**Paquete java.time de Java 8: Fechas y horas**

*Por Diego Silva*  
Publicado en Diciembre 2014

En este artículo veremos sobre un nuevo paquete que existe en Java 8, se trata del paquete [java.time](http://docs.oracle.com/javase/8/docs/api/java/time/package-frame.html). Este paquete es una extensión a las clases java.util.Date y java.util.Calendar que vemos un poco limitado para manejo de fechas, horas y localización.

Las clases definidas en este paquete representan los principales conceptos de fecha - hora, incluyendo instantes, fechas, horas, periodos, zonas de tiempo, etc. Están basados en el sistema de calendario ISO, el cual el calendario mundial *de-facto* que sigue las reglas del calendario Gregoriano

**Enumerados de mes y de día de la semana**   
Existe un enum donde se definen todos los días de la semana. Lo cual tiene sentido hacerlo enum porque siempre habrán siete días de la semana :). Este enum se llama [java.time.DayOfWeek](http://docs.oracle.com/javase/8/docs/api/java/time/DayOfWeek.html)

DayOfWeek lunes = DayOfWeek.MONDAY;

Este enum tiene algunos métodos interesantes que permite manipular días hacía adelante y hacia atrás:

DayOfWeek lunes = DayOfWeek.MONDAY;

System.out.printf("8 días será: %s%n",lunes.plus(8));

System.out.printf("2 días antes fue: %s%n",lunes.minus(2));

Además, con el método getDisplayName() se puede acceder al texto que corresponde a la fecha, dependiendo del Locale actual, o el que definamos. Para mi país probé con esto:

DayOfWeek lunes = DayOfWeek.MONDAY;

Locale l = new Locale("es","PE");

System.out.println("TextStyle.FULL:" + lunes.getDisplayName(TextStyle.FULL, l));

System.out.println("TextStyle.NARROW:" + lunes.getDisplayName(TextStyle.NARROW, l));

System.out.println("TextStyle.SHORT:" + lunes.getDisplayName(TextStyle.SHORT, l));

Deberían probar con Locale l = Locale.KOREA, es muy interesante lo que sale.

Para los meses, existe el enum [java.time.Month](http://docs.oracle.com/javase/8/docs/api/java/time/Month.html) que básicamente hace lo mismo:

Locale l = new Locale("pt"); //probamos con portugues

Month mes = Month.MARCH;

System.out.printf("Dos meses más y será: %s%n", mes.plus(2));

System.out.printf("Hace 1 mes fué: %s%n", mes.minus(1));

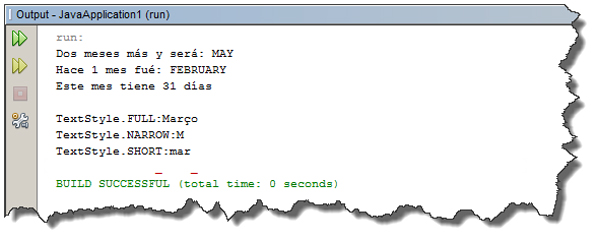
System.out.printf("Este mes tiene %s días %n ", mes.maxLength());

System.out.printf("TextStyle.FULL:%s%n", mes.getDisplayName(TextStyle.FULL, l));

System.out.printf("TextStyle.NARROW:%s%n", mes.getDisplayName(TextStyle.NARROW, l));

System.out.printf("TextStyle.SHORT:%s%n", mes.getDisplayName(TextStyle.SHORT, l));

.. y el resultado sería así:

  
**Clases de fecha**   
Las clases de fecha como el [java.time.LocalDate](http://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) manejan la fecha, pero, a diferencia del java.util.Date, es que es solo trabaja fecha, y no hora. Esto nos permitirá manipular la fecha para registrar fechas específicas como el día de cumpleaños o de matrimonio. Aquí unos ejemplos:

LocalDate date = LocalDate.of(1999, Month.AUGUST, 23);

DayOfWeek dia=date.getDayOfWeek();

System.out.printf("El día que conocí a quien es mi esposa fue el %s y fue un %s%n",date,dia);

Para representar el mes de un año específico, usamos la clase [java.time.YearMonth](http://docs.oracle.com/javase/8/docs/api/java/time/YearMonth.html) y también podemos obtener la cantidad de días de ese mes, sobre todo cuando jugamos con los bisiestos

YearMonth mes = YearMonth.now();

System.out.printf("Este mes es %s y tiene %d días%n", mes, mes.lengthOfMonth());

mes = YearMonth.of(2004, Month.FEBRUARY);

System.out.printf("El mes %s tuvo %d días,%n", mes, mes.lengthOfMonth());

mes = YearMonth.of(2002, Month.FEBRUARY);

System.out.printf("el mes %s tuvo %d días,%n", mes, mes.lengthOfMonth());

mes = YearMonth.of(2000, Month.FEBRUARY);

System.out.printf("el mes %s tuvo %d días%n", mes, mes.lengthOfMonth());

mes = YearMonth.of(1800, Month.FEBRUARY);

System.out.printf("Pero el mes %s tuvo %d días ¿Sabías que no es considerado bisiesto?%n",

mes, mes.lengthOfMonth());

La clase [java.time.MonthDay](http://docs.oracle.com/javase/8/docs/api/java/time/MonthDay.html) representa a un día de un mes en particular, tal como decir que el año nuevo es el 1 de enero.

MonthDay dia=MonthDay.of(Month.FEBRUARY, 29);

System.out.printf("El día %s %s es válido para el año 2010%n",dia,dia.isValidYear(2010)?"":"no");

//la respuesta será NO

Y la clase [java.util.Year](http://docs.oracle.com/javase/8/docs/api/java/time/Year.html) nos permite manipular y conocer sobre un año en específico, sin importar el día o mes.

Year año = Year.now();

System.out.printf("Este año es %s y %s es bisiesto%n", año, año.isLeap() ? "sí" : "no");

**Clase de Hora**   
La clase [java.time.LocalTime](http://docs.oracle.com/javase/8/docs/api/java/time/LocalTime.html) es similar a las otras cosas que comienza con el prefijo Local, pero se centra únicamente en la hora. Esta clase es muy útil para representar horas y tiempos de un día, tales como la hora de inicio de una película o el horario de atención de una biblioteca. Se centra únicamente en la hora de un día cualquiera, pero no en una fecha específica ¿Se entiende? Con el java.util.Date solo podemos manipular la hora de un día de un año en especial, de una zona de horario en especial, pero  con el LocalTime solo nos centramos en la hora en sí, sin importar que día sea. Normalmente lo manipularíamos con una cadena y de ahí hacemos raros algoritmos para saber si esa cadena está dentro de la hora actual.

Aquí un pequeño ejemplo de su uso:

LocalTime justoAhora = LocalTime.now();

System.out.printf("En este momento son las %d horas con %d minutos y %d segundos\n", justoAhora.getHour(),

justoAhora.getMinute(), justoAhora.getSecond());

Como se puede ver, no tiene nada que ver la fecha, solo se manipuló la hora

**La clase de hora/fecha**   
La clase [java.time.LocalDateTime](http://docs.oracle.com/javase/8/docs/api/java/time/LocalDateTime.html) manipula la fecha y la hora sin importar la zona horaria. Esta clase es usada para representar la fecha (año, mes, día) junto con la hora (hora, minuto, segundo, nanosegundo) y es - en efecto - la combinación de LocalDate y LocalTime. Esta clase puede ser usada para especificar un evento, tal como la final de Champions League 2014 en la hora local del evento.

Además del método now que viene en cada clase vista hasta ahora, la clase LocalDateTime tiene varios métodos of (o métodos con prefijo of) que crean una instancia de LocalDateTime. También hay un método from que convierte una instancia de otro formato de tiempo a la instancia LocalDateTime. También hay métodos para agregar y quitar horas, minutos, días, semanas y meses. Aquí muestro algunos ejemplos:

LocalDateTime ahora = LocalDateTime.now();

System.out.printf("La hora es: %s%n", ahora);

LocalDateTime algunDia = LocalDateTime.of(1976, Month.MARCH, 27, 6, 10);

System.out.printf("Yo nací el %s%n", algunDia);

System.out.printf("Hace seis meses fue %s%n", LocalDateTime.now().minusMonths(6));

**Formateador de fecha y hora.**

**https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html**

## Class DateTimeFormatter

* [java.lang.Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html)
  + java.time.format.DateTimeFormatter

public final class DateTimeFormatter

extends [Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html)

Formatter for printing and parsing date-time objects.

This class provides the main application entry point for printing and parsing and provides common implementations of DateTimeFormatter:

* + Using predefined constants, such as [ISO\_LOCAL\_DATE](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#ISO_LOCAL_DATE)
  + Using pattern letters, such as uuuu-MMM-dd
  + Using localized styles, such as long or medium

More complex formatters are provided by [DateTimeFormatterBuilder](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatterBuilder.html).

The main date-time classes provide two methods - one for formatting, format(DateTimeFormatter formatter), and one for parsing, parse(CharSequence text, DateTimeFormatter formatter).

For example:

LocalDate date = LocalDate.now();

String text = date.format(formatter);

LocalDate parsedDate = LocalDate.parse(text, formatter);

In addition to the format, formatters can be created with desired Locale, Chronology, ZoneId, and DecimalStyle.

The [withLocale](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#withLocale-java.util.Locale-) method returns a new formatter that overrides the locale. The locale affects some aspects of formatting and parsing. For example, the [ofLocalizedDate](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#ofLocalizedDate-java.time.format.FormatStyle-) provides a formatter that uses the locale specific date format.

The [withChronology](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#withChronology-java.time.chrono.Chronology-) method returns a new formatter that overrides the chronology. If overridden, the date-time value is converted to the chronology before formatting. During parsing the date-time value is converted to the chronology before it is returned.

The [withZone](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#withZone-java.time.ZoneId-) method returns a new formatter that overrides the zone. If overridden, the date-time value is converted to a ZonedDateTime with the requested ZoneId before formatting. During parsing the ZoneId is applied before the value is returned.

The [withDecimalStyle](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#withDecimalStyle-java.time.format.DecimalStyle-) method returns a new formatter that overrides the [DecimalStyle](https://docs.oracle.com/javase/8/docs/api/java/time/format/DecimalStyle.html). The DecimalStyle symbols are used for formatting and parsing.

Some applications may need to use the older [java.text.Format](https://docs.oracle.com/javase/8/docs/api/java/text/Format.html) class for formatting. The [toFormat()](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#toFormat--) method returns an implementation of java.text.Format.

### Predefined Formatters

| **Formatter** | **Description** | **Example** |
| --- | --- | --- |
| [ofLocalizedDate(dateStyle)](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#ofLocalizedDate-java.time.format.FormatStyle-) | Formatter with date style from the locale | '2011-12-03' |
| [ofLocalizedTime(timeStyle)](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#ofLocalizedTime-java.time.format.FormatStyle-) | Formatter with time style from the locale | '10:15:30' |
| [ofLocalizedDateTime(dateTimeStyle)](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#ofLocalizedDateTime-java.time.format.FormatStyle-) | Formatter with a style for date and time from the locale | '3 Jun 2008 11:05:30' |
| [ofLocalizedDateTime(dateStyle,timeStyle)](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#ofLocalizedDateTime-java.time.format.FormatStyle-) | Formatter with date and time styles from the locale | '3 Jun 2008 11:05' |
| [BASIC\_ISO\_DATE](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#BASIC_ISO_DATE) | Basic ISO date | '20111203' |
| [ISO\_LOCAL\_DATE](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#ISO_LOCAL_DATE) | ISO Local Date | '2011-12-03' |
| [ISO\_OFFSET\_DATE](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#ISO_OFFSET_DATE) | ISO Date with offset | '2011-12-03+01:00' |
| [ISO\_DATE](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#ISO_DATE) | ISO Date with or without offset | '2011-12-03+01:00'; '2011-12-03' |
| [ISO\_LOCAL\_TIME](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#ISO_LOCAL_TIME) | Time without offset | '10:15:30' |
| [ISO\_OFFSET\_TIME](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#ISO_OFFSET_TIME) | Time with offset | '10:15:30+01:00' |
| [ISO\_TIME](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#ISO_TIME) | Time with or without offset | '10:15:30+01:00'; '10:15:30' |
| [ISO\_LOCAL\_DATE\_TIME](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#ISO_LOCAL_DATE_TIME) | ISO Local Date and Time | '2011-12-03T10:15:30' |
| [ISO\_OFFSET\_DATE\_TIME](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#ISO_OFFSET_DATE_TIME) | Date Time with Offset | 2011-12-03T10:15:30+01:00' |
| [ISO\_ZONED\_DATE\_TIME](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#ISO_ZONED_DATE_TIME) | Zoned Date Time | '2011-12-03T10:15:30+01:00[Europe/Paris]' |
| [ISO\_DATE\_TIME](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#ISO_DATE_TIME) | Date and time with ZoneId | '2011-12-03T10:15:30+01:00[Europe/Paris]' |
| [ISO\_ORDINAL\_DATE](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#ISO_ORDINAL_DATE) | Year and day of year | '2012-337' |
| [ISO\_WEEK\_DATE](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#ISO_WEEK_DATE) | Year and Week | 2012-W48-6' |
| [ISO\_INSTANT](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#ISO_INSTANT) | Date and Time of an Instant | '2011-12-03T10:15:30Z' |
| [RFC\_1123\_DATE\_TIME](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#RFC_1123_DATE_TIME) | RFC 1123 / RFC 822 | 'Tue, 3 Jun 2008 11:05:30 GMT' |

### Patterns for Formatting and Parsing

Patterns are based on a simple sequence of letters and symbols. A pattern is used to create a Formatter using the [ofPattern(String)](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#ofPattern-java.lang.String-) and [ofPattern(String, Locale)](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#ofPattern-java.lang.String-java.util.Locale-) methods. For example, "d MMM uuuu" will format 2011-12-03 as '3 Dec 2011'. A formatter created from a pattern can be used as many times as necessary, it is immutable and is thread-safe.

For example:

LocalDate date = LocalDate.now();

DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy MM dd");

String text = date.format(formatter);

LocalDate parsedDate = LocalDate.parse(text, formatter);

All letters 'A' to 'Z' and 'a' to 'z' are reserved as pattern letters. The following pattern letters are defined:

Symbol Meaning Presentation Examples

------ ------- ------------ -------

G era text AD; Anno Domini; A

u year year 2004; 04

y year-of-era year 2004; 04

D day-of-year number 189

M/L month-of-year number/text 7; 07; Jul; July; J

d day-of-month number 10

Q/q quarter-of-year number/text 3; 03; Q3; 3rd quarter

Y week-based-year year 1996; 96

w week-of-week-based-year number 27

W week-of-month number 4

E day-of-week text Tue; Tuesday; T

e/c localized day-of-week number/text 2; 02; Tue; Tuesday; T

F week-of-month number 3

a am-pm-of-day text PM

h clock-hour-of-am-pm (1-12) number 12

K hour-of-am-pm (0-11) number 0

k clock-hour-of-am-pm (1-24) number 0

H hour-of-day (0-23) number 0

m minute-of-hour number 30

s second-of-minute number 55

S fraction-of-second fraction 978

A milli-of-day number 1234

n nano-of-second number 987654321

N nano-of-day number 1234000000

V time-zone ID zone-id America/Los\_Angeles; Z; -08:30

z time-zone name zone-name Pacific Standard Time; PST

O localized zone-offset offset-O GMT+8; GMT+08:00; UTC-08:00;

X zone-offset 'Z' for zero offset-X Z; -08; -0830; -08:30; -083015; -08:30:15;

x zone-offset offset-x +0000; -08; -0830; -08:30; -083015; -08:30:15;

Z zone-offset offset-Z +0000; -0800; -08:00;

p pad next pad modifier 1

' escape for text delimiter

'' single quote literal '

[ optional section start

] optional section end

# reserved for future use

{ reserved for future use

} reserved for future use

The count of pattern letters determines the format.

**Text**: The text style is determined based on the number of pattern letters used. Less than 4 pattern letters will use the [short form](https://docs.oracle.com/javase/8/docs/api/java/time/format/TextStyle.html#SHORT). Exactly 4 pattern letters will use the [full form](https://docs.oracle.com/javase/8/docs/api/java/time/format/TextStyle.html#FULL). Exactly 5 pattern letters will use the [narrow form](https://docs.oracle.com/javase/8/docs/api/java/time/format/TextStyle.html#NARROW). Pattern letters 'L', 'c', and 'q' specify the stand-alone form of the text styles.

**Number**: If the count of letters is one, then the value is output using the minimum number of digits and without padding. Otherwise, the count of digits is used as the width of the output field, with the value zero-padded as necessary. The following pattern letters have constraints on the count of letters. Only one letter of 'c' and 'F' can be specified. Up to two letters of 'd', 'H', 'h', 'K', 'k', 'm', and 's' can be specified. Up to three letters of 'D' can be specified.

**Number/Text**: If the count of pattern letters is 3 or greater, use the Text rules above. Otherwise use the Number rules above.

**Fraction**: Outputs the nano-of-second field as a fraction-of-second. The nano-of-second value has nine digits, thus the count of pattern letters is from 1 to 9. If it is less than 9, then the nano-of-second value is truncated, with only the most significant digits being output.

**Year**: The count of letters determines the minimum field width below which padding is used. If the count of letters is two, then a [reduced](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatterBuilder.html#appendValueReduced-java.time.temporal.TemporalField-int-int-int-) two digit form is used. For printing, this outputs the rightmost two digits. For parsing, this will parse using the base value of 2000, resulting in a year within the range 2000 to 2099 inclusive. If the count of letters is less than four (but not two), then the sign is only output for negative years as per [SignStyle.NORMAL](https://docs.oracle.com/javase/8/docs/api/java/time/format/SignStyle.html#NORMAL). Otherwise, the sign is output if the pad width is exceeded, as per [SignStyle.EXCEEDS\_PAD](https://docs.oracle.com/javase/8/docs/api/java/time/format/SignStyle.html#EXCEEDS_PAD).

**ZoneId**: This outputs the time-zone ID, such as 'Europe/Paris'. If the count of letters is two, then the time-zone ID is output. Any other count of letters throws IllegalArgumentException.

**Zone names**: This outputs the display name of the time-zone ID. If the count of letters is one, two or three, then the short name is output. If the count of letters is four, then the full name is output. Five or more letters throws IllegalArgumentException.

**Offset X and x**: This formats the offset based on the number of pattern letters. One letter outputs just the hour, such as '+01', unless the minute is non-zero in which case the minute is also output, such as '+0130'. Two letters outputs the hour and minute, without a colon, such as '+0130'. Three letters outputs the hour and minute, with a colon, such as '+01:30'. Four letters outputs the hour and minute and optional second, without a colon, such as '+013015'. Five letters outputs the hour and minute and optional second, with a colon, such as '+01:30:15'. Six or more letters throws IllegalArgumentException. Pattern letter 'X' (upper case) will output 'Z' when the offset to be output would be zero, whereas pattern letter 'x' (lower case) will output '+00', '+0000', or '+00:00'.

**Offset O**: This formats the localized offset based on the number of pattern letters. One letter outputs the [short](https://docs.oracle.com/javase/8/docs/api/java/time/format/TextStyle.html#SHORT) form of the localized offset, which is localized offset text, such as 'GMT', with hour without leading zero, optional 2-digit minute and second if non-zero, and colon, for example 'GMT+8'. Four letters outputs the [full](https://docs.oracle.com/javase/8/docs/api/java/time/format/TextStyle.html#FULL) form, which is localized offset text, such as 'GMT, with 2-digit hour and minute field, optional second field if non-zero, and colon, for example 'GMT+08:00'. Any other count of letters throws IllegalArgumentException.

**Offset Z**: This formats the offset based on the number of pattern letters. One, two or three letters outputs the hour and minute, without a colon, such as '+0130'. The output will be '+0000' when the offset is zero. Four letters outputs the [full](https://docs.oracle.com/javase/8/docs/api/java/time/format/TextStyle.html#FULL) form of localized offset, equivalent to four letters of Offset-O. The output will be the corresponding localized offset text if the offset is zero. Five letters outputs the hour, minute, with optional second if non-zero, with colon. It outputs 'Z' if the offset is zero. Six or more letters throws IllegalArgumentException.

**Optional section**: The optional section markers work exactly like calling [DateTimeFormatterBuilder.optionalStart()](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatterBuilder.html#optionalStart--) and [DateTimeFormatterBuilder.optionalEnd()](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatterBuilder.html#optionalEnd--).

**Pad modifier**: Modifies the pattern that immediately follows to be padded with spaces. The pad width is determined by the number of pattern letters. This is the same as calling [DateTimeFormatterBuilder.padNext(int)](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatterBuilder.html#padNext-int-).

For example, 'ppH' outputs the hour-of-day padded on the left with spaces to a width of 2.

Any unrecognized letter is an error. Any non-letter character, other than '[', ']', '{', '}', '#' and the single quote will be output directly. Despite this, it is recommended to use single quotes around all characters that you want to output directly to ensure that future changes do not break your application.

### Resolving

Parsing is implemented as a two-phase operation. First, the text is parsed using the layout defined by the formatter, producing a Map of field to value, a ZoneId and a Chronology. Second, the parsed data is resolved, by validating, combining and simplifying the various fields into more useful ones.

Five parsing methods are supplied by this class. Four of these perform both the parse and resolve phases. The fifth method, [parseUnresolved(CharSequence, ParsePosition)](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#parseUnresolved-java.lang.CharSequence-java.text.ParsePosition-), only performs the first phase, leaving the result unresolved. As such, it is essentially a low-level operation.

The resolve phase is controlled by two parameters, set on this class.

The [ResolverStyle](https://docs.oracle.com/javase/8/docs/api/java/time/format/ResolverStyle.html) is an enum that offers three different approaches, strict, smart and lenient. The smart option is the default. It can be set using [withResolverStyle(ResolverStyle)](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#withResolverStyle-java.time.format.ResolverStyle-).

The [withResolverFields(TemporalField...)](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#withResolverFields-java.time.temporal.TemporalField...-) parameter allows the set of fields that will be resolved to be filtered before resolving starts. For example, if the formatter has parsed a year, month, day-of-month and day-of-year, then there are two approaches to resolve a date: (year + month + day-of-month) and (year + day-of-year). The resolver fields allows one of the two approaches to be selected. If no resolver fields are set then both approaches must result in the same date.

Resolving separate fields to form a complete date and time is a complex process with behaviour distributed across a number of classes. It follows these steps:

* + The chronology is determined. The chronology of the result is either the chronology that was parsed, or if no chronology was parsed, it is the chronology set on this class, or if that is null, it is IsoChronology.
  + The ChronoField date fields are resolved. This is achieved using [Chronology.resolveDate(Map, ResolverStyle)](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/Chronology.html#resolveDate-java.util.Map-java.time.format.ResolverStyle-). Documentation about field resolution is located in the implementation of Chronology.
  + The ChronoField time fields are resolved. This is documented on [ChronoField](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/ChronoField.html) and is the same for all chronologies.
  + Any fields that are not ChronoField are processed. This is achieved using [TemporalField.resolve(Map, TemporalAccessor, ResolverStyle)](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalField.html#resolve-java.util.Map-java.time.temporal.TemporalAccessor-java.time.format.ResolverStyle-). Documentation about field resolution is located in the implementation of TemporalField.
  + The ChronoField date and time fields are re-resolved. This allows fields in step four to produce ChronoField values and have them be processed into dates and times.
  + A LocalTime is formed if there is at least an hour-of-day available. This involves providing default values for minute, second and fraction of second.
  + Any remaining unresolved fields are cross-checked against any date and/or time that was resolved. Thus, an earlier stage would resolve (year + month + day-of-month) to a date, and this stage would check that day-of-week was valid for the date.
  + If an [excess number of days](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#parsedExcessDays--) was parsed then it is added to the date if a date is available.

Implementation Requirements:

This class is immutable and thread-safe.

Since:

1.8