo com **Instant**.

```
Instant agora = Clock.systemDefaultZone().instant();
Instant agoraNow = Instant.now();
System.out.println(agora);
System.out.println(agoraNow);
System.out.println(agora.toEpochMilli());
Instant tokyo = Clock.system(ZoneId.of("Asia/Tokyo")).instant();
System.out.println(tokio);
System.out.println(tokio.toEpochMilli());
System.out.println("Sistema: " + System.currentTimeMillis());
2016-03-22T19:41:20.941Z
2016-03-22T19:41:20.941Z
1458675680941
2016-03-22T19:41:20.946Z
1458675680946
Sistema: 1458675680946
```



- ▶ **Tempo de CPU** ou tempo de processamento (*CPU time/Process time*) é a quantidade de tempo que o CPU consome no processamento das instruções de um programa, **não incluindo os tempos de espera relacionadas com operações de *input/output*.**
- ▶ Por contraste, **intervalo de tempo real** ou tempo de relógio de parede (cf. *Elapsed real time* e *wall clock time*), é o tempo, medido por um relógio normal, passado entre o início e o fim de um dado conjunto de instruções ou programa, **incluindo instruções de *input/output*.**
- ▶ Quando pretendemos medir tempos de computações medimos de facto *elapsed time*. Em computadores de um só processador, o *elapsed time* é em geral igual ou superior ao *CPU time*. Em máquinas com múltiplos processadores fala-se mesmo em *Total CPU time* que é a soma dos tempos de CPU gastos em cada processador.
- ▶ Em Java o método `System.currentTimeMillis()` ; é um dos mais usados para a realização de medidas de tempos de computação. Porém, haverá que ter em atenção a granularidade da medida, ou seja, ao fim de quanto tempo é que o valor é modificado.
- ▶ No caso de `System.currentTimeMillis()` ; a granularidade é maior do que 1 milissegundo pelo que **invocações repetidas do método podem gerar o mesmo resultado várias vezes** e, de repente, estes valores aumentarem de várias dezenas de milissegundos.



- Assim, o método mais rigoroso para medições destes tempos será **System.nanoTime()**; método que foi utilizado na classe **Crono** que usaremos para medir *elapsed time*.

```
// Tempo de escrita de 1 milhão linhas com PrintWriter
```

```
long inicio = System.currentTimeMillis();  
long inicioNano = System.nanoTime();  
Instant instInic = Instant.now();  
Crono.start();  
String linha = "AX6754 9.55 20 X1234 N 11 2";  
try {  
    PrintWriter pw = new PrintWriter("linhas");  
    for(int i = 0; i <= 999999; i++) pw.println(linha);  
    pw.close();  
}  
catch(IOException e) { System.out.println(e.getMessage()); }  
long fim = System.currentTimeMillis();  
long fimNano = System.nanoTime();  
Instant instFinal = Instant.now();  
Crono.stop();  
// TEMPOS  
System.out.println("Em Millis: " + (fim - inicio));  
System.out.println("Em Nanos: " + (fimNano - inicioNano));  
System.out.println("Usando Duration: " + Duration.between(instInic, instFinal).getNano());  
System.out.println("Instantes + between : " + NANOS.between(instInic, instFinal));  
System.out.println("ET crono : " + Crono.print());
```

**Em Millis: 153**

**Em Nanos: 153675185**

**Usando Duration: 153000000**

**Instantes + between : 153000000**

**ET crono : 0.156054451**

- A maioria dos exercícios que realizamos até agora, consistiram em escrevermos código que, dado um ou mais objectos temporais, executavam operações com vista a obter informação a partir de tal ou tais objectos temporais, por exemplo, *meses desde o início desse ano, se a data corresponde ou não a um dado dia da semana, datas de realização de pagamentos a x meses*, etc.;
- **Em múltiplas aplicações comerciais existem datas e tempos muito particulares**, por exemplo relacionadas com contratos e regras, como datas-limite, prazos, períodos de validade, horas de abertura e fecho da bolsa de valores, tempos de voo, etc., a que precisamos de estar atentos e indagar;
- **Porém, o código que escrevemos em cada caso não está encapsulado para ser reutilizado** nos mais diversos contextos. Se o encapsularmos num método de uma qualquer classe, então tal código não poderá ser passado como parâmetro para nenhum contexto.
- **Coloca-se assim o problema de como encapsular código que realiza operações sobre datas e tempos mas de modo a que possa ser passado como parâmetro para um contexto qualquer, por exemplo, numa operação sobre colecções ou *streams*.**
- A interface **TemporalField** (cf. **IsoFields**, **ChroFields**) e o tipo enumerado **ChronoField** fornecem mecanismos para interrogar datas e tempos mas não fornecem encapsulamento e limitam-se a devolver respostas do tipo **long**. Outras interfaces e enumerados têm outros propósitos e foram igualmente usados, cf. **ChronoUnit**, **TemporalAdjuster**, etc.



- ▶ Porém, a maioria das classes data-tempo e auxiliares (com excepção de **Duration** e **Period**), implementam a interface **TemporalAccessor** que tem definido um método **default** que permite realizar queries a tais receptores. Trata-se do método `<R> R query(TemporalQuery<R> tq);` que recebe como parâmetro a formulação do *query* sob a forma de uma **TemporalQuery<R>** e dá como resultado um objecto do tipo **R**. Sabendo implementar **TemporalQuery** temos muitos *queries*.
- ▶ A **@FunctionalInterface TemporalQuery<R>** permite capturar a lógica (algoritmo) da realização da operação que interroga (consulta) uma classe temporal e devolve um resultado do tipo pretendido (**qualquer tipo**). O método abstracto da interface é o método com assinatura **R queryFrom(TemporalAccessor ta);** que aceita um objecto temporal e devolve um resultado de tipo **R**.
- ▶ A interface é genérica e **por ser funcional** pode ser implementada **por uma classe** ou por uma **expressão lambda**.
  - ▶ Adicionalmente, a classe **TemporalQueries** fornece 7 implementações de **TemporalQuery<R>** pré-definidas que vamos usar como primeiros exemplos, pois não temos que as implementar.
  - ▶ Vejamos então os pré-definidos (que não são de grande utilidade) a título de exemplo. Se não forem aplicáveis a um dado objecto devolvem **null**.

<code>static TemporalQuery&lt;Chronology&gt;</code>	<code>chronology()</code> A query for the Chronology.
<code>static TemporalQuery&lt;LocalDate&gt;</code>	<code>localDate()</code> A query for LocalDate returning null if not found.
<code>static TemporalQuery&lt;LocalTime&gt;</code>	<code>localTime()</code> A query for LocalTime returning null if not found.
<code>static TemporalQuery&lt;ZoneOffset&gt;</code>	<code>offset()</code> A query for ZoneOffset returning null if not found.
<code>static TemporalQuery&lt;TemporalUnit&gt;</code>	<code>precision()</code> A query for the smallest supported unit.
<code>static TemporalQuery&lt;ZoneId&gt;</code>	<code>zone()</code> A lenient query for the ZoneId, falling back to the ZoneOffset.
<code>static TemporalQuery&lt;ZoneId&gt;</code>	<code>zoneId()</code> A strict query for the ZoneId.

```
LocalDateTime ldt = LocalDateTime.of(2017, 3, 21, 15, 30, 00);
out.println(ldt.query(TemporalQueries.chronology()));
out.println(ldt.query(TemporalQueries.localDate()));
out.println(ldt.query(TemporalQueries.localTime()));
// com importações
out.println(ldt.query(offset()));
out.println(ldt.query(zoneId()));
out.println(ldt.query(zone()));
out.println(ldt.query(precision()));
```

ISO  
2017-03-21  
15:30  
null  
null  
null  
Nanos

**Padrão 1:** `R = temporal.query(TemporalQuery);`

## ■ Métodos de criação e utilização de TemporalQuery<R>

1) Como com qualquer interface, vamos criar uma classe que implementa a interface e passar uma instância dessa classe para o query.

```
import java.time.temporal.TemporalAccessor;  
import java.time.temporal.TemporalQuery;  
import java.time.DayOfWeek;  
import static java.time.DayOfWeek.MONDAY;  
import static java.time.temporal.ChronoField.DAY_OF_WEEK;  
  
public class Testa_Segunda implements TemporalQuery<Boolean> {  
    // método que redefine o método queryFrom() default  
    @Override  
    public Boolean queryFrom(TemporalAccessor tac) {  
        // return tac.get(DAY_OF_WEEK) == 1;  
        return DayOfWeek.of(tac.get(DAY_OF_WEEK)).equals(MONDAY);  
    }  
}
```

### Utilização:

```
LocalDateTime dataTeste = LocalDateTime.of(2017, 10, 2, 15, 30, 0);  
out.println("É segunda? " + dataTeste.query(new Testa_Segunda()));
```



2) Criar uma classe que não implementa a interface, definir um método de classe com um nome qualquer que satisfaça a assinatura de `queryFrom()` e depois usá-lo numa expressão lambda.

```
import java.time.temporal.TemporalAccessor;  
import java.time.DayOfWeek;  
import static java.time.DayOfWeek.WEDNESDAY;  
import static java.time.temporal.ChronoField.DAY_OF_WEEK;  
  
public class Testa_Quarta {  
    // método que implementa o método queryFrom()  
    public static Boolean e_Quarta(TemporalAccessor tacs) {  
        // return tacs.get(DAY_OF_WEEK) == 3;  
        return DayOfWeek.of(tacs.get(DAY_OF_WEEK)).equals(WEDNESDAY);  
    }  
}
```

### Utilização:

```
LocalDateTime dataTeste = LocalDateTime.of(2017, 10, 2, 15, 30, 0);  
out.println("É quarta? " + dataTeste.query(Testa_Quarta::e_Quarta));
```

---

**Nota:** Em ambas as soluções deveríamos ter testado se o `TemporalAccessor` é compatível com `LocalDate`, por exemplo usando `try { LocalDate ld = LocalDate.from(tacs); } catch(DateTimeException e) { ..... }`



- Vamos agora repescar um dos nossos exercícios práticos e encapsular o código para que possa ser invocado via método **query()** e, assim, tornar-se reutilizável em vários contextos.

```
public class Util_Datas {  
    public static LocalDate Data_10DiasUteisApos(TemporalAccessor tacs) {  
        // DEZ DIAS UTEIS MAIS TARDE  
        LocalDate dataRef = null;  
        try { dataRef = LocalDate.from(tacs); }  
        catch (DateTimeException e) { return null; }  
        int conta = 0;  
        while (conta < 10) {  
            DayOfWeek dia = dataRef.getDayOfWeek();  
            if (! (dia.equals(SATURDAY) || dia.equals(SUNDAY))) conta++;  
            dataRef = dataRef.plus(1, DAYS);  
        }  
        return dataRef;  
    }  
}
```

```
out.println("10 dias úteis depois : " +  
            dataTeste.query(Util_Datas::Data_10DiasUteisApos));
```



from

```
public static LocalDate from(TemporalAccessor temporal)
```

Obtains an instance of `LocalDate` from a temporal object.

This obtains a local date based on the specified temporal. A `TemporalAccessor` represents an arbitrary set of date and time information, which this factory converts to an instance of `LocalDate`.

The conversion uses the `TemporalQueries.localDate()` query, which relies on extracting the `EPOCH_DAY` field.

This method matches the signature of the functional interface `TemporalQuery` allowing it to be used as a query via method reference, `LocalDate::from`.

**Vamos ver como este método está implementado na classe `LocalDate`.**

```
public static LocalDate from(TemporalAccessor temporal) {  
    Objects.requireNonNull(temporal, "temporal is null");    // temporal != null ?  
    LocalDate date = temporal.query(TemporalQueries.localDate());  
    if (date == null) {  
        throw new DateTimeException("Unable to obtain LocalDate from TemporalAccessor: " +  
            temporal + " of type " + temporal.getClass().getName());  
    }  
    return date;  
}
```

**Utilização:**

```
LocalDateTime dataTeste = LocalDateTime.of(2017, 10, 2, 15, 30, 0);  
out.println(" Data " + dataTeste.query(LocalDate::from));
```

► Agora que sabemos criar implementações de **TemporalQuery<R>** resta-nos tomar uma última decisão e de fácil escolha:

- 1) **Vamos continuar a criar uma classe para cada TemporalQuery<R>** ou,
- 2) **Criamos uma classe de utilidades**, por exemplo **Util\_TempQueries**, onde cada *query* é representado por um método de classe (static).

```
public class Util_TempQueries {  
    // Classe de utilidades com implementações de TemporalQuery<R>  
    public static LocalDate Data_10DiasUteisApos(TemporalAccessor tacs) {  
        // DEZ DIAS UTEIS MAIS TARDE  
        LocalDate dataRef = null;  
        try { dataRef = LocalDate.from(tacs); }  
        catch(DateTimeException e) { return null; }  
        int conta = 0;  
        while(conta <= 10) {  
            DayOfWeek dia = dataRef.getDayOfWeek();  
            if(! (dia.equals(SATURDAY) || dia.equals(SUNDAY)))    conta++;  
            else dataRef = dataRef.plus(1, DAYS);  
        }  
        return dataRef;  
    }  
  
    public static Boolean e_Quarta(TemporalAccessor tacs) {  
        // É UMA QUARTA-FEIRA ?  
        LocalDate dataRef = null;  
        try { dataRef = LocalDate.from(tacs); }  
        catch(DateTimeException e) { return null; }  
        return DayOfWeek.of(tacs.get(DAY_OF_WEEK)).equals(WEDNESDAY);  
    }  
    // etc  
}
```

- A criação de `TemporalQuery<R>` usando *inner classes* é possível mas não tem sentido em JAVA8.

```
private static final TemporalQuery<LocalDate> queryComInner =  
    new TemporalQuery<LocalDate>() {  
        @Override  
        public LocalDate queryFrom(TemporalAccessor temporal) {  
            // código .....  
            return .....;  
        }  
    }
```

← Redefinição de `queryFrom()` *on-the-fly*

**Padrão 2:** `R res = TemporalQuery.queryFrom(temporal);`

```
LocalDate ld = queryComInner.queryFrom(LocalDate.now());
```

- ▶ A classe **java.time.format.DateTimeFormatter** providencia um grande conjunto de métodos para formatar e para realizar a verificação sintática (*parsing*) de datas e tempos.
- ▶ As classes de tipo data-tempo possuem os seus métodos próprios para formatação e para *parsing* de datas e/ou tempos, designadamente:

```
format(DateTimeFormatter formatter);  
parse(CharSequence text);  
parse(CharSequence text, DateTimeFormatter formatter);
```

String  
StringBuffer, StringBuilder  
Segment, CharBuffer

- ▶ Porém, alguns destes métodos necessitam de receber como parâmetro uma instância da classe **DateTimeFormatter**, instância essa que representa um dado formatador que irá formatar o objeto temporal segundo um dado tipo ou padrão de formatação. **DateTimeFormatter** possui um grande número de formatadores predefinidos que se apresentam a seguir.



- Vejamos então um exemplo baseado em `LocalDate` que cria um formatador predefinido, cf. `ISO_LOCAL_DATE`, e que produz o respetivo formato sob a forma de uma `String`.

```
LocalDate data = LocalDate.of(2016, 3, 26);
DateTimeFormatter isoLd = DateTimeFormatter.ISO_LOCAL_DATE;
String dataEmTxt = data.format(isoLd);
// Equivalente a data.format(ISO_LOCAL_DATE); com import
System.out.println("ISO LOCAL: " + dataEmTxt);
// ISO LOCAL: 2016-03-26
// Em seguida, faz-se o parsing desta string para mostrar que a
// data original e a data verificada são idênticas.
LocalDate dataVerificada = LocalDate.parse(dataEmTxt, isoLd);
System.out.println(data);
System.out.println(dataVerificada);
// 2016-03-26
// 2016-03-26
```

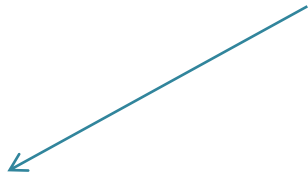
- Para além de `ISO_LOCAL_DATE` existem mais 14 formatadores predefinidos, todos utilizáveis a partir da classe `DateTimeFormatter`, conforme o exemplo acima. Vejamos mais alguns exemplos usando agora uma instância de `LocalDateTime`.



```
LocalDateTime ldTime = LocalDateTime.of(2016, 7, 22, 10, 35);
System.out.println("BASIC: " +
    ldTime.format(DateTimeFormatter.BASIC_ISO_DATE));
System.out.println("ISO DATE: " +
    ldTime.format(DateTimeFormatter.ISO_DATE));
System.out.println("ISO TIME: " +
    ldTime.format(DateTimeFormatter.ISO_TIME));
System.out.println("ISO DATE_TIME: " +
    ldTime.format(DateTimeFormatter.ISO_DATE_TIME));
System.out.println("ISO WEEK_DATE: " +
    ldTime.format(DateTimeFormatter.ISO_WEEK_DATE));
System.out.println("ISO ORDINAL_DATE: " +
    ldTime.format(DateTimeFormatter.ISO_ORDINAL_DATE));
System.out.println("ISO ORDINAL_DATE_TIME: " +
    ldTime.format(DateTimeFormatter.ISO_DATE_TIME));
```

```
// resultados
BASIC: 20160722
ISO DATE: 2016-07-22
ISO TIME: 10:35:00
ISO DATE_TIME: 2016-07-22T10:35:00
ISO WEEK_DATE: 2016-W29-6
ISO ORDINAL_DATE: 2016-203
ISO LOCAL_DATE_TIME: 2016-07-22T10:35:00
```

Formatos devem respeitar campos  
existentes




Exception in thread "main" java.time.temporal.UnsupportedTemporalTypeException: Unsupported field: InstantSeconds



- Formatações específicas podem ser também realizadas a partir de **patterns definidos pelo programador** usando `DateTimeFormatter.ofPattern(CharSequence pattern);`.

Vamos passar a **LocalDateTime** anterior para vários formatos e verificar os resultados.

```
//String dataEmDMA = ldTime.format(DateTimeFormatter.ofPattern("dd MM yyyy"));  
String dataEmDMA = ldTime.format(ofPattern("dd MM yyyy"));  
out.println(dataEmDMA);  
// 22 07 2017  
out.println(ldTime.format(ofPattern("dd MM yyyy hh:mm:ss")));  
// 22 07 2017 10:35:00  
out.println(ldTime.format(ofPattern("dd MM yyyy hh:mm:ss,SSS")));  
// 22 07 2017 10:35:00,000  
out.println(ldTime.format(ofPattern("dd-MM-uu hh:mm:ss")));  
// 22-07-17 10:35:00
```



y	= year	(yy or yyyy)
M	= month	(MM)
d	= day in month	(dd)
h	= hour (0-12)	(hh)
H	= hour (0-23)	(HH)
m	= minute in hour	(mm)
s	= seconds	(ss)
S	= milliseconds	(SSS)
z	= time zone text	(e.g. Pacific Standard Time...)
Z	= time zone, time offset	(e.g. -0800)

- Para a criação de um padrão de formatação (*pattern*) existem portanto caracteres especiais definidos na classe **DateTimeFormatter** bem como regras para a criação do *pattern*. Do exemplo anterior pode inferir-se que **d** significa dia, **M** significa mês e **y** ou **u** significa ano da era, etc. Porém o número de caracteres usados em cada caso pode ter significados distintos. Exemplos:

```
out.println(ldTime.format(ofPattern("dd MMM yyyy")));  
// 22 jul 2017
```

```
out.println(ldTime.format(ofPattern("dd MMMM yyyy, HH:mm:ss")));  
// 22 Julho 2017, 10:35:00
```

- Como se pode verificar pelos exemplos, teremos à nossa disposição uma infinidade de padrões. Um formatador criado a partir de um padrão, desde que associado a um identificador, é imutável e pode ser usado onde quer que o seu contexto de declaração permita.

```
DateTimeFormatter anoAteSegundo = ofPattern("dd MMMM yyyy HH:mm:ss");
```

- O método **DateTimeFormatter.ofLocalizedDate(FormatStyle style)**; cria um formatador que usa o padrão de formatação definido pelo **Locale** do sistema. O parâmetro **java.time.format.FormatStyle** é um tipo enumerado que possui 4 constantes, SHORT, MEDIUM, LONG e FULL, que definem o grau de detalhe de apresentação do objeto temporal.

- No exemplo vamos verificar alguns dados do nosso `java.util.Locale` e usar o método anterior para criar um formatador de datas local usando o estilo **FormatStyle.SHORT** ou apenas **SHORT**.

```
out.println("O meu LOCALE: " + Locale.getDefault());  
out.println("O meu PAÍS: " +  
    Locale.getDefault().getDisplayCountry());  
out.println("A minha Língua/PAÍS: " +  
    Locale.getDefault().getDisplayName());  
out.println("A minha Língua: " +  
    Locale.getDefault().getDisplayLanguage());  
  
out.println(ldTime.format(ofLocalizedDate(SHORT)));  
out.println(ldTime.format(ofLocalizedDate(MEDIUM)));  
out.println(ldTime.format(ofLocalizedDateTime(SHORT)));  
out.println(ldTime.format(ofLocalizedDateTime(MEDIUM)));
```

```
O meu LOCALE: pt_PT  
O meu PAÍS: Portugal  
A minha Língua/PAÍS: português (Portugal)  
A minha Língua/PAÍS: português  
22-07-2017  
22/jul/2017  
22-07-2017 10:35  
22/jul/2017 10:35:00
```

- A expressão `Locale[] locales = Locale.getAvailableLocales();` permitirá aos mais curiosos criar um *array* com todos os `Locale` definidos e fazer os respectivos `toString()` para ficarem a conhecer os seus identificadores.

- Vamos ver um último exemplo semelhante aos anteriores mas usando os métodos

```
DateTimeFormatter withLocale(Locale loc);  
DateTimeFormatter ofPattern(CharSequence pt, Locale loc);
```

que dão como resultado um formatador que formata uma data no padrão "dd.MMMM.uuuu" usando as características de um dado `Locale`.

```
DateTimeFormatter dtFormFranca = ofPattern("dd.MMMM.uuuu").withLocale(Locale.FRENCH);  
DateTimeFormatter dtFormJapao = ofPattern("dd.MMMM.uuuu").withLocale(JAPAN);  
DateTimeFormatter dtFormChina = ofPattern("dd.MMMM.uuuu", CHINA);  
DateTimeFormatter dtFormItalia = ofPattern("dd.MMMM.uuuu").withLocale(ITALIAN);  
DateTimeFormatter dtFormAlemanha = ofPattern("dd.MMMM.uuuu", GERMANY);  
DateTimeFormatter dtFormLocJapan = ofLocalizedDateTime(MEDIUM).withLocale(Japan);
```

```
out.println(ldTime.format(dtFormFranca));  
out.println(ldTime.format(dtFormJapao));  
out.println(ldTime.format(dtFormChina));  
out.println(ldTime.format(dtFormItalia));  
out.println(ldTime.format(dtFormAlemanha));
```

```
22. juillet. 2017  
22. 7月. 2017  
22. 七月. 2017  
22. luglio. 2017  
22. Juli. 2017  
2017/07/22 10:35:00
```



- ▶ Resta-nos agora falar da verificação sintática (*parsing*) de objetos temporais, que se vai basear nas várias formas do método **`parse()`** implementado nas várias classes temporais. Na sua forma mais simples este método recebe uma **`CharSequence`** que é a data-tempo a validar, e devolve um objeto temporal correspondente usando o formato `ISO_LOCAL_DATE`.
- ▶ Um erro de *parsing* gera a exceção **`java.time.format.DateTimeParseException`** que é uma subclasse de **`java.time.DateTimeException`**.
- ▶ Assim, qualquer tentativa de *parsing* deve ser programada dentro de um *try & catch*.

```
try {  
    LocalDate ldISO = LocalDate.parse("2016-01-20");  
    out.println(ldISO);  
}  
catch (DateTimeParseException dtex) { out.println(dtex.getMessage()); }
```

- Se estivermos a fazer o *parsing* dentro de um método, deveremos programar:

```
public static LocalDate strToLocalDate(CharSequence data) {  
    try { return LocalDate.parse(data); }  
    catch (DateTimeParseException ex) { return null; }  
}
```

- Usando o tipo **Optional<T>** de Java8, o código anterior ficaria:

```
public static Optional<LocalDate> strToOptLocalDate(CharSequence data) {  
    try { return Optional.of(LocalDate.parse(data)); }  
    catch (DateTimeParseException ex)  
        { return Optional.empty(); }  
}
```

```
Optional<LocalDate> opLdate = Utils_DataTime.strToOptLocalDate(charData);  
out.println(opLdate.isPresent() ? opLdate.get() : "erro ");
```

- Muitas vezes não iremos realizar o *parsing* tão livremente mas verificar se uma data-tempo satisfaz um dado formato predefinido em `DateTimeFormatter`. Por exemplo, vamos ver se uma data satisfaz o formato `ISO_WEEK_DATE`. Nestes casos precisamos de usar o método,

```
parse(CharSequence text, DateTimeFormatter formatter);
```

das respetivas classes e criar o formatador que servirá de verificador da sintaxe. Vejamos um exemplo bem sucedido.

```
LocalDate ldWeek = LocalDate.parse("2016-W13-1",  
                                   DateTimeFormatter.ISO_WEEK_DATE);  
System.out.println(ldWeek);  
2016-03-28
```

- O exemplo anterior revela que para além da validação da data em tal formato, foi realizada a sua conversão para o formato ISO de `LocalDate`.

```
String ldTimeStr = "2016-04-08 12:30";  
DateTimeFormatter form1 = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm");  
LocalDateTime ldateTime1 = LocalDateTime.parse(ldTimeStr, form1);  
out.println(ldateTime1);  
2016-04-08T12:30
```


- ▶ Antes de Java 8, as classes `Date`, `Calendar` e `TimeZone` e, em especial, a subclasse de `Calendar` designada `GregorianCalendar`, eram as classes que representavam instantes (cf. `Date`), datas e tempos (cf. `GregorianCalendar`) e fusos horários (cf. `TimeZone`).
- ▶ Todas estas classes temporais não abstratas **produziam objetos mutáveis**, isto é, cujo estado interno podia ser modificado por exemplo usando métodos `set`, o que causava grande insegurança em aplicações que usavam múltiplas *threads* de execução.
- ▶ A retrocompatibilidade de Java 8 é assegurada pela criação de alguns métodos que asseguram a conversão de instâncias de `java.util.Date` e `java.util.Calendar` em objetos de `java.time`, designadamente:

```
Calendar.toInstant();  
Date.toInstant();  
GregorianCalendar.toZonedDateTime();  
TimeZone.toZoneId();  
java.sql.Date.toLocalDate();  
java.sql.Time.toLocalTime();  
java.sql.Timestamp.toInstant();
```



***Implicam muito  
refactoring em todo o  
caso***



 O package ***java.time***, por toda a arquitectura e funcionalidade que apresentámos, é unanimemente reconhecido pelos conhecedores de Java como uma das grandes e fundamentais criações (***features***) introduzidas em Java8, em especial para o desenvolvimento de aplicações empresariais e para aplicações móveis.

