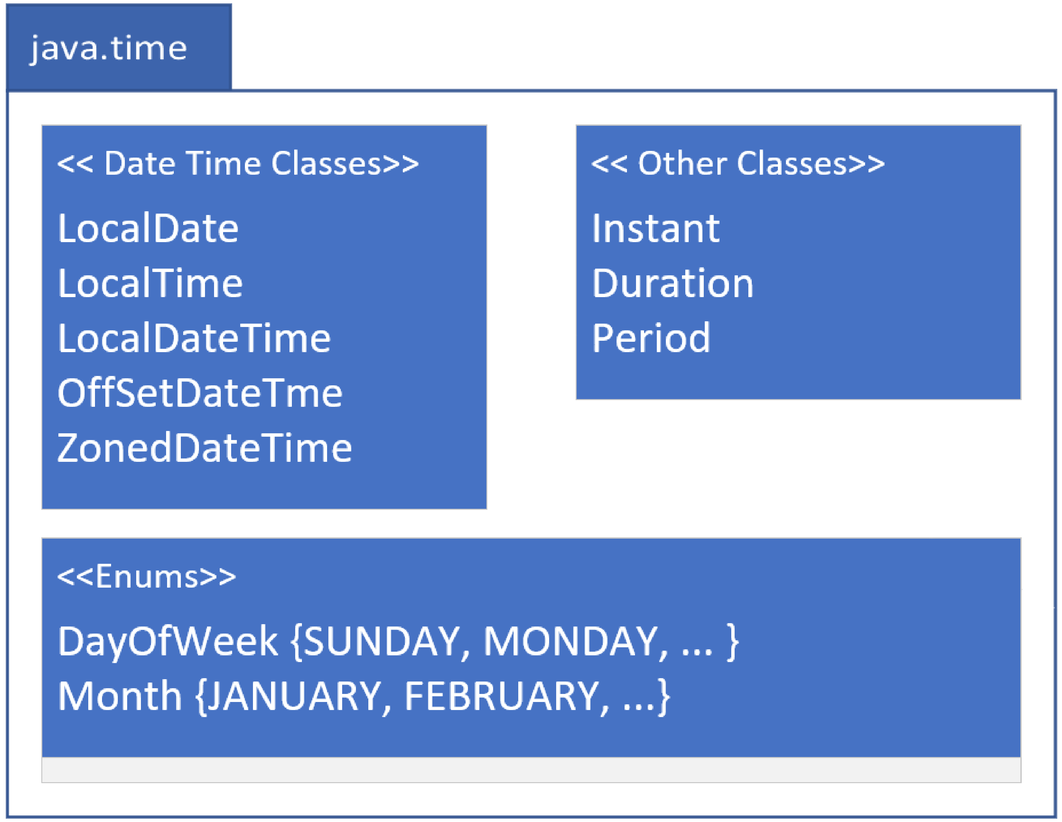
java.time


java.time
<< Date Time Classes>>
LocalDate
LocalTime
LocalDateTime
OffSetDateTme
ZonedDateTime
<< Other Classes>>
Instant
Duration
Period
<<Enums>>
DayOfWeek {SUNDAY, MONDAY, ... }
Month {JANUARY, FEBRUARY, ...}

java.time related packages

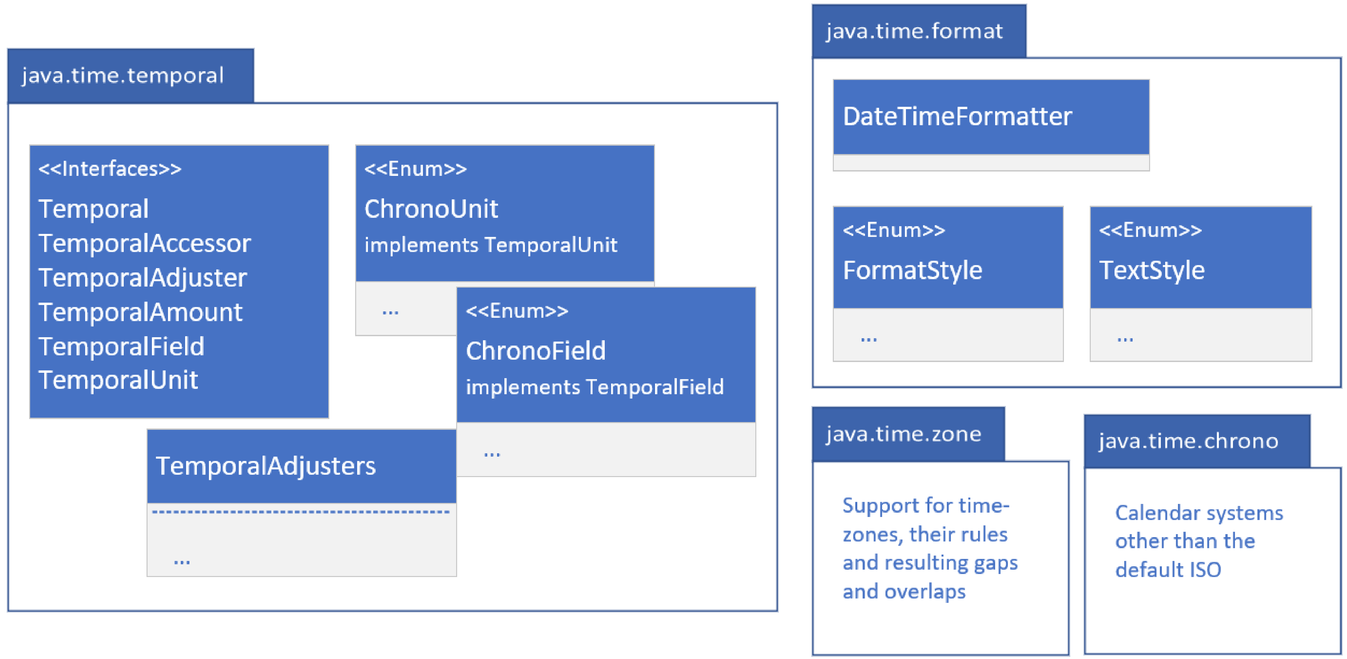
Java has other packages under the java.time umbrella, as shown on this slide.

These are java.time.temporal, and java.time.format.

You'll likely be using these two packages, and the types within them.

In addition, there are the java.time.zone and java.time.chrono packages.

You're less likely to need the functionality specifically related to these last two packages, so I won't be covering them, but you should be aware of them.

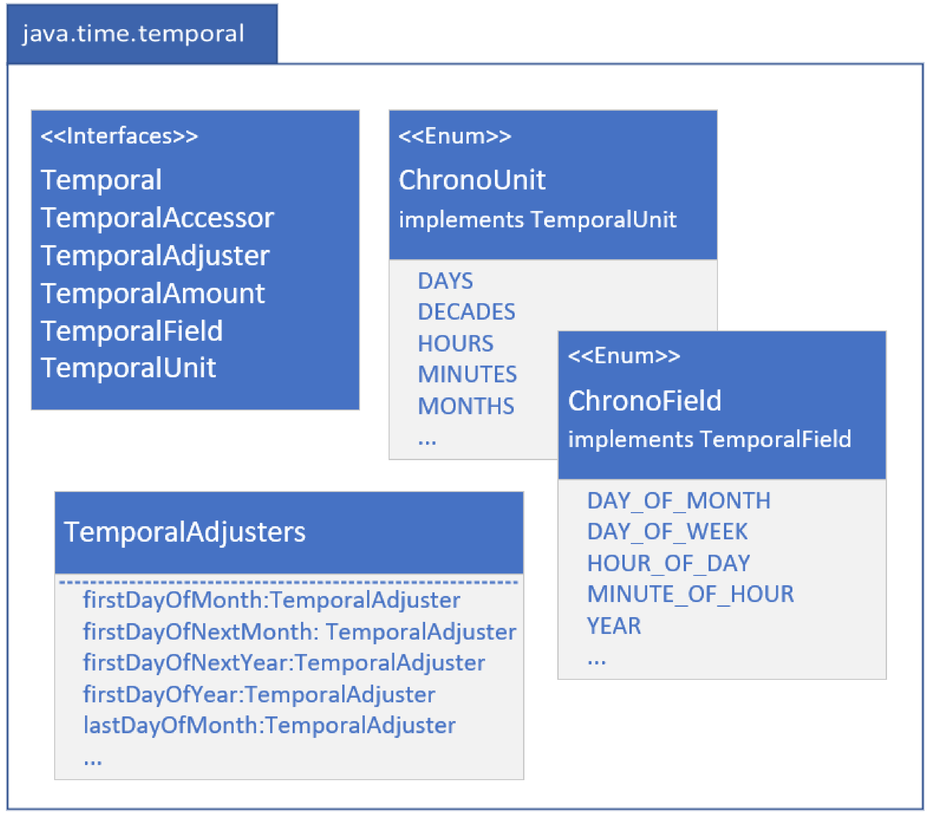


java.time.temporal

The java.time.temporal package contains important interfaces, that the java.time classes implement.

These include the Temporal, and TemporalAccessor interfaces, that describe a uniform way to read from, or write to, a date time object.

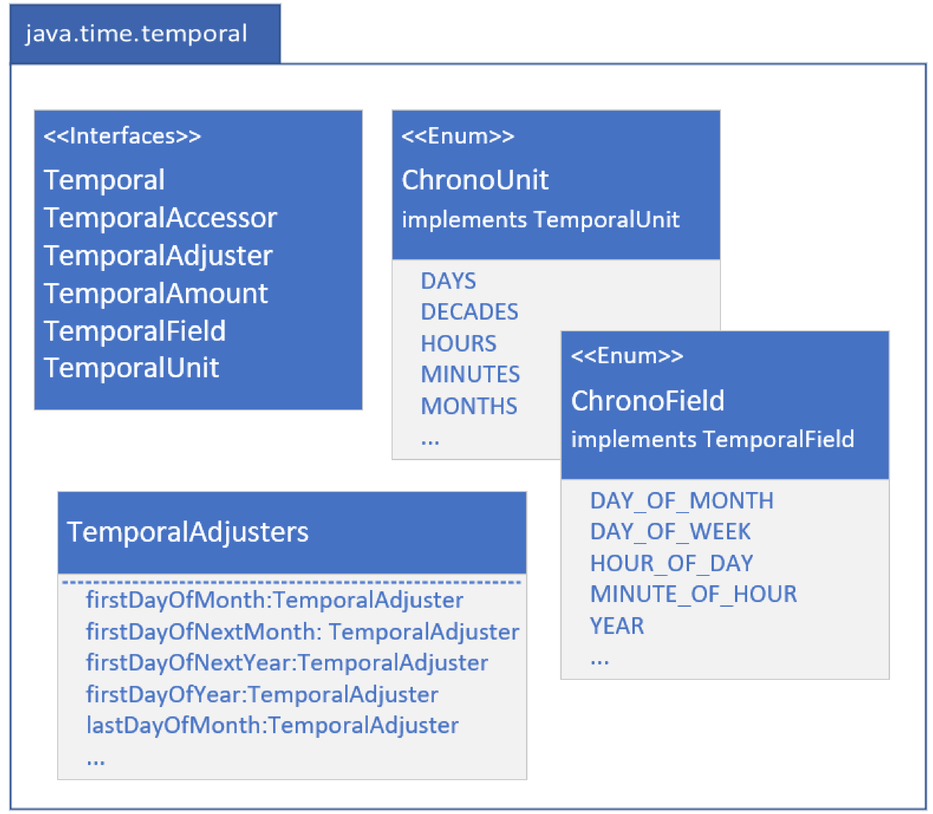
The TemporalAdjuster, TemporalAmount, TemporalField and TemporalUnit interfaces are quite often used as method parameters, to specify what kind of information you want specifically about a date time object.



java.time.temporal

There are two enums in this package that I'll be covering, and shown here, ChronoField and ChronoUnit.

The TemporalAdjusters class is a helper class to return specific implementations of TemporalAdjuster, which can give you helpful dates such as first day of month, or last day of year, etc.

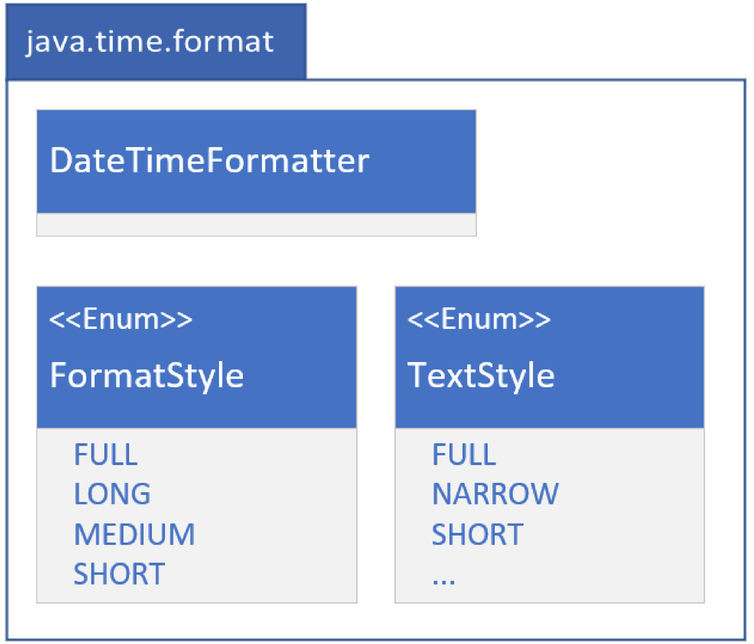


java.time.format

I've shown you how to format date time, using the **"%t"** specifier, in a formatted String.

The java.time.format package gives you a great deal more options, as well as support for localization, which I'll be covering a little bit later in this section.

There are enums to support dates and time in defined styles, called Full, long, medium and short, and I'll show you these in a couple of examples in a bit.



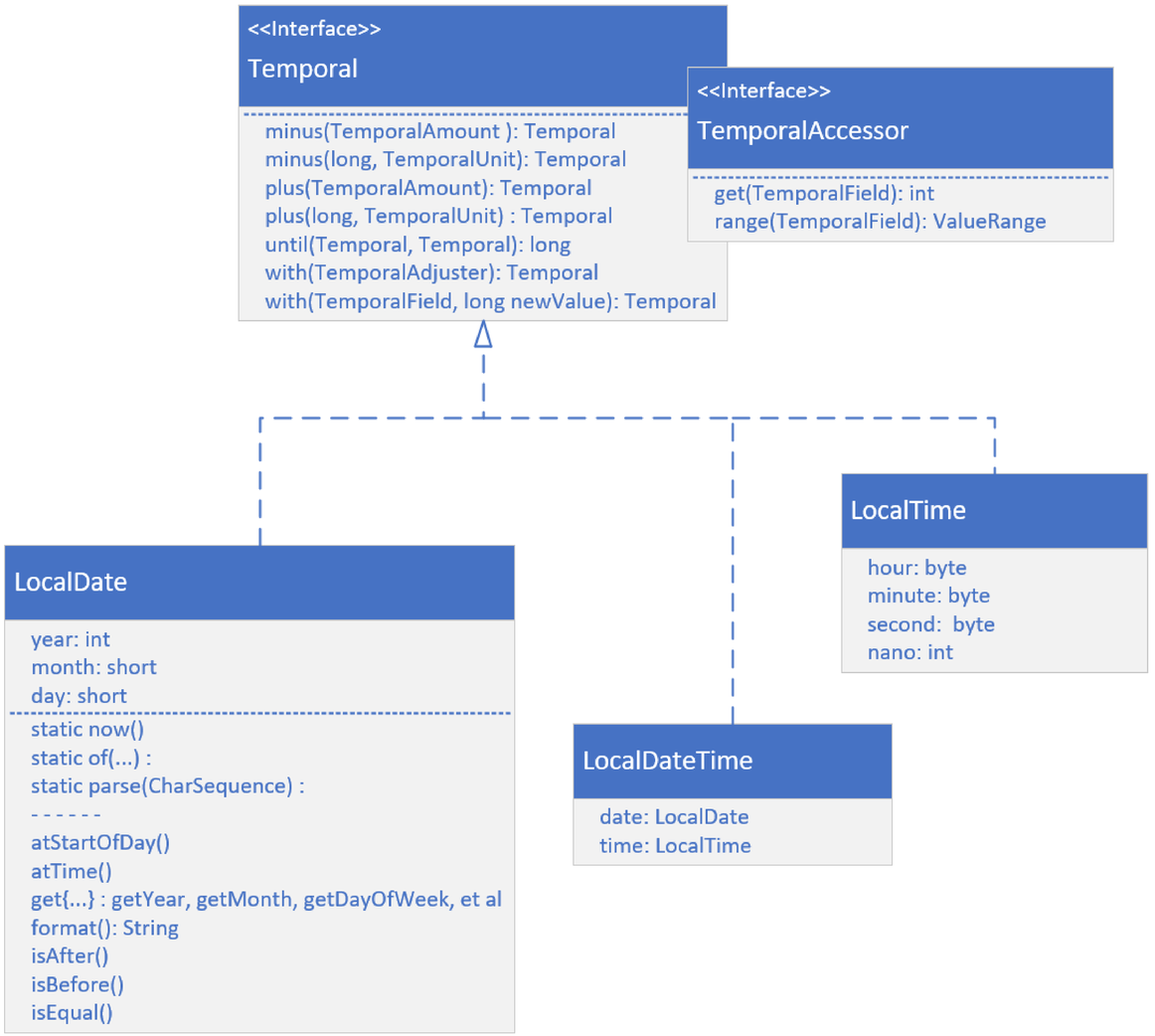
LocalDate, LocalTime and LocalDateTime

On this slide, I'm showing the most common classes for Date and Time, when you don't need to include time zone data.

These are LocalDate, Local Time, and LocalDateTime.

Each implements both the Temporal and the TemporalAccessor interfaces, and the methods on those interfaces, as I'm showing here.

These include the get and range methods from TemporalAccessor.



LocalDate, LocalTime and LocalDateTime

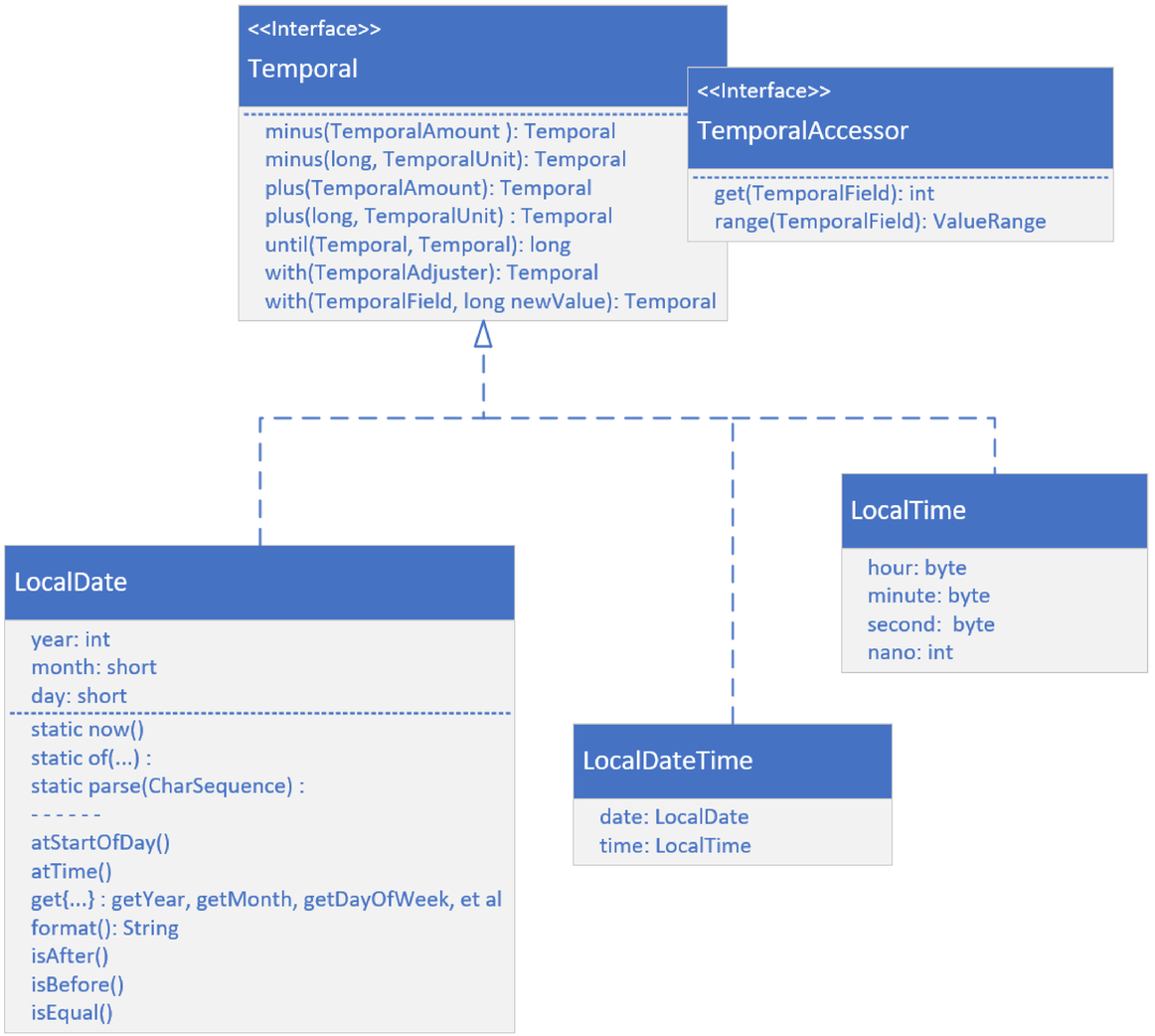
From Temporal, there are methods to add or subtract units of time from your objects.

These are the plus and minus methods.

Field values can be directly set on a returned copy, using the with methods.

I'm also showing methods on LocalDate, that in general, are methods available in some form, on any of the temporal objects.

These have prefixes such as: at, get, and is.



Creating instances of Temporal Implementations

There are several static methods, which return a new instance of a temporal class.

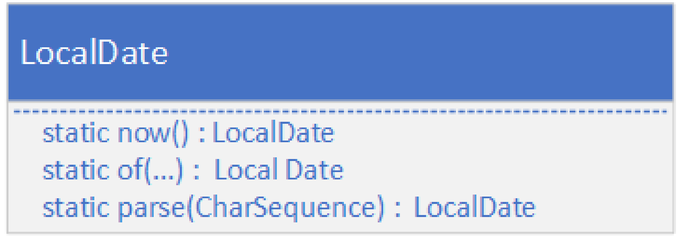
There is the `now` method, that gives you a representation of the current moment, for the class you specify.

You can create instances, using any of the several overloaded versions of the **of**, factory method.

Alternately, you can use the `parse` method, with a character sequence, usually a String, that's formatted in a predetermined way, or you can pass in a defined format.

The methods shown on this slide are purposely shown without too much detail.

You can use similarly named methods on any of the temporal implementations, with varying parameters and return types.



Temporal Instances Are Immutable

All temporal instances of classes, in the java.time package, are immutable and thread-safe.

A new instance is returned, from methods that write to a temporal instance, such as the plus or minus methods, for example.

You'll need to assign the result of these methods to a variable.

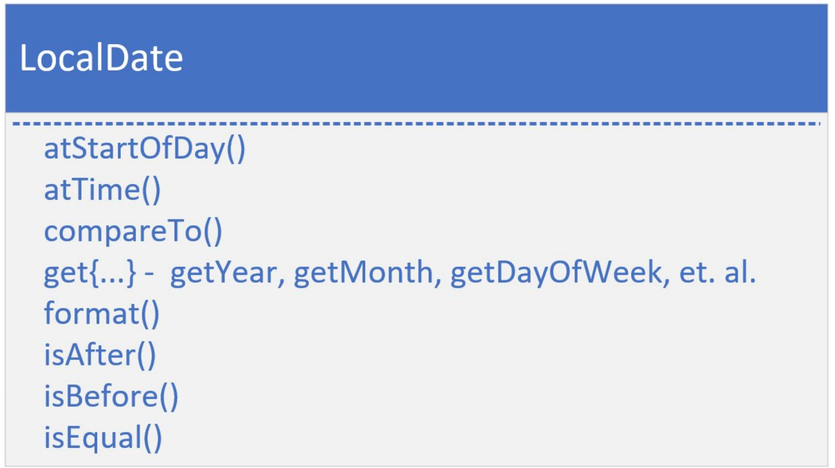
The original instance won't be modified.

Methods of Temporal Implementations

In addition to the methods on the Temporal and TemporalAccessor interfaces, most implementations have methods prefixed with at, get, and is, as well as a format method.

The at methods allow you to combine temporal instances, in this example, a time is combined with the LocalDate for both the atStartOfDay and atTime methods, and return LocalDateTime instances.

The Date and Time instances implement Comparable, so each has a compareTo method



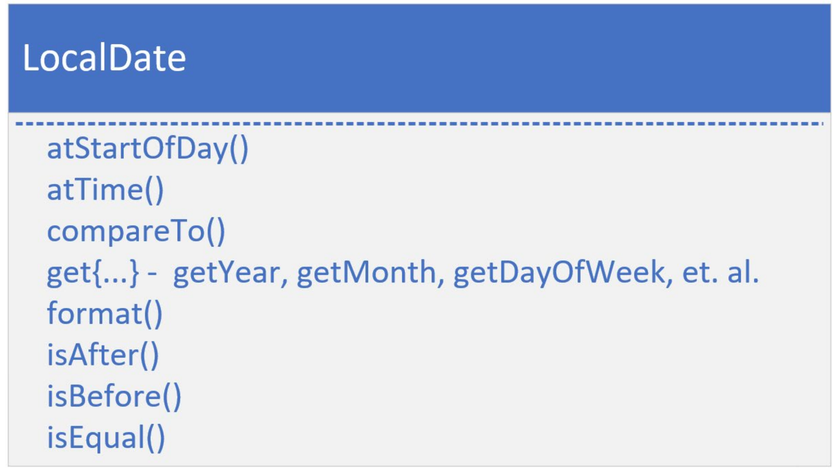
Methods of Temporal Implementations

The get methods are specific to the class, so for LocalDate, you'd have date related getters, like getYear, getMonth, getDayofWeek, and so on.

For LocalTime, these would be getHour, getMinute, etc.

Each class supports isAfter, isBefore and isEqual, so you can compare units of date or time.

And there's a format method, to output a formatted date.



LocalDate, LocalTime, and LocalDateTime classes

On this slide, I want you to see that Java is internally storing the date fields, and the time fields, as separate numeric fields.

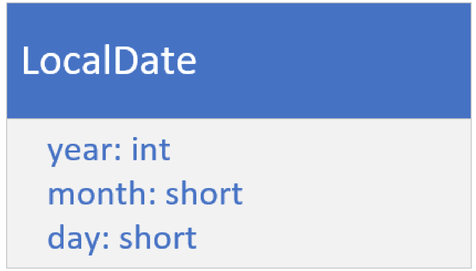
These fields can be retrieved, with or without context, of the other fields.

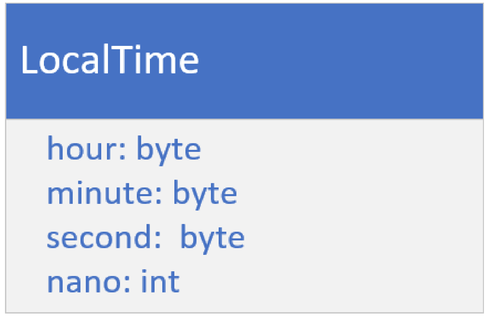
For example, you can get day of the month, or day of the year.

LocalDate uses an int for year, and a short for month and day.

LocalTime uses bytes for the hour, minute, and second, and an int for nanoseconds.

LocalDateTime has two fields, a date, with a type of LocalDate, and a time, with a type of LocalTime.



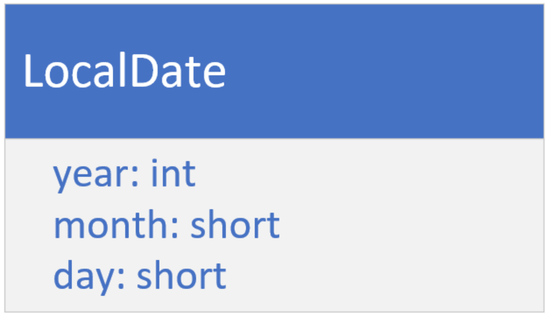


The LocalDate Class

LocalDate is a class for storing and managing a date, with a year, month, and day, without reference to a specific time zone.

This kind of date might be used for an anniversary date, a birth date, or a special holiday like Cinco de Mayo (the 5th of May), or Thanksgiving Day.

This class doesn't have a clock time component.



The LocalTime Class

The LocalTime class provides a description of the local time as seen on a wall clock.

It contains neither a date or a time zone.

Time is represented as hour-minute-second, with nanosecond precision, if it's available.