[SKIP NAVIGATION LINKS](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#skip.navbar.top)

* [OVERVIEW](https://docs.oracle.com/javase/8/docs/api/overview-summary.html)
* [PACKAGE](https://docs.oracle.com/javase/8/docs/api/java/time/package-summary.html)
* CLASS
* [USE](https://docs.oracle.com/javase/8/docs/api/java/time/class-use/LocalDate.html)
* [TREE](https://docs.oracle.com/javase/8/docs/api/java/time/package-tree.html)
* [DEPRECATED](https://docs.oracle.com/javase/8/docs/api/deprecated-list.html)
* [INDEX](https://docs.oracle.com/javase/8/docs/api/index-files/index-1.html)
* [HELP](https://docs.oracle.com/javase/8/docs/api/help-doc.html)

**Java™ Platform  
Standard Ed. 8**

* [**PREV CLASS**](https://docs.oracle.com/javase/8/docs/api/java/time/Instant.html)
* [**NEXT CLASS**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDateTime.html)
* [FRAMES](https://docs.oracle.com/javase/8/docs/api/index.html?java/time/LocalDate.html)
* [NO FRAMES](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html)
* [ALL CLASSES](https://docs.oracle.com/javase/8/docs/api/allclasses-noframe.html)
* SUMMARY:
* NESTED |
* [FIELD](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#field.summary) |
* CONSTR |
* [METHOD](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#method.summary)
* DETAIL:
* [FIELD](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#field.detail) |
* CONSTR |
* [METHOD](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#method.detail)

compact1, compact2, compact3

java.time

## Class LocalDate

* [java.lang.Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html)
  + java.time.LocalDate
* **All Implemented Interfaces:**

[Serializable](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html), [Comparable](https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html)<[ChronoLocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html)>, [ChronoLocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html), [Temporal](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/Temporal.html), [TemporalAccessor](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalAccessor.html), [TemporalAdjuster](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalAdjuster.html)

public final class **LocalDate**

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

implements [Temporal](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/Temporal.html), [TemporalAdjuster](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalAdjuster.html), [ChronoLocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html), [Serializable](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html)

A date without a time-zone in the ISO-8601 calendar system, such as 2007-12-03.

LocalDate is an immutable date-time object that represents a date, often viewed as year-month-day. Other date fields, such as day-of-year, day-of-week and week-of-year, can also be accessed. For example, the value "2nd October 2007" can be stored in a LocalDate.

This class does not store or represent a time or time-zone. Instead, it is a description of the date, as used for birthdays. It cannot represent an instant on the time-line without additional information such as an offset or time-zone.

The ISO-8601 calendar system is the modern civil calendar system used today in most of the world. It is equivalent to the proleptic Gregorian calendar system, in which today's rules for leap years are applied for all time. For most applications written today, the ISO-8601 rules are entirely suitable. However, any application that makes use of historical dates, and requires them to be accurate will find the ISO-8601 approach unsuitable.

This is a [value-based](https://docs.oracle.com/javase/8/docs/api/java/lang/doc-files/ValueBased.html) class; use of identity-sensitive operations (including reference equality (==), identity hash code, or synchronization) on instances of LocalDate may have unpredictable results and should be avoided. The equals method should be used for comparisons.

**Implementation Requirements:**

This class is immutable and thread-safe.

**Since:**

1.8

**See Also:**

[Serialized Form](https://docs.oracle.com/javase/8/docs/api/serialized-form.html#java.time.LocalDate)

### *Field Summary*

|  |  |
| --- | --- |
| **Fields** | |
| **Modifier and Type** | **Field and Description** |
| static [**LocalDate**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) | [**MAX**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#MAX)  The maximum supported LocalDate, '+999999999-12-31'. |
| static [**LocalDate**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) | [**MIN**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#MIN)  The minimum supported LocalDate, '-999999999-01-01'. |

### *Method Summary*

|  |  |
| --- | --- |
| **All Methods**[**Static Methods**](javascript:show(1);)[**Instance Methods**](javascript:show(2);)[**Concrete Methods**](javascript:show(8);) | |
| **Modifier and Type** | **Method and Description** |
| [**Temporal**](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/Temporal.html) | [**adjustInto**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#adjustInto-java.time.temporal.Temporal-)([**Temporal**](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/Temporal.html) temporal)  Adjusts the specified temporal object to have the same date as this object. |
| [**LocalDateTime**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDateTime.html) | [**atStartOfDay**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#atStartOfDay--)()  Combines this date with the time of midnight to create a LocalDateTime at the start of this date. |
| [**ZonedDateTime**](https://docs.oracle.com/javase/8/docs/api/java/time/ZonedDateTime.html) | [**atStartOfDay**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#atStartOfDay-java.time.ZoneId-)([**ZoneId**](https://docs.oracle.com/javase/8/docs/api/java/time/ZoneId.html) zone)  Returns a zoned date-time from this date at the earliest valid time according to the rules in the time-zone. |
| [**LocalDateTime**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDateTime.html) | [**atTime**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#atTime-int-int-)(int hour, int minute)  Combines this date with a time to create a LocalDateTime. |
| [**LocalDateTime**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDateTime.html) | [**atTime**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#atTime-int-int-int-)(int hour, int minute, int second)  Combines this date with a time to create a LocalDateTime. |
| [**LocalDateTime**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDateTime.html) | [**atTime**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#atTime-int-int-int-int-)(int hour, int minute, int second, int nanoOfSecond)  Combines this date with a time to create a LocalDateTime. |
| [**LocalDateTime**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDateTime.html) | [**atTime**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#atTime-java.time.LocalTime-)([**LocalTime**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalTime.html) time)  Combines this date with a time to create a LocalDateTime. |
| [**OffsetDateTime**](https://docs.oracle.com/javase/8/docs/api/java/time/OffsetDateTime.html) | [**atTime**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#atTime-java.time.OffsetTime-)([**OffsetTime**](https://docs.oracle.com/javase/8/docs/api/java/time/OffsetTime.html) time)  Combines this date with an offset time to create an OffsetDateTime. |
| int | [**compareTo**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#compareTo-java.time.chrono.ChronoLocalDate-)([**ChronoLocalDate**](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html) other)  Compares this date to another date. |
| boolean | [**equals**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#equals-java.lang.Object-)([**Object**](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html) obj)  Checks if this date is equal to another date. |
| [**String**](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html) | [**format**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#format-java.time.format.DateTimeFormatter-)([**DateTimeFormatter**](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html) formatter)  Formats this date using the specified formatter. |
| static [**LocalDate**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) | [**from**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#from-java.time.temporal.TemporalAccessor-)([**TemporalAccessor**](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalAccessor.html) temporal)  Obtains an instance of LocalDate from a temporal object. |
| int | [**get**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#get-java.time.temporal.TemporalField-)([**TemporalField**](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalField.html) field)  Gets the value of the specified field from this date as an int. |
| [**IsoChronology**](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/IsoChronology.html) | [**getChronology**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#getChronology--)()  Gets the chronology of this date, which is the ISO calendar system. |
| int | [**getDayOfMonth**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#getDayOfMonth--)()  Gets the day-of-month field. |
| [**DayOfWeek**](https://docs.oracle.com/javase/8/docs/api/java/time/DayOfWeek.html) | [**getDayOfWeek**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#getDayOfWeek--)()  Gets the day-of-week field, which is an enum DayOfWeek. |
| int | [**getDayOfYear**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#getDayOfYear--)()  Gets the day-of-year field. |
| [**Era**](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/Era.html) | [**getEra**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#getEra--)()  Gets the era applicable at this date. |
| long | [**getLong**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#getLong-java.time.temporal.TemporalField-)([**TemporalField**](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalField.html) field)  Gets the value of the specified field from this date as a long. |
| [**Month**](https://docs.oracle.com/javase/8/docs/api/java/time/Month.html) | [**getMonth**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#getMonth--)()  Gets the month-of-year field using the Month enum. |
| int | [**getMonthValue**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#getMonthValue--)()  Gets the month-of-year field from 1 to 12. |
| int | [**getYear**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#getYear--)()  Gets the year field. |
| int | [**hashCode**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#hashCode--)()  A hash code for this date. |
| boolean | [**isAfter**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#isAfter-java.time.chrono.ChronoLocalDate-)([**ChronoLocalDate**](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html) other)  Checks if this date is after the specified date. |
| boolean | [**isBefore**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#isBefore-java.time.chrono.ChronoLocalDate-)([**ChronoLocalDate**](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html) other)  Checks if this date is before the specified date. |
| boolean | [**isEqual**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#isEqual-java.time.chrono.ChronoLocalDate-)([**ChronoLocalDate**](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html) other)  Checks if this date is equal to the specified date. |
| boolean | [**isLeapYear**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#isLeapYear--)()  Checks if the year is a leap year, according to the ISO proleptic calendar system rules. |
| boolean | [**isSupported**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#isSupported-java.time.temporal.TemporalField-)([**TemporalField**](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalField.html) field)  Checks if the specified field is supported. |
| boolean | [**isSupported**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#isSupported-java.time.temporal.TemporalUnit-)([**TemporalUnit**](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalUnit.html) unit)  Checks if the specified unit is supported. |
| int | [**lengthOfMonth**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#lengthOfMonth--)()  Returns the length of the month represented by this date. |
| int | [**lengthOfYear**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#lengthOfYear--)()  Returns the length of the year represented by this date. |
| [**LocalDate**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) | [**minus**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#minus-long-java.time.temporal.TemporalUnit-)(long amountToSubtract, [**TemporalUnit**](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalUnit.html) unit)  Returns a copy of this date with the specified amount subtracted. |
| [**LocalDate**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) | [**minus**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#minus-java.time.temporal.TemporalAmount-)([**TemporalAmount**](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalAmount.html) amountToSubtract)  Returns a copy of this date with the specified amount subtracted. |
| [**LocalDate**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) | [**minusDays**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#minusDays-long-)(long daysToSubtract)  Returns a copy of this LocalDate with the specified number of days subtracted. |
| [**LocalDate**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) | [**minusMonths**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#minusMonths-long-)(long monthsToSubtract)  Returns a copy of this LocalDate with the specified number of months subtracted. |
| [**LocalDate**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) | [**minusWeeks**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#minusWeeks-long-)(long weeksToSubtract)  Returns a copy of this LocalDate with the specified number of weeks subtracted. |
| [**LocalDate**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) | [**minusYears**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#minusYears-long-)(long yearsToSubtract)  Returns a copy of this LocalDate with the specified number of years subtracted. |
| static [**LocalDate**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) | [**now**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#now--)()  Obtains the current date from the system clock in the default time-zone. |
| static [**LocalDate**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) | [**now**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#now-java.time.Clock-)([**Clock**](https://docs.oracle.com/javase/8/docs/api/java/time/Clock.html) clock)  Obtains the current date from the specified clock. |
| static [**LocalDate**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) | [**now**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#now-java.time.ZoneId-)([**ZoneId**](https://docs.oracle.com/javase/8/docs/api/java/time/ZoneId.html) zone)  Obtains the current date from the system clock in the specified time-zone. |
| static [**LocalDate**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) | [**of**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#of-int-int-int-)(int year, int month, int dayOfMonth)  Obtains an instance of LocalDate from a year, month and day. |
| static [**LocalDate**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) | [**of**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#of-int-java.time.Month-int-)(int year, [**Month**](https://docs.oracle.com/javase/8/docs/api/java/time/Month.html) month, int dayOfMonth)  Obtains an instance of LocalDate from a year, month and day. |
| static [**LocalDate**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) | [**ofEpochDay**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#ofEpochDay-long-)(long epochDay)  Obtains an instance of LocalDate from the epoch day count. |
| static [**LocalDate**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) | [**ofYearDay**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#ofYearDay-int-int-)(int year, int dayOfYear)  Obtains an instance of LocalDate from a year and day-of-year. |
| static [**LocalDate**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) | [**parse**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#parse-java.lang.CharSequence-)([**CharSequence**](https://docs.oracle.com/javase/8/docs/api/java/lang/CharSequence.html) text)  Obtains an instance of LocalDate from a text string such as 2007-12-03. |
| static [**LocalDate**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) | [**parse**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#parse-java.lang.CharSequence-java.time.format.DateTimeFormatter-)([**CharSequence**](https://docs.oracle.com/javase/8/docs/api/java/lang/CharSequence.html) text, [**DateTimeFormatter**](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html) formatter)  Obtains an instance of LocalDate from a text string using a specific formatter. |
| [**LocalDate**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) | [**plus**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#plus-long-java.time.temporal.TemporalUnit-)(long amountToAdd, [**TemporalUnit**](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalUnit.html) unit)  Returns a copy of this date with the specified amount added. |
| [**LocalDate**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) | [**plus**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#plus-java.time.temporal.TemporalAmount-)([**TemporalAmount**](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalAmount.html) amountToAdd)  Returns a copy of this date with the specified amount added. |
| [**LocalDate**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) | [**plusDays**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#plusDays-long-)(long daysToAdd)  Returns a copy of this LocalDate with the specified number of days added. |
| [**LocalDate**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) | [**plusMonths**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#plusMonths-long-)(long monthsToAdd)  Returns a copy of this LocalDate with the specified number of months added. |
| [**LocalDate**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) | [**plusWeeks**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#plusWeeks-long-)(long weeksToAdd)  Returns a copy of this LocalDate with the specified number of weeks added. |
| [**LocalDate**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) | [**plusYears**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#plusYears-long-)(long yearsToAdd)  Returns a copy of this LocalDate with the specified number of years added. |
| <R> R | [**query**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#query-java.time.temporal.TemporalQuery-)([**TemporalQuery**](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalQuery.html)<R> query)  Queries this date using the specified query. |
| [**ValueRange**](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/ValueRange.html) | [**range**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#range-java.time.temporal.TemporalField-)([**TemporalField**](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalField.html) field)  Gets the range of valid values for the specified field. |
| long | [**toEpochDay**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#toEpochDay--)()  Converts this date to the Epoch Day. |
| [**String**](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html) | [**toString**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#toString--)()  Outputs this date as a String, such as 2007-12-03. |
| [**Period**](https://docs.oracle.com/javase/8/docs/api/java/time/Period.html) | [**until**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#until-java.time.chrono.ChronoLocalDate-)([**ChronoLocalDate**](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html) endDateExclusive)  Calculates the period between this date and another date as a Period. |
| long | [**until**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#until-java.time.temporal.Temporal-java.time.temporal.TemporalUnit-)([**Temporal**](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/Temporal.html) endExclusive, [**TemporalUnit**](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalUnit.html) unit)  Calculates the amount of time until another date in terms of the specified unit. |
| [**LocalDate**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) | [**with**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#with-java.time.temporal.TemporalAdjuster-)([**TemporalAdjuster**](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalAdjuster.html) adjuster)  Returns an adjusted copy of this date. |
| [**LocalDate**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) | [**with**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#with-java.time.temporal.TemporalField-long-)([**TemporalField**](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalField.html) field, long newValue)  Returns a copy of this date with the specified field set to a new value. |
| [**LocalDate**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) | [**withDayOfMonth**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#withDayOfMonth-int-)(int dayOfMonth)  Returns a copy of this LocalDate with the day-of-month altered. |
| [**LocalDate**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) | [**withDayOfYear**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#withDayOfYear-int-)(int dayOfYear)  Returns a copy of this LocalDate with the day-of-year altered. |
| [**LocalDate**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) | [**withMonth**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#withMonth-int-)(int month)  Returns a copy of this LocalDate with the month-of-year altered. |
| [**LocalDate**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) | [**withYear**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#withYear-int-)(int year)  Returns a copy of this LocalDate with the year altered. |

### Methods inherited from class java.lang.[Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html)

[clone](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#clone--), [finalize](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#finalize--), [getClass](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#getClass--), [notify](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#notify--), [notifyAll](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#notifyAll--), [wait](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#wait--), [wait](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#wait-long-), [wait](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#wait-long-int-)

### Methods inherited from interface java.time.chrono.[ChronoLocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html)

[timeLineOrder](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html#timeLineOrder--)

### *Field Detail*

#### MIN

public static final [LocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) MIN

The minimum supported LocalDate, '-999999999-01-01'. This could be used by an application as a "far past" date.

#### MAX

public static final [LocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) MAX

The maximum supported LocalDate, '+999999999-12-31'. This could be used by an application as a "far future" date.

### *Method Detail*

#### now

public static [LocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) now()

Obtains the current date from the system clock in the default time-zone.

This will query the [system clock](https://docs.oracle.com/javase/8/docs/api/java/time/Clock.html#systemDefaultZone--) in the default time-zone to obtain the current date.

Using this method will prevent the ability to use an alternate clock for testing because the clock is hard-coded.

**Returns:**

the current date using the system clock and default time-zone, not null

#### now

public static [LocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) now([ZoneId](https://docs.oracle.com/javase/8/docs/api/java/time/ZoneId.html) zone)

Obtains the current date from the system clock in the specified time-zone.

This will query the [system clock](https://docs.oracle.com/javase/8/docs/api/java/time/Clock.html#system-java.time.ZoneId-) to obtain the current date. Specifying the time-zone avoids dependence on the default time-zone.

Using this method will prevent the ability to use an alternate clock for testing because the clock is hard-coded.

**Parameters:**

zone - the zone ID to use, not null

**Returns:**

the current date using the system clock, not null

#### now

public static [LocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) now([Clock](https://docs.oracle.com/javase/8/docs/api/java/time/Clock.html) clock)

Obtains the current date from the specified clock.

This will query the specified clock to obtain the current date - today. Using this method allows the use of an alternate clock for testing. The alternate clock may be introduced using [dependency injection](https://docs.oracle.com/javase/8/docs/api/java/time/Clock.html).

**Parameters:**

clock - the clock to use, not null

**Returns:**

the current date, not null

#### of

* + - public static [LocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) of(int year,
    - [Month](https://docs.oracle.com/javase/8/docs/api/java/time/Month.html) month,

int dayOfMonth)

Obtains an instance of LocalDate from a year, month and day.

This returns a LocalDate with the specified year, month and day-of-month. The day must be valid for the year and month, otherwise an exception will be thrown.

**Parameters:**

year - the year to represent, from MIN\_YEAR to MAX\_YEAR

month - the month-of-year to represent, not null

dayOfMonth - the day-of-month to represent, from 1 to 31

**Returns:**

the local date, not null

**Throws:**

[DateTimeException](https://docs.oracle.com/javase/8/docs/api/java/time/DateTimeException.html) - if the value of any field is out of range, or if the day-of-month is invalid for the month-year

#### of

* + - public static [LocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) of(int year,
    - int month,

int dayOfMonth)

Obtains an instance of LocalDate from a year, month and day.

This returns a LocalDate with the specified year, month and day-of-month. The day must be valid for the year and month, otherwise an exception will be thrown.

**Parameters:**

year - the year to represent, from MIN\_YEAR to MAX\_YEAR

month - the month-of-year to represent, from 1 (January) to 12 (December)

dayOfMonth - the day-of-month to represent, from 1 to 31

**Returns:**

the local date, not null

**Throws:**

[DateTimeException](https://docs.oracle.com/javase/8/docs/api/java/time/DateTimeException.html) - if the value of any field is out of range, or if the day-of-month is invalid for the month-year

#### ofYearDay

* + - public static [LocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) ofYearDay(int year,

int dayOfYear)

Obtains an instance of LocalDate from a year and day-of-year.

This returns a LocalDate with the specified year and day-of-year. The day-of-year must be valid for the year, otherwise an exception will be thrown.

**Parameters:**

year - the year to represent, from MIN\_YEAR to MAX\_YEAR

dayOfYear - the day-of-year to represent, from 1 to 366

**Returns:**

the local date, not null

**Throws:**

[DateTimeException](https://docs.oracle.com/javase/8/docs/api/java/time/DateTimeException.html) - if the value of any field is out of range, or if the day-of-year is invalid for the year

#### ofEpochDay

public static [LocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) ofEpochDay(long epochDay)

Obtains an instance of LocalDate from the epoch day count.

This returns a LocalDate with the specified epoch-day. The [EPOCH\_DAY](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/ChronoField.html#EPOCH_DAY) is a simple incrementing count of days where day 0 is 1970-01-01. Negative numbers represent earlier days.

**Parameters:**

epochDay - the Epoch Day to convert, based on the epoch 1970-01-01

**Returns:**

the local date, not null

**Throws:**

[DateTimeException](https://docs.oracle.com/javase/8/docs/api/java/time/DateTimeException.html) - if the epoch day exceeds the supported date range

#### from

public static [LocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) from([TemporalAccessor](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalAccessor.html) 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()](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalQueries.html#localDate--) query, which relies on extracting the [EPOCH\_DAY](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/ChronoField.html#EPOCH_DAY) field.

This method matches the signature of the functional interface [TemporalQuery](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalQuery.html) allowing it to be used as a query via method reference, LocalDate::from.

**Specified by:**

[from](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html#from-java.time.temporal.TemporalAccessor-) in interface [ChronoLocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html)

**Parameters:**

temporal - the temporal object to convert, not null

**Returns:**

the local date, not null

**Throws:**

[DateTimeException](https://docs.oracle.com/javase/8/docs/api/java/time/DateTimeException.html) - if unable to convert to a LocalDate

**See Also:**

[Chronology.date(TemporalAccessor)](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/Chronology.html#date-java.time.temporal.TemporalAccessor-)

#### parse

public static [LocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) parse([CharSequence](https://docs.oracle.com/javase/8/docs/api/java/lang/CharSequence.html) text)

Obtains an instance of LocalDate from a text string such as 2007-12-03.

The string must represent a valid date and is parsed using [DateTimeFormatter.ISO\_LOCAL\_DATE](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#ISO_LOCAL_DATE).

**Parameters:**

text - the text to parse such as "2007-12-03", not null

**Returns:**

the parsed local date, not null

**Throws:**

[DateTimeParseException](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeParseException.html) - if the text cannot be parsed

#### parse

* + - public static [LocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) parse([CharSequence](https://docs.oracle.com/javase/8/docs/api/java/lang/CharSequence.html) text,

[DateTimeFormatter](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html) formatter)

Obtains an instance of LocalDate from a text string using a specific formatter.

The text is parsed using the formatter, returning a date.

**Parameters:**

text - the text to parse, not null

formatter - the formatter to use, not null

**Returns:**

the parsed local date, not null

**Throws:**

[DateTimeParseException](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeParseException.html) - if the text cannot be parsed

#### isSupported

public boolean isSupported([TemporalField](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalField.html) field)

Checks if the specified field is supported.

This checks if this date can be queried for the specified field. If false, then calling the [range](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#range-java.time.temporal.TemporalField-), [get](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#get-java.time.temporal.TemporalField-) and [with(TemporalField, long)](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#with-java.time.temporal.TemporalField-long-) methods will throw an exception.

If the field is a [ChronoField](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/ChronoField.html) then the query is implemented here. The supported fields are:

* + - * 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
      * ERA

All other ChronoField instances will return false.

If the field is not a ChronoField, then the result of this method is obtained by invoking TemporalField.isSupportedBy(TemporalAccessor) passing this as the argument. Whether the field is supported is determined by the field.

**Specified by:**

[isSupported](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html#isSupported-java.time.temporal.TemporalField-) in interface [ChronoLocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html)

**Specified by:**

[isSupported](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalAccessor.html#isSupported-java.time.temporal.TemporalField-) in interface [TemporalAccessor](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalAccessor.html)

**Parameters:**

field - the field to check, null returns false

**Returns:**

true if the field is supported on this date, false if not

#### isSupported

public boolean isSupported([TemporalUnit](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalUnit.html) unit)

Checks if the specified unit is supported.

This checks if the specified unit can be added to, or subtracted from, this date. If false, then calling the [plus(long, TemporalUnit)](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#plus-long-java.time.temporal.TemporalUnit-) and [minus](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#minus-long-java.time.temporal.TemporalUnit-) methods will throw an exception.

If the unit is a [ChronoUnit](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/ChronoUnit.html) then the query is implemented here. The supported units are:

* + - * DAYS
      * WEEKS
      * MONTHS
      * YEARS
      * DECADES
      * CENTURIES
      * MILLENNIA
      * ERAS

All other ChronoUnit instances will return false.

If the unit is not a ChronoUnit, then the result of this method is obtained by invoking TemporalUnit.isSupportedBy(Temporal) passing this as the argument. Whether the unit is supported is determined by the unit.

**Specified by:**

[isSupported](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html#isSupported-java.time.temporal.TemporalUnit-) in interface [ChronoLocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html)

**Specified by:**

[isSupported](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/Temporal.html#isSupported-java.time.temporal.TemporalUnit-) in interface [Temporal](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/Temporal.html)

**Parameters:**

unit - the unit to check, null returns false

**Returns:**

true if the unit can be added/subtracted, false if not

#### range

public [ValueRange](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/ValueRange.html) range([TemporalField](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalField.html) field)

Gets the range of valid values for the specified field.

The range object expresses the minimum and maximum valid values for a field. This date is used to enhance the accuracy of the returned range. If it is not possible to return the range, because the field is not supported or for some other reason, an exception is thrown.

If the field is a [ChronoField](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/ChronoField.html) then the query is implemented here. The [supported fields](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#isSupported-java.time.temporal.TemporalField-) will return appropriate range instances. All other ChronoField instances will throw an UnsupportedTemporalTypeException.

If the field is not a ChronoField, then the result of this method is obtained by invoking TemporalField.rangeRefinedBy(TemporalAccessor) passing this as the argument. Whether the range can be obtained is determined by the field.

**Specified by:**

[range](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalAccessor.html#range-java.time.temporal.TemporalField-) in interface [TemporalAccessor](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalAccessor.html)

**Parameters:**

field - the field to query the range for, not null

**Returns:**

the range of valid values for the field, not null

**Throws:**

[DateTimeException](https://docs.oracle.com/javase/8/docs/api/java/time/DateTimeException.html) - if the range for the field cannot be obtained

[UnsupportedTemporalTypeException](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/UnsupportedTemporalTypeException.html) - if the field is not supported

#### get

public int get([TemporalField](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalField.html) field)

Gets the value of the specified field from this date as an int.

This queries this date for the value of the specified field. The returned value will always be within the valid range of values for the field. If it is not possible to return the value, because the field is not supported or for some other reason, an exception is thrown.

If the field is a [ChronoField](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/ChronoField.html) then the query is implemented here. The [supported fields](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#isSupported-java.time.temporal.TemporalField-) will return valid values based on this date, except EPOCH\_DAY and PROLEPTIC\_MONTH which are too large to fit in an int and throw a DateTimeException. All other ChronoField instances will throw an UnsupportedTemporalTypeException.

If the field is not a ChronoField, then the result of this method is obtained by invoking TemporalField.getFrom(TemporalAccessor) passing this as the argument. Whether the value can be obtained, and what the value represents, is determined by the field.

**Specified by:**

[get](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalAccessor.html#get-java.time.temporal.TemporalField-) in interface [TemporalAccessor](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalAccessor.html)

**Parameters:**

field - the field to get, not null

**Returns:**

the value for the field

**Throws:**

[DateTimeException](https://docs.oracle.com/javase/8/docs/api/java/time/DateTimeException.html) - if a value for the field cannot be obtained or the value is outside the range of valid values for the field

[UnsupportedTemporalTypeException](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/UnsupportedTemporalTypeException.html) - if the field is not supported or the range of values exceeds an int

[ArithmeticException](https://docs.oracle.com/javase/8/docs/api/java/lang/ArithmeticException.html) - if numeric overflow occurs

#### getLong

public long getLong([TemporalField](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalField.html) field)

Gets the value of the specified field from this date as a long.

This queries this date for the value of the specified field. If it is not possible to return the value, because the field is not supported or for some other reason, an exception is thrown.

If the field is a [ChronoField](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/ChronoField.html) then the query is implemented here. The [supported fields](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#isSupported-java.time.temporal.TemporalField-) will return valid values based on this date. All other ChronoField instances will throw an UnsupportedTemporalTypeException.

If the field is not a ChronoField, then the result of this method is obtained by invoking TemporalField.getFrom(TemporalAccessor) passing this as the argument. Whether the value can be obtained, and what the value represents, is determined by the field.

**Specified by:**

[getLong](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalAccessor.html#getLong-java.time.temporal.TemporalField-) in interface [TemporalAccessor](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalAccessor.html)

**Parameters:**

field - the field to get, not null

**Returns:**

the value for the field

**Throws:**

[DateTimeException](https://docs.oracle.com/javase/8/docs/api/java/time/DateTimeException.html) - if a value for the field cannot be obtained

[UnsupportedTemporalTypeException](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/UnsupportedTemporalTypeException.html) - if the field is not supported

[ArithmeticException](https://docs.oracle.com/javase/8/docs/api/java/lang/ArithmeticException.html) - if numeric overflow occurs

#### getChronology

public [IsoChronology](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/IsoChronology.html) getChronology()

Gets the chronology of this date, which is the ISO calendar system.

The Chronology represents the calendar system in use. The ISO-8601 calendar system is the modern civil calendar system used today in most of the world. It is equivalent to the proleptic Gregorian calendar system, in which today's rules for leap years are applied for all time.

**Specified by:**

[getChronology](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html#getChronology--) in interface [ChronoLocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html)

**Returns:**

the ISO chronology, not null

#### getEra

public [Era](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/Era.html) getEra()

Gets the era applicable at this date.

The official ISO-8601 standard does not define eras, however IsoChronology does. It defines two eras, 'CE' from year one onwards and 'BCE' from year zero backwards. Since dates before the Julian-Gregorian cutover are not in line with history, the cutover between 'BCE' and 'CE' is also not aligned with the commonly used eras, often referred to using 'BC' and 'AD'.

Users of this class should typically ignore this method as it exists primarily to fulfill the [ChronoLocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html) contract where it is necessary to support the Japanese calendar system.

The returned era will be a singleton capable of being compared with the constants in [IsoChronology](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/IsoChronology.html) using the == operator.

**Specified by:**

[getEra](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html#getEra--) in interface [ChronoLocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html)

**Returns:**

the IsoChronology era constant applicable at this date, not null

#### getYear

public int getYear()

Gets the year field.

This method returns the primitive int value for the year.

The year returned by this method is proleptic as per get(YEAR). To obtain the year-of-era, use get(YEAR\_OF\_ERA).

**Returns:**

the year, from MIN\_YEAR to MAX\_YEAR

#### getMonthValue

public int getMonthValue()

Gets the month-of-year field from 1 to 12.

This method returns the month as an int from 1 to 12. Application code is frequently clearer if the enum [Month](https://docs.oracle.com/javase/8/docs/api/java/time/Month.html) is used by calling [getMonth()](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#getMonth--).

**Returns:**

the month-of-year, from 1 to 12

**See Also:**

[getMonth()](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#getMonth--)

#### getMonth

public [Month](https://docs.oracle.com/javase/8/docs/api/java/time/Month.html) getMonth()

Gets the month-of-year field using the Month enum.

This method returns the enum [Month](https://docs.oracle.com/javase/8/docs/api/java/time/Month.html) for the month. This avoids confusion as to what int values mean. If you need access to the primitive int value then the enum provides the [int value](https://docs.oracle.com/javase/8/docs/api/java/time/Month.html#getValue--).

**Returns:**

the month-of-year, not null

**See Also:**

[getMonthValue()](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#getMonthValue--)

#### getDayOfMonth

public int getDayOfMonth()

Gets the day-of-month field.

This method returns the primitive int value for the day-of-month.

**Returns:**

the day-of-month, from 1 to 31

#### getDayOfYear

public int getDayOfYear()

Gets the day-of-year field.

This method returns the primitive int value for the day-of-year.

**Returns:**

the day-of-year, from 1 to 365, or 366 in a leap year

#### getDayOfWeek

public [DayOfWeek](https://docs.oracle.com/javase/8/docs/api/java/time/DayOfWeek.html) getDayOfWeek()

Gets the day-of-week field, which is an enum DayOfWeek.

This method returns the enum [DayOfWeek](https://docs.oracle.com/javase/8/docs/api/java/time/DayOfWeek.html) for the day-of-week. This avoids confusion as to what int values mean. If you need access to the primitive int value then the enum provides the [int value](https://docs.oracle.com/javase/8/docs/api/java/time/DayOfWeek.html#getValue--).

Additional information can be obtained from the DayOfWeek. This includes textual names of the values.

**Returns:**

the day-of-week, not null

#### isLeapYear

public boolean isLeapYear()

Checks if the year is a leap year, according to the ISO proleptic calendar system rules.

This method applies the current rules for leap years across the whole time-line. In general, a year is a leap year if it is divisible by four without remainder. However, years divisible by 100, are not leap years, with the exception of years divisible by 400 which are.

For example, 1904 is a leap year it is divisible by 4. 1900 was not a leap year as it is divisible by 100, however 2000 was a leap year as it is divisible by 400.

The calculation is proleptic - applying the same rules into the far future and far past. This is historically inaccurate, but is correct for the ISO-8601 standard.

**Specified by:**

[isLeapYear](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html#isLeapYear--) in interface [ChronoLocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html)

**Returns:**

true if the year is leap, false otherwise

#### lengthOfMonth

public int lengthOfMonth()

Returns the length of the month represented by this date.

This returns the length of the month in days. For example, a date in January would return 31.

**Specified by:**

[lengthOfMonth](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html#lengthOfMonth--) in interface [ChronoLocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html)

**Returns:**

the length of the month in days

#### lengthOfYear

public int lengthOfYear()

Returns the length of the year represented by this date.

This returns the length of the year in days, either 365 or 366.

**Specified by:**

[lengthOfYear](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html#lengthOfYear--) in interface [ChronoLocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html)

**Returns:**

366 if the year is leap, 365 otherwise

#### with

public [LocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) with([TemporalAdjuster](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalAdjuster.html) adjuster)

Returns an adjusted copy of this date.

This returns a LocalDate, based on this one, with the date adjusted. The adjustment takes place using the specified adjuster strategy object. Read the documentation of the adjuster to understand what adjustment will be made.

A simple adjuster might simply set the one of the fields, such as the year field. A more complex adjuster might set the date to the last day of the month.

A selection of common adjustments is provided in [TemporalAdjusters](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalAdjusters.html). These include finding the "last day of the month" and "next Wednesday". Key date-time classes also implement the TemporalAdjuster interface, such as [Month](https://docs.oracle.com/javase/8/docs/api/java/time/Month.html) and [MonthDay](https://docs.oracle.com/javase/8/docs/api/java/time/MonthDay.html). The adjuster is responsible for handling special cases, such as the varying lengths of month and leap years.

For example this code returns a date on the last day of July:

import static java.time.Month.\*;

import static java.time.temporal.TemporalAdjusters.\*;

result = localDate.with(JULY).with(lastDayOfMonth());

The result of this method is obtained by invoking the [TemporalAdjuster.adjustInto(Temporal)](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalAdjuster.html#adjustInto-java.time.temporal.Temporal-) method on the specified adjuster passing this as the argument.

This instance is immutable and unaffected by this method call.

**Specified by:**

[with](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html#with-java.time.temporal.TemporalAdjuster-) in interface [ChronoLocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html)

**Specified by:**

[with](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/Temporal.html#with-java.time.temporal.TemporalAdjuster-) in interface [Temporal](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/Temporal.html)

**Parameters:**

adjuster - the adjuster to use, not null

**Returns:**

a LocalDate based on this with the adjustment made, not null

**Throws:**

[DateTimeException](https://docs.oracle.com/javase/8/docs/api/java/time/DateTimeException.html) - if the adjustment cannot be made

[ArithmeticException](https://docs.oracle.com/javase/8/docs/api/java/lang/ArithmeticException.html) - if numeric overflow occurs

#### with

* + - public [LocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) with([TemporalField](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalField.html) field,

long newValue)

Returns a copy of this date with the specified field set to a new value.

This returns a LocalDate, based on this one, with the value for the specified field changed. This can be used to change any supported field, such as the year, month or day-of-month. If it is not possible to set the value, because the field is not supported or for some other reason, an exception is thrown.

In some cases, changing the specified field can cause the resulting date to become invalid, such as changing the month from 31st January to February would make the day-of-month invalid. In cases like this, the field is responsible for resolving the date. Typically it will choose the previous valid date, which would be the last valid day of February in this example.

If the field is a [ChronoField](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/ChronoField.html) then the adjustment is implemented here. The supported fields behave as follows:

* + - * DAY\_OF\_WEEK - Returns a LocalDate with the specified day-of-week. The date is adjusted up to 6 days forward or backward within the boundary of a Monday to Sunday week.
      * ALIGNED\_DAY\_OF\_WEEK\_IN\_MONTH - Returns a LocalDate with the specified aligned-day-of-week. The date is adjusted to the specified month-based aligned-day-of-week. Aligned weeks are counted such that the first week of a given month starts on the first day of that month. This may cause the date to be moved up to 6 days into the following month.
      * ALIGNED\_DAY\_OF\_WEEK\_IN\_YEAR - Returns a LocalDate with the specified aligned-day-of-week. The date is adjusted to the specified year-based aligned-day-of-week. Aligned weeks are counted such that the first week of a given year starts on the first day of that year. This may cause the date to be moved up to 6 days into the following year.
      * DAY\_OF\_MONTH - Returns a LocalDate with the specified day-of-month. The month and year will be unchanged. If the day-of-month is invalid for the year and month, then a DateTimeException is thrown.
      * DAY\_OF\_YEAR - Returns a LocalDate with the specified day-of-year. The year will be unchanged. If the day-of-year is invalid for the year, then a DateTimeException is thrown.
      * EPOCH\_DAY - Returns a LocalDate with the specified epoch-day. This completely replaces the date and is equivalent to [ofEpochDay(long)](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#ofEpochDay-long-).
      * ALIGNED\_WEEK\_OF\_MONTH - Returns a LocalDate with the specified aligned-week-of-month. Aligned weeks are counted such that the first week of a given month starts on the first day of that month. This adjustment moves the date in whole week chunks to match the specified week. The result will have the same day-of-week as this date. This may cause the date to be moved into the following month.
      * ALIGNED\_WEEK\_OF\_YEAR - Returns a LocalDate with the specified aligned-week-of-year. Aligned weeks are counted such that the first week of a given year starts on the first day of that year. This adjustment moves the date in whole week chunks to match the specified week. The result will have the same day-of-week as this date. This may cause the date to be moved into the following year.
      * MONTH\_OF\_YEAR - Returns a LocalDate with the specified month-of-year. The year will be unchanged. The day-of-month will also be unchanged, unless it would be invalid for the new month and year. In that case, the day-of-month is adjusted to the maximum valid value for the new month and year.
      * PROLEPTIC\_MONTH - Returns a LocalDate with the specified proleptic-month. The day-of-month will be unchanged, unless it would be invalid for the new month and year. In that case, the day-of-month is adjusted to the maximum valid value for the new month and year.
      * YEAR\_OF\_ERA - Returns a LocalDate with the specified year-of-era. The era and month will be unchanged. The day-of-month will also be unchanged, unless it would be invalid for the new month and year. In that case, the day-of-month is adjusted to the maximum valid value for the new month and year.
      * YEAR - Returns a LocalDate with the specified year. The month will be unchanged. The day-of-month will also be unchanged, unless it would be invalid for the new month and year. In that case, the day-of-month is adjusted to the maximum valid value for the new month and year.
      * ERA - Returns a LocalDate with the specified era. The year-of-era and month will be unchanged. The day-of-month will also be unchanged, unless it would be invalid for the new month and year. In that case, the day-of-month is adjusted to the maximum valid value for the new month and year.

In all cases, if the new value is outside the valid range of values for the field then a DateTimeException will be thrown.

All other ChronoField instances will throw an UnsupportedTemporalTypeException.

If the field is not a ChronoField, then the result of this method is obtained by invoking TemporalField.adjustInto(Temporal, long) passing this as the argument. In this case, the field determines whether and how to adjust the instant.

This instance is immutable and unaffected by this method call.

**Specified by:**

[with](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html#with-java.time.temporal.TemporalField-long-) in interface [ChronoLocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html)

**Specified by:**

[with](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/Temporal.html#with-java.time.temporal.TemporalField-long-) in interface [Temporal](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/Temporal.html)

**Parameters:**

field - the field to set in the result, not null

newValue - the new value of the field in the result

**Returns:**

a LocalDate based on this with the specified field set, not null

**Throws:**

[DateTimeException](https://docs.oracle.com/javase/8/docs/api/java/time/DateTimeException.html) - if the field cannot be set

[UnsupportedTemporalTypeException](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/UnsupportedTemporalTypeException.html) - if the field is not supported

[ArithmeticException](https://docs.oracle.com/javase/8/docs/api/java/lang/ArithmeticException.html) - if numeric overflow occurs

#### withYear

public [LocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) withYear(int year)

Returns a copy of this LocalDate with the year altered.

If the day-of-month is invalid for the year, it will be changed to the last valid day of the month.

This instance is immutable and unaffected by this method call.

**Parameters:**

year - the year to set in the result, from MIN\_YEAR to MAX\_YEAR

**Returns:**

a LocalDate based on this date with the requested year, not null

**Throws:**

[DateTimeException](https://docs.oracle.com/javase/8/docs/api/java/time/DateTimeException.html) - if the year value is invalid

#### withMonth

public [LocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) withMonth(int month)

Returns a copy of this LocalDate with the month-of-year altered.

If the day-of-month is invalid for the year, it will be changed to the last valid day of the month.

This instance is immutable and unaffected by this method call.

**Parameters:**

month - the month-of-year to set in the result, from 1 (January) to 12 (December)

**Returns:**

a LocalDate based on this date with the requested month, not null

**Throws:**

[DateTimeException](https://docs.oracle.com/javase/8/docs/api/java/time/DateTimeException.html) - if the month-of-year value is invalid

#### withDayOfMonth

public [LocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) withDayOfMonth(int dayOfMonth)

Returns a copy of this LocalDate with the day-of-month altered.

If the resulting date is invalid, an exception is thrown.

This instance is immutable and unaffected by this method call.

**Parameters:**

dayOfMonth - the day-of-month to set in the result, from 1 to 28-31

**Returns:**

a LocalDate based on this date with the requested day, not null

**Throws:**

[DateTimeException](https://docs.oracle.com/javase/8/docs/api/java/time/DateTimeException.html) - if the day-of-month value is invalid, or if the day-of-month is invalid for the month-year

#### withDayOfYear

public [LocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) withDayOfYear(int dayOfYear)

Returns a copy of this LocalDate with the day-of-year altered.

If the resulting date is invalid, an exception is thrown.

This instance is immutable and unaffected by this method call.

**Parameters:**

dayOfYear - the day-of-year to set in the result, from 1 to 365-366

**Returns:**

a LocalDate based on this date with the requested day, not null

**Throws:**

[DateTimeException](https://docs.oracle.com/javase/8/docs/api/java/time/DateTimeException.html) - if the day-of-year value is invalid, or if the day-of-year is invalid for the year

#### plus

public [LocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) plus([TemporalAmount](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalAmount.html) amountToAdd)

Returns a copy of this date with the specified amount added.

This returns a LocalDate, based on this one, with the specified amount added. The amount is typically [Period](https://docs.oracle.com/javase/8/docs/api/java/time/Period.html) but may be any other type implementing the [TemporalAmount](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalAmount.html) interface.

The calculation is delegated to the amount object by calling [TemporalAmount.addTo(Temporal)](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalAmount.html#addTo-java.time.temporal.Temporal-). The amount implementation is free to implement the addition in any way it wishes, however it typically calls back to [plus(long, TemporalUnit)](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#plus-long-java.time.temporal.TemporalUnit-). Consult the documentation of the amount implementation to determine if it can be successfully added.

This instance is immutable and unaffected by this method call.

**Specified by:**

[plus](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html#plus-java.time.temporal.TemporalAmount-) in interface [ChronoLocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html)

**Specified by:**

[plus](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/Temporal.html#plus-java.time.temporal.TemporalAmount-) in interface [Temporal](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/Temporal.html)

**Parameters:**

amountToAdd - the amount to add, not null

**Returns:**

a LocalDate based on this date with the addition made, not null

**Throws:**

[DateTimeException](https://docs.oracle.com/javase/8/docs/api/java/time/DateTimeException.html) - if the addition cannot be made

[ArithmeticException](https://docs.oracle.com/javase/8/docs/api/java/lang/ArithmeticException.html) - if numeric overflow occurs

#### plus

* + - public [LocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) plus(long amountToAdd,

[TemporalUnit](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalUnit.html) unit)

Returns a copy of this date with the specified amount added.

This returns a LocalDate, based on this one, with the amount in terms of the unit added. If it is not possible to add the amount, because the unit is not supported or for some other reason, an exception is thrown.

In some cases, adding the amount can cause the resulting date to become invalid. For example, adding one month to 31st January would result in 31st February. In cases like this, the unit is responsible for resolving the date. Typically it will choose the previous valid date, which would be the last valid day of February in this example.

If the field is a [ChronoUnit](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/ChronoUnit.html) then the addition is implemented here. The supported fields behave as follows:

* + - * DAYS - Returns a LocalDate with the specified number of days added. This is equivalent to [plusDays(long)](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#plusDays-long-).
      * WEEKS - Returns a LocalDate with the specified number of weeks added. This is equivalent to [plusWeeks(long)](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#plusWeeks-long-) and uses a 7 day week.
      * MONTHS - Returns a LocalDate with the specified number of months added. This is equivalent to [plusMonths(long)](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#plusMonths-long-). The day-of-month will be unchanged unless it would be invalid for the new month and year. In that case, the day-of-month is adjusted to the maximum valid value for the new month and year.
      * YEARS - Returns a LocalDate with the specified number of years added. This is equivalent to [plusYears(long)](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#plusYears-long-). The day-of-month will be unchanged unless it would be invalid for the new month and year. In that case, the day-of-month is adjusted to the maximum valid value for the new month and year.
      * DECADES - Returns a LocalDate with the specified number of decades added. This is equivalent to calling [plusYears(long)](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#plusYears-long-) with the amount multiplied by 10. The day-of-month will be unchanged unless it would be invalid for the new month and year. In that case, the day-of-month is adjusted to the maximum valid value for the new month and year.
      * CENTURIES - Returns a LocalDate with the specified number of centuries added. This is equivalent to calling [plusYears(long)](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#plusYears-long-) with the amount multiplied by 100. The day-of-month will be unchanged unless it would be invalid for the new month and year. In that case, the day-of-month is adjusted to the maximum valid value for the new month and year.
      * MILLENNIA - Returns a LocalDate with the specified number of millennia added. This is equivalent to calling [plusYears(long)](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#plusYears-long-) with the amount multiplied by 1,000. The day-of-month will be unchanged unless it would be invalid for the new month and year. In that case, the day-of-month is adjusted to the maximum valid value for the new month and year.
      * ERAS - Returns a LocalDate with the specified number of eras added. Only two eras are supported so the amount must be one, zero or minus one. If the amount is non-zero then the year is changed such that the year-of-era is unchanged. The day-of-month will be unchanged unless it would be invalid for the new month and year. In that case, the day-of-month is adjusted to the maximum valid value for the new month and year.

All other ChronoUnit instances will throw an UnsupportedTemporalTypeException.

If the field is not a ChronoUnit, then the result of this method is obtained by invoking TemporalUnit.addTo(Temporal, long) passing this as the argument. In this case, the unit determines whether and how to perform the addition.

This instance is immutable and unaffected by this method call.

**Specified by:**

[plus](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html#plus-long-java.time.temporal.TemporalUnit-) in interface [ChronoLocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html)

**Specified by:**

[plus](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/Temporal.html#plus-long-java.time.temporal.TemporalUnit-) in interface [Temporal](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/Temporal.html)

**Parameters:**

amountToAdd - the amount of the unit to add to the result, may be negative

unit - the unit of the amount to add, not null

**Returns:**

a LocalDate based on this date with the specified amount added, not null

**Throws:**

[DateTimeException](https://docs.oracle.com/javase/8/docs/api/java/time/DateTimeException.html) - if the addition cannot be made

[UnsupportedTemporalTypeException](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/UnsupportedTemporalTypeException.html) - if the unit is not supported

[ArithmeticException](https://docs.oracle.com/javase/8/docs/api/java/lang/ArithmeticException.html) - if numeric overflow occurs

#### plusYears

public [LocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) plusYears(long yearsToAdd)

Returns a copy of this LocalDate with the specified number of years added.

This method adds the specified amount to the years field in three steps:

* + - * Add the input years to the year field
      * Check if the resulting date would be invalid
      * Adjust the day-of-month to the last valid day if necessary

For example, 2008-02-29 (leap year) plus one year would result in the invalid date 2009-02-29 (standard year). Instead of returning an invalid result, the last valid day of the month, 2009-02-28, is selected instead.

This instance is immutable and unaffected by this method call.

**Parameters:**

yearsToAdd - the years to add, may be negative

**Returns:**

a LocalDate based on this date with the years added, not null

**Throws:**

[DateTimeException](https://docs.oracle.com/javase/8/docs/api/java/time/DateTimeException.html) - if the result exceeds the supported date range

#### plusMonths

public [LocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) plusMonths(long monthsToAdd)

Returns a copy of this LocalDate with the specified number of months added.

This method adds the specified amount to the months field in three steps:

* + - * Add the input months to the month-of-year field
      * Check if the resulting date would be invalid
      * Adjust the day-of-month to the last valid day if necessary

For example, 2007-03-31 plus one month would result in the invalid date 2007-04-31. Instead of returning an invalid result, the last valid day of the month, 2007-04-30, is selected instead.

This instance is immutable and unaffected by this method call.

**Parameters:**

monthsToAdd - the months to add, may be negative

**Returns:**

a LocalDate based on this date with the months added, not null

**Throws:**

[DateTimeException](https://docs.oracle.com/javase/8/docs/api/java/time/DateTimeException.html) - if the result exceeds the supported date range

#### plusWeeks

public [LocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) plusWeeks(long weeksToAdd)

Returns a copy of this LocalDate with the specified number of weeks added.

This method adds the specified amount in weeks to the days field incrementing the month and year fields as necessary to ensure the result remains valid. The result is only invalid if the maximum/minimum year is exceeded.

For example, 2008-12-31 plus one week would result in 2009-01-07.

This instance is immutable and unaffected by this method call.

**Parameters:**

weeksToAdd - the weeks to add, may be negative

**Returns:**

a LocalDate based on this date with the weeks added, not null

**Throws:**

[DateTimeException](https://docs.oracle.com/javase/8/docs/api/java/time/DateTimeException.html) - if the result exceeds the supported date range

#### plusDays

public [LocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) plusDays(long daysToAdd)

Returns a copy of this LocalDate with the specified number of days added.

This method adds the specified amount to the days field incrementing the month and year fields as necessary to ensure the result remains valid. The result is only invalid if the maximum/minimum year is exceeded.

For example, 2008-12-31 plus one day would result in 2009-01-01.

This instance is immutable and unaffected by this method call.

**Parameters:**

daysToAdd - the days to add, may be negative

**Returns:**

a LocalDate based on this date with the days added, not null

**Throws:**

[DateTimeException](https://docs.oracle.com/javase/8/docs/api/java/time/DateTimeException.html) - if the result exceeds the supported date range

#### minus

public [LocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) minus([TemporalAmount](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalAmount.html) amountToSubtract)

Returns a copy of this date with the specified amount subtracted.

This returns a LocalDate, based on this one, with the specified amount subtracted. The amount is typically [Period](https://docs.oracle.com/javase/8/docs/api/java/time/Period.html) but may be any other type implementing the [TemporalAmount](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalAmount.html) interface.

The calculation is delegated to the amount object by calling [TemporalAmount.subtractFrom(Temporal)](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalAmount.html#subtractFrom-java.time.temporal.Temporal-). The amount implementation is free to implement the subtraction in any way it wishes, however it typically calls back to [minus(long, TemporalUnit)](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#minus-long-java.time.temporal.TemporalUnit-). Consult the documentation of the amount implementation to determine if it can be successfully subtracted.

This instance is immutable and unaffected by this method call.

**Specified by:**

[minus](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html#minus-java.time.temporal.TemporalAmount-) in interface [ChronoLocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html)

**Specified by:**

[minus](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/Temporal.html#minus-java.time.temporal.TemporalAmount-) in interface [Temporal](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/Temporal.html)

**Parameters:**

amountToSubtract - the amount to subtract, not null

**Returns:**

a LocalDate based on this date with the subtraction made, not null

**Throws:**

[DateTimeException](https://docs.oracle.com/javase/8/docs/api/java/time/DateTimeException.html) - if the subtraction cannot be made

[ArithmeticException](https://docs.oracle.com/javase/8/docs/api/java/lang/ArithmeticException.html) - if numeric overflow occurs

#### minus

* + - public [LocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) minus(long amountToSubtract,

[TemporalUnit](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalUnit.html) unit)

Returns a copy of this date with the specified amount subtracted.

This returns a LocalDate, based on this one, with the amount in terms of the unit subtracted. If it is not possible to subtract the amount, because the unit is not supported or for some other reason, an exception is thrown.

This method is equivalent to [plus(long, TemporalUnit)](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#plus-long-java.time.temporal.TemporalUnit-) with the amount negated. See that method for a full description of how addition, and thus subtraction, works.

This instance is immutable and unaffected by this method call.

**Specified by:**

[minus](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html#minus-long-java.time.temporal.TemporalUnit-) in interface [ChronoLocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html)

**Specified by:**

[minus](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/Temporal.html#minus-long-java.time.temporal.TemporalUnit-) in interface [Temporal](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/Temporal.html)

**Parameters:**

amountToSubtract - the amount of the unit to subtract from the result, may be negative

unit - the unit of the amount to subtract, not null

**Returns:**

a LocalDate based on this date with the specified amount subtracted, not null

**Throws:**

[DateTimeException](https://docs.oracle.com/javase/8/docs/api/java/time/DateTimeException.html) - if the subtraction cannot be made

[UnsupportedTemporalTypeException](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/UnsupportedTemporalTypeException.html) - if the unit is not supported

[ArithmeticException](https://docs.oracle.com/javase/8/docs/api/java/lang/ArithmeticException.html) - if numeric overflow occurs

#### minusYears

public [LocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) minusYears(long yearsToSubtract)

Returns a copy of this LocalDate with the specified number of years subtracted.

This method subtracts the specified amount from the years field in three steps:

* + - * Subtract the input years from the year field
      * Check if the resulting date would be invalid
      * Adjust the day-of-month to the last valid day if necessary

For example, 2008-02-29 (leap year) minus one year would result in the invalid date 2007-02-29 (standard year). Instead of returning an invalid result, the last valid day of the month, 2007-02-28, is selected instead.

This instance is immutable and unaffected by this method call.

**Parameters:**

yearsToSubtract - the years to subtract, may be negative

**Returns:**

a LocalDate based on this date with the years subtracted, not null

**Throws:**

[DateTimeException](https://docs.oracle.com/javase/8/docs/api/java/time/DateTimeException.html) - if the result exceeds the supported date range

#### minusMonths

public [LocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) minusMonths(long monthsToSubtract)

Returns a copy of this LocalDate with the specified number of months subtracted.

This method subtracts the specified amount from the months field in three steps:

* + - * Subtract the input months from the month-of-year field
      * Check if the resulting date would be invalid
      * Adjust the day-of-month to the last valid day if necessary

For example, 2007-03-31 minus one month would result in the invalid date 2007-02-31. Instead of returning an invalid result, the last valid day of the month, 2007-02-28, is selected instead.

This instance is immutable and unaffected by this method call.

**Parameters:**

monthsToSubtract - the months to subtract, may be negative

**Returns:**

a LocalDate based on this date with the months subtracted, not null

**Throws:**

[DateTimeException](https://docs.oracle.com/javase/8/docs/api/java/time/DateTimeException.html) - if the result exceeds the supported date range

#### minusWeeks

public [LocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) minusWeeks(long weeksToSubtract)

Returns a copy of this LocalDate with the specified number of weeks subtracted.

This method subtracts the specified amount in weeks from the days field decrementing the month and year fields as necessary to ensure the result remains valid. The result is only invalid if the maximum/minimum year is exceeded.

For example, 2009-01-07 minus one week would result in 2008-12-31.

This instance is immutable and unaffected by this method call.

**Parameters:**

weeksToSubtract - the weeks to subtract, may be negative

**Returns:**

a LocalDate based on this date with the weeks subtracted, not null

**Throws:**

[DateTimeException](https://docs.oracle.com/javase/8/docs/api/java/time/DateTimeException.html) - if the result exceeds the supported date range

#### minusDays

public [LocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) minusDays(long daysToSubtract)

Returns a copy of this LocalDate with the specified number of days subtracted.

This method subtracts the specified amount from the days field decrementing the month and year fields as necessary to ensure the result remains valid. The result is only invalid if the maximum/minimum year is exceeded.

For example, 2009-01-01 minus one day would result in 2008-12-31.

This instance is immutable and unaffected by this method call.

**Parameters:**

daysToSubtract - the days to subtract, may be negative

**Returns:**

a LocalDate based on this date with the days subtracted, not null

**Throws:**

[DateTimeException](https://docs.oracle.com/javase/8/docs/api/java/time/DateTimeException.html) - if the result exceeds the supported date range

#### query

public <R> R query([TemporalQuery](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalQuery.html)<R> query)

Queries this date using the specified query.

This queries this date using the specified query strategy object. The TemporalQuery object defines the logic to be used to obtain the result. Read the documentation of the query to understand what the result of this method will be.

The result of this method is obtained by invoking the [TemporalQuery.queryFrom(TemporalAccessor)](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalQuery.html#queryFrom-java.time.temporal.TemporalAccessor-) method on the specified query passing this as the argument.

**Specified by:**

[query](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html#query-java.time.temporal.TemporalQuery-) in interface [ChronoLocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html)

**Specified by:**

[query](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalAccessor.html#query-java.time.temporal.TemporalQuery-) in interface [TemporalAccessor](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalAccessor.html)

**Type Parameters:**

R - the type of the result

**Parameters:**

query - the query to invoke, not null

**Returns:**

the query result, null may be returned (defined by the query)

**Throws:**

[DateTimeException](https://docs.oracle.com/javase/8/docs/api/java/time/DateTimeException.html) - if unable to query (defined by the query)

[ArithmeticException](https://docs.oracle.com/javase/8/docs/api/java/lang/ArithmeticException.html) - if numeric overflow occurs (defined by the query)

#### adjustInto

public [Temporal](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/Temporal.html) adjustInto([Temporal](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/Temporal.html) temporal)

Adjusts the specified temporal object to have the same date as this object.

This returns a temporal object of the same observable type as the input with the date changed to be the same as this.

The adjustment is equivalent to using [Temporal.with(TemporalField, long)](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/Temporal.html#with-java.time.temporal.TemporalField-long-) passing [ChronoField.EPOCH\_DAY](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/ChronoField.html#EPOCH_DAY) as the field.

In most cases, it is clearer to reverse the calling pattern by using [Temporal.with(TemporalAdjuster)](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/Temporal.html#with-java.time.temporal.TemporalAdjuster-):

// these two lines are equivalent, but the second approach is recommended

temporal = thisLocalDate.adjustInto(temporal);

temporal = temporal.with(thisLocalDate);

This instance is immutable and unaffected by this method call.

**Specified by:**

[adjustInto](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html#adjustInto-java.time.temporal.Temporal-) in interface [ChronoLocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html)

**Specified by:**

[adjustInto](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalAdjuster.html#adjustInto-java.time.temporal.Temporal-) in interface [TemporalAdjuster](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalAdjuster.html)

**Parameters:**

temporal - the target object to be adjusted, not null

**Returns:**

the adjusted object, not null

**Throws:**

[DateTimeException](https://docs.oracle.com/javase/8/docs/api/java/time/DateTimeException.html) - if unable to make the adjustment

[ArithmeticException](https://docs.oracle.com/javase/8/docs/api/java/lang/ArithmeticException.html) - if numeric overflow occurs

#### until

* + - public long until([Temporal](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/Temporal.html) endExclusive,

[TemporalUnit](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalUnit.html) unit)

Calculates the amount of time until another date in terms of the specified unit.

This calculates the amount of time between two LocalDate objects in terms of a single TemporalUnit. The start and end points are this and the specified date. The result will be negative if the end is before the start. The Temporal passed to this method is converted to a LocalDate using [from(TemporalAccessor)](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#from-java.time.temporal.TemporalAccessor-). For example, the amount in days between two dates can be calculated using startDate.until(endDate, DAYS).

The calculation returns a whole number, representing the number of complete units between the two dates. For example, the amount in months between 2012-06-15 and 2012-08-14 will only be one month as it is one day short of two months.

There are two equivalent ways of using this method. The first is to invoke this method. The second is to use [TemporalUnit.between(Temporal, Temporal)](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalUnit.html#between-java.time.temporal.Temporal-java.time.temporal.Temporal-):

// these two lines are equivalent

amount = start.until(end, MONTHS);

amount = MONTHS.between(start, end);

The choice should be made based on which makes the code more readable.

The calculation is implemented in this method for [ChronoUnit](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/ChronoUnit.html). The units DAYS, WEEKS, MONTHS, YEARS, DECADES, CENTURIES, MILLENNIA and ERAS are supported. Other ChronoUnit values will throw an exception.

If the unit is not a ChronoUnit, then the result of this method is obtained by invoking TemporalUnit.between(Temporal, Temporal) passing this as the first argument and the converted input temporal as the second argument.

This instance is immutable and unaffected by this method call.

**Specified by:**

[until](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html#until-java.time.temporal.Temporal-java.time.temporal.TemporalUnit-) in interface [ChronoLocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html)

**Specified by:**

[until](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/Temporal.html#until-java.time.temporal.Temporal-java.time.temporal.TemporalUnit-) in interface [Temporal](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/Temporal.html)

**Parameters:**

endExclusive - the end date, exclusive, which is converted to a LocalDate, not null

unit - the unit to measure the amount in, not null

**Returns:**

the amount of time between this date and the end date

**Throws:**

[DateTimeException](https://docs.oracle.com/javase/8/docs/api/java/time/DateTimeException.html) - if the amount cannot be calculated, or the end temporal cannot be converted to a LocalDate

[UnsupportedTemporalTypeException](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/UnsupportedTemporalTypeException.html) - if the unit is not supported

[ArithmeticException](https://docs.oracle.com/javase/8/docs/api/java/lang/ArithmeticException.html) - if numeric overflow occurs

#### until

public [Period](https://docs.oracle.com/javase/8/docs/api/java/time/Period.html) until([ChronoLocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html) endDateExclusive)

Calculates the period between this date and another date as a Period.

This calculates the period between two dates in terms of years, months and days. The start and end points are this and the specified date. The result will be negative if the end is before the start. The negative sign will be the same in each of year, month and day.

The calculation is performed using the ISO calendar system. If necessary, the input date will be converted to ISO.

The start date is included, but the end date is not. The period is calculated by removing complete months, then calculating the remaining number of days, adjusting to ensure that both have the same sign. The number of months is then normalized into years and months based on a 12 month year. A month is considered to be complete if the end day-of-month is greater than or equal to the start day-of-month. For example, from 2010-01-15 to 2011-03-18 is "1 year, 2 months and 3 days".

There are two equivalent ways of using this method. The first is to invoke this method. The second is to use [Period.between(LocalDate, LocalDate)](https://docs.oracle.com/javase/8/docs/api/java/time/Period.html#between-java.time.LocalDate-java.time.LocalDate-):

// these two lines are equivalent

period = start.until(end);

period = Period.between(start, end);

The choice should be made based on which makes the code more readable.

**Specified by:**

[until](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html#until-java.time.chrono.ChronoLocalDate-) in interface [ChronoLocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html)

**Parameters:**

endDateExclusive - the end date, exclusive, which may be in any chronology, not null

**Returns:**

the period between this date and the end date, not null

#### format

public [String](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html) format([DateTimeFormatter](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html) formatter)

Formats this date using the specified formatter.

This date will be passed to the formatter to produce a string.

**Specified by:**

[format](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html#format-java.time.format.DateTimeFormatter-) in interface [ChronoLocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html)

**Parameters:**

formatter - the formatter to use, not null

**Returns:**

the formatted date string, not null

**Throws:**

[DateTimeException](https://docs.oracle.com/javase/8/docs/api/java/time/DateTimeException.html) - if an error occurs during printing

#### atTime

public [LocalDateTime](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDateTime.html) atTime([LocalTime](https://docs.oracle.com/javase/8/docs/api/java/time/LocalTime.html) time)

Combines this date with a time to create a LocalDateTime.

This returns a LocalDateTime formed from this date at the specified time. All possible combinations of date and time are valid.

**Specified by:**

[atTime](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html#atTime-java.time.LocalTime-) in interface [ChronoLocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html)

**Parameters:**

time - the time to combine with, not null

**Returns:**

the local date-time formed from this date and the specified time, not null

#### atTime

* + - public [LocalDateTime](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDateTime.html) atTime(int hour,

int minute)

Combines this date with a time to create a LocalDateTime.

This returns a LocalDateTime formed from this date at the specified hour and minute. The seconds and nanosecond fields will be set to zero. The individual time fields must be within their valid range. All possible combinations of date and time are valid.

**Parameters:**

hour - the hour-of-day to use, from 0 to 23

minute - the minute-of-hour to use, from 0 to 59

**Returns:**

the local date-time formed from this date and the specified time, not null

**Throws:**

[DateTimeException](https://docs.oracle.com/javase/8/docs/api/java/time/DateTimeException.html) - if the value of any field is out of range

#### atTime

* + - public [LocalDateTime](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDateTime.html) atTime(int hour,
    - int minute,

int second)

Combines this date with a time to create a LocalDateTime.

This returns a LocalDateTime formed from this date at the specified hour, minute and second. The nanosecond field will be set to zero. The individual time fields must be within their valid range. All possible combinations of date and time are valid.

**Parameters:**

hour - the hour-of-day to use, from 0 to 23

minute - the minute-of-hour to use, from 0 to 59

second - the second-of-minute to represent, from 0 to 59

**Returns:**

the local date-time formed from this date and the specified time, not null

**Throws:**

[DateTimeException](https://docs.oracle.com/javase/8/docs/api/java/time/DateTimeException.html) - if the value of any field is out of range

#### atTime

* + - public [LocalDateTime](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDateTime.html) atTime(int hour,
    - int minute,
    - int second,

int nanoOfSecond)

Combines this date with a time to create a LocalDateTime.

This returns a LocalDateTime formed from this date at the specified hour, minute, second and nanosecond. The individual time fields must be within their valid range. All possible combinations of date and time are valid.

**Parameters:**

hour - the hour-of-day to use, from 0 to 23

minute - the minute-of-hour to use, from 0 to 59

second - the second-of-minute to represent, from 0 to 59

nanoOfSecond - the nano-of-second to represent, from 0 to 999,999,999

**Returns:**

the local date-time formed from this date and the specified time, not null

**Throws:**

[DateTimeException](https://docs.oracle.com/javase/8/docs/api/java/time/DateTimeException.html) - if the value of any field is out of range

#### atTime

public [OffsetDateTime](https://docs.oracle.com/javase/8/docs/api/java/time/OffsetDateTime.html) atTime([OffsetTime](https://docs.oracle.com/javase/8/docs/api/java/time/OffsetTime.html) time)

Combines this date with an offset time to create an OffsetDateTime.

This returns an OffsetDateTime formed from this date at the specified time. All possible combinations of date and time are valid.

**Parameters:**

time - the time to combine with, not null

**Returns:**

the offset date-time formed from this date and the specified time, not null

#### atStartOfDay

public [LocalDateTime](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDateTime.html) atStartOfDay()

Combines this date with the time of midnight to create a LocalDateTime at the start of this date.

This returns a LocalDateTime formed from this date at the time of midnight, 00:00, at the start of this date.

**Returns:**

the local date-time of midnight at the start of this date, not null

#### atStartOfDay

public [ZonedDateTime](https://docs.oracle.com/javase/8/docs/api/java/time/ZonedDateTime.html) atStartOfDay([ZoneId](https://docs.oracle.com/javase/8/docs/api/java/time/ZoneId.html) zone)

Returns a zoned date-time from this date at the earliest valid time according to the rules in the time-zone.

Time-zone rules, such as daylight savings, mean that not every local date-time is valid for the specified zone, thus the local date-time may not be midnight.

In most cases, there is only one valid offset for a local date-time. In the case of an overlap, there are two valid offsets, and the earlier one is used, corresponding to the first occurrence of midnight on the date. In the case of a gap, the zoned date-time will represent the instant just after the gap.

If the zone ID is a [ZoneOffset](https://docs.oracle.com/javase/8/docs/api/java/time/ZoneOffset.html), then the result always has a time of midnight.

To convert to a specific time in a given time-zone call [atTime(LocalTime)](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#atTime-java.time.LocalTime-) followed by [LocalDateTime.atZone(ZoneId)](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDateTime.html#atZone-java.time.ZoneId-).

**Parameters:**

zone - the zone ID to use, not null

**Returns:**

the zoned date-time formed from this date and the earliest valid time for the zone, not null

#### toEpochDay

public long toEpochDay()

**Description copied from interface:**[**ChronoLocalDate**](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html#toEpochDay--)

Converts this date to the Epoch Day.

The [Epoch Day count](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/ChronoField.html#EPOCH_DAY) is a simple incrementing count of days where day 0 is 1970-01-01 (ISO). This definition is the same for all chronologies, enabling conversion.

This default implementation queries the EPOCH\_DAY field.

**Specified by:**

[toEpochDay](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html#toEpochDay--) in interface [ChronoLocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html)

**Returns:**

the Epoch Day equivalent to this date

#### compareTo

public int compareTo([ChronoLocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html) other)

Compares this date to another date.

The comparison is primarily based on the date, from earliest to latest. It is "consistent with equals", as defined by [Comparable](https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html).

If all the dates being compared are instances of LocalDate, then the comparison will be entirely based on the date. If some dates being compared are in different chronologies, then the chronology is also considered, see [ChronoLocalDate.compareTo(java.time.chrono.ChronoLocalDate)](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html#compareTo-java.time.chrono.ChronoLocalDate-).

**Specified by:**

[compareTo](https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html#compareTo-T-) in interface [Comparable](https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html)<[ChronoLocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html)>

**Specified by:**

[compareTo](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html#compareTo-java.time.chrono.ChronoLocalDate-) in interface [ChronoLocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html)

**Parameters:**

other - the other date to compare to, not null

**Returns:**

the comparator value, negative if less, positive if greater

#### isAfter

public boolean isAfter([ChronoLocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html) other)

Checks if this date is after the specified date.

This checks to see if this date represents a point on the local time-line after the other date.

LocalDate a = LocalDate.of(2012, 6, 30);

LocalDate b = LocalDate.of(2012, 7, 1);

a.isAfter(b) == false

a.isAfter(a) == false

b.isAfter(a) == true

This method only considers the position of the two dates on the local time-line. It does not take into account the chronology, or calendar system. This is different from the comparison in [compareTo(ChronoLocalDate)](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#compareTo-java.time.chrono.ChronoLocalDate-), but is the same approach as [ChronoLocalDate.timeLineOrder()](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html#timeLineOrder--).

**Specified by:**

[isAfter](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html#isAfter-java.time.chrono.ChronoLocalDate-) in interface [ChronoLocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html)

**Parameters:**

other - the other date to compare to, not null

**Returns:**

true if this date is after the specified date

#### isBefore

public boolean isBefore([ChronoLocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html) other)

Checks if this date is before the specified date.

This checks to see if this date represents a point on the local time-line before the other date.

LocalDate a = LocalDate.of(2012, 6, 30);

LocalDate b = LocalDate.of(2012, 7, 1);

a.isBefore(b) == true

a.isBefore(a) == false

b.isBefore(a) == false

This method only considers the position of the two dates on the local time-line. It does not take into account the chronology, or calendar system. This is different from the comparison in [compareTo(ChronoLocalDate)](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#compareTo-java.time.chrono.ChronoLocalDate-), but is the same approach as [ChronoLocalDate.timeLineOrder()](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html#timeLineOrder--).

**Specified by:**

[isBefore](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html#isBefore-java.time.chrono.ChronoLocalDate-) in interface [ChronoLocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html)

**Parameters:**

other - the other date to compare to, not null

**Returns:**

true if this date is before the specified date

#### isEqual

public boolean isEqual([ChronoLocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html) other)

Checks if this date is equal to the specified date.

This checks to see if this date represents the same point on the local time-line as the other date.

LocalDate a = LocalDate.of(2012, 6, 30);

LocalDate b = LocalDate.of(2012, 7, 1);

a.isEqual(b) == false

a.isEqual(a) == true

b.isEqual(a) == false

This method only considers the position of the two dates on the local time-line. It does not take into account the chronology, or calendar system. This is different from the comparison in [compareTo(ChronoLocalDate)](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#compareTo-java.time.chrono.ChronoLocalDate-) but is the same approach as [ChronoLocalDate.timeLineOrder()](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html#timeLineOrder--).

**Specified by:**

[isEqual](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html#isEqual-java.time.chrono.ChronoLocalDate-) in interface [ChronoLocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html)

**Parameters:**

other - the other date to compare to, not null

**Returns:**

true if this date is equal to the specified date

#### equals

public boolean equals([Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html) obj)

Checks if this date is equal to another date.

Compares this LocalDate with another ensuring that the date is the same.

Only objects of type LocalDate are compared, other types return false. To compare the dates of two TemporalAccessor instances, including dates in two different chronologies, use [ChronoField.EPOCH\_DAY](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/ChronoField.html#EPOCH_DAY) as a comparator.

**Specified by:**

[equals](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html#equals-java.lang.Object-) in interface [ChronoLocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html)

**Overrides:**

[equals](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#equals-java.lang.Object-) in class [Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html)

**Parameters:**

obj - the object to check, null returns false

**Returns:**

true if this is equal to the other date

**See Also:**

[Object.hashCode()](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#hashCode--), [HashMap](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)

#### hashCode

public int hashCode()

A hash code for this date.

**Specified by:**

[hashCode](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html#hashCode--) in interface [ChronoLocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html)

**Overrides:**

[hashCode](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#hashCode--) in class [Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html)

**Returns:**

a suitable hash code

**See Also:**

[Object.equals(java.lang.Object)](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#equals-java.lang.Object-), [System.identityHashCode(java.lang.Object)](https://docs.oracle.com/javase/8/docs/api/java/lang/System.html#identityHashCode-java.lang.Object-)

#### toString

public [String](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html) toString()

Outputs this date as a String, such as 2007-12-03.

The output will be in the ISO-8601 format uuuu-MM-dd.

**Specified by:**

[toString](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html#toString--) in interface [ChronoLocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/ChronoLocalDate.html)

**Overrides:**

[toString](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#toString--) in class [Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html)

**Returns:**

a string representation of this date, not null

[SKIP NAVIGATION LINKS](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html" \l "skip.navbar.bottom" \o "Skip navigation links)

* [OVERVIEW](https://docs.oracle.com/javase/8/docs/api/overview-summary.html)
* [PACKAGE](https://docs.oracle.com/javase/8/docs/api/java/time/package-summary.html)
* CLASS
* [USE](https://docs.oracle.com/javase/8/docs/api/java/time/class-use/LocalDate.html)
* [TREE](https://docs.oracle.com/javase/8/docs/api/java/time/package-tree.html)
* [DEPRECATED](https://docs.oracle.com/javase/8/docs/api/deprecated-list.html)
* [INDEX](https://docs.oracle.com/javase/8/docs/api/index-files/index-1.html)
* [HELP](https://docs.oracle.com/javase/8/docs/api/help-doc.html)

**Java™ Platform  
Standard Ed. 8**

* [**PREV CLASS**](https://docs.oracle.com/javase/8/docs/api/java/time/Instant.html)
* [**NEXT CLASS**](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDateTime.html)
* [FRAMES](https://docs.oracle.com/javase/8/docs/api/index.html?java/time/LocalDate.html)
* [NO FRAMES](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html)
* [ALL CLASSES](https://docs.oracle.com/javase/8/docs/api/allclasses-noframe.html)
* SUMMARY:
* NESTED |
* [FIELD](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#field.summary) |
* CONSTR |
* [METHOD](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#method.summary)
* DETAIL:
* [FIELD](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#field.detail) |
* CONSTR |
* [METHOD](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#method.detail)

[Submit a bug or feature](http://bugreport.sun.com/bugreport/)  
For further API reference and developer documentation, see [Java SE Documentation](https://docs.oracle.com/javase/8/docs/index.html). That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.  
[Copyright](https://docs.oracle.com/javase/8/docs/legal/cpyr.html) © 1993, 2021, Oracle and/or its affiliates. All rights reserved. Use is subject to [license terms](http://download.oracle.com/otndocs/jcp/java_se-8-mrel-spec/license.html). Also see the [documentation redistribution policy](http://www.oracle.com/technetwork/java/redist-137594.html). Modify Preferencias sobre cookies. Modify [Ad Choices](https://www.oracle.com/legal/privacy/marketing-cloud-data-cloud-privacy-policy.html#12).