Set Your Clocks the Java Date Time API is Here

Daniel Hinojosa

Who is this for?

- Java/Groovy/Scala/Clojure Developers interested in Java 8 APIs
- Those who enjoyed or currently enjoy Joda Time
- Those who have burned by `java.util.Date` and `java.util.Calendar`

Introduction to some terms

- Epochs
- Time Keepers
- ISO 8601
- Java Calendars and Dates
- Joda Time

Epoch

- Serves as a reference point from which time is measured
- The beginning of time of a particular context
- Current and historical calendar eras exist, each with its own epoch
- Based mostly on religious or political milestones

Political Asian Epochs & Calendars

Japan

- Japanese based eras
- Each epoch begins with the current ascension of the current emperor
- Current era, Heisei era begins January 8, 1989 after the death of Hirohito

China

- Followed similar rules to Japan until 1912
- After 1912 : Gregorian Calendar adopted in the Republican Era
- Common Era : People’s Republic of China established, continues to use Gregorian Calendar

India:

- The Saka Era. Vernal Equinox since year 78.

North Korea

- Starts 1912: but because Kim Il-Sung was born on that date.

Thailand

- Founding of Bangkok in 1782, but in 1941 uses 543BC started Thai Solar Calendar.

Religious Epochs & Calendars

Judaism

- Anno Mundi - Hebrew Calendar date from "Creation" March 18, 3952 BC/BCE

Islam

- Anno Hegiræ - The year of the Hijra (the Migration of Mohammed to Medina in 622)

Hindu

- Bikram Samvat (Bikram Sambat)
  - Calendar established by Indian emperor Vikramaditya.
  - It is a popularly used calendar in India and the official calendar of Bangladesh and Nepal
- Hindu Calendar
  - Various Calendars
    - Nepali Calendar
    - Assamese Calendar
    - Bengali Calendar
    - Malayalam Calendar
    - Tamil Calendar
    - Telugu Calendar
    - Kannada Calendar
  - All they have in common are the same months
  - The month that starts each year is different
  - Basis for the Buddhist Calendar

- Yuga
  - Four age cycle epochs
  - Life in the universe is created and destroyed every 4.1 to 8.2 billion years
  - One full day and night for Brahma

European Calendar

- Roman Calendar
  - Changed Various Times
  - Year Started on March 1
  - Consisted of 304 days
- Julian Calendar
  - Julius Caesar
  - Established 46 BC
  - 365.25 days

Christianity

- Anno Domini (AD)
- Used Globally
- Associated with the incarnation of Jesus
- Origins of what is known as civil time
- Gregorian Calendar
  - Pope Gregory XIII
  - Refines the “leap year” to 365.2425 (0.002%)

Computer Epochs

- Matlab, Turbo DB, tdbengine – January 1, 0
- Symbian, Go, .NET, RXX, Rata Die – January 1, 1
- NTFS, COBOL, Win32/64 – January 1, 1601
- MS SQL Server – January 1, 1753
- Microsoft Products, Lotus 1-2-3 - January 0, 1900.
- Common LISP, CICS, Network Time Protocol – January 1, 1900
- LabView, Mac OS ⇐ 9.0, Palm – January 1, 1904

source: http://en.wikipedia.org/wiki/Epoch_(reference_date) (http://en.wikipedia.org/wiki/Epoch_(reference_date))

The *nix/Java Standard

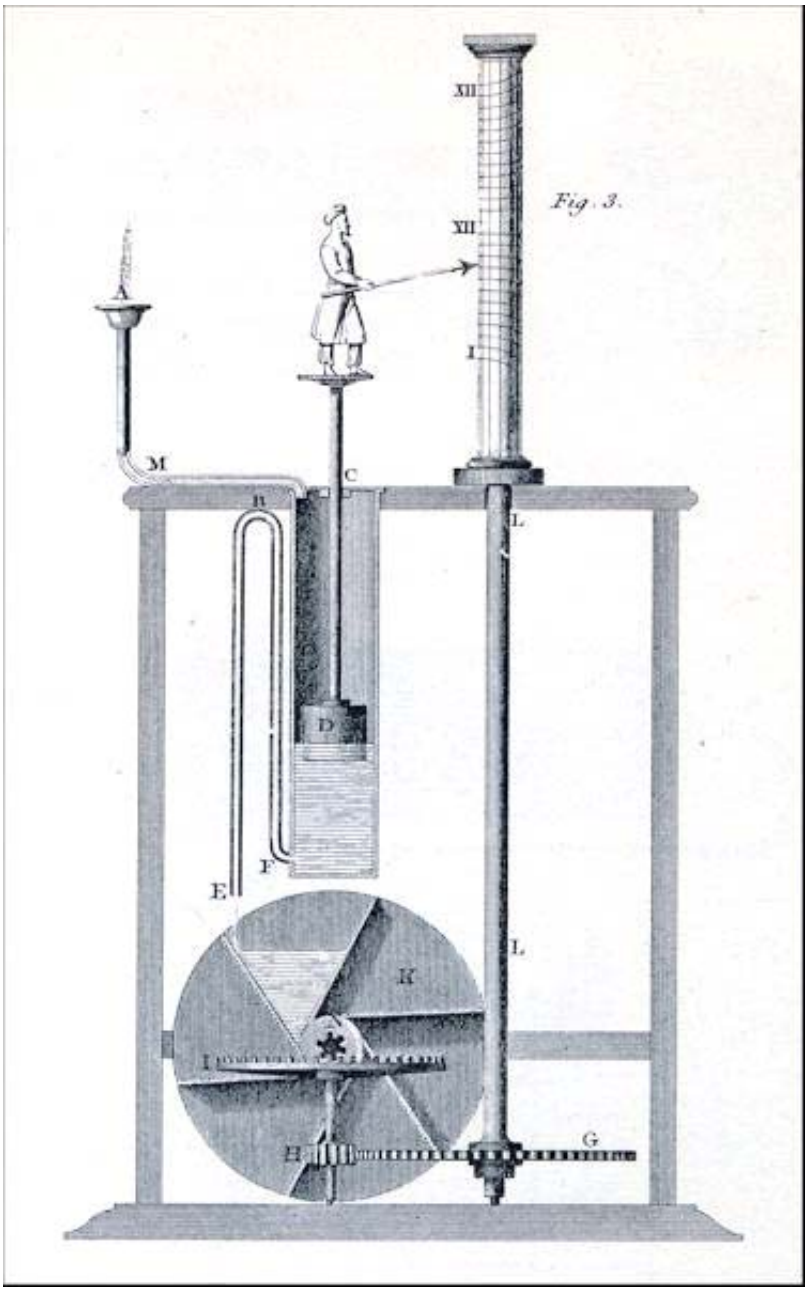
- January 1, 1970 Midnight GMT.
- Unix Time – The Day Unix was born
- Unix, Linux, Ruby, Mac OS X, Java, JavaScript, C, PHP
- Windows doesn't adhere to Unix Time

How we measure time

Sundials

Water Clocks/Clepsydras


ΑΝΤΙΟΤΙΔΟΣ
XX

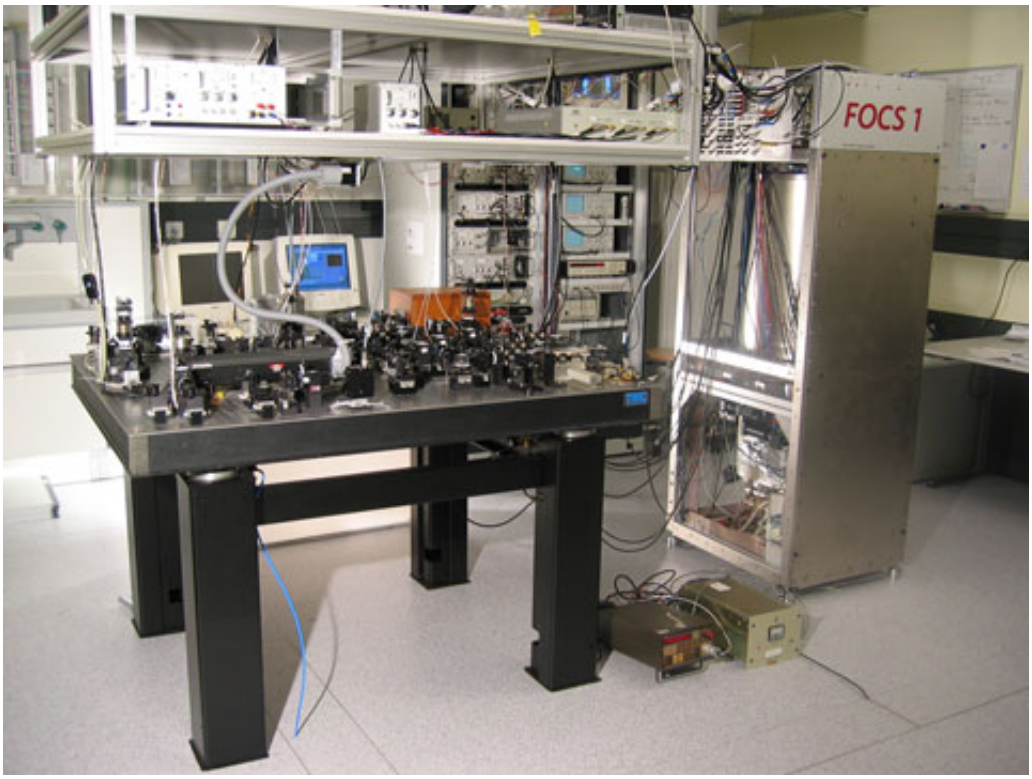


Pendulum

Quartz

32,768 Hz

Atomic Clock



Cesium 133: 9,192,631,770 cycles of radiation

ISO 8601 Standard

- Standard and Collaborative means of managing date and time
- Based on the cesium-133 atom atomic clock

ISO 8601 Formats

Format	Example
Date	2014-01-01
Combined Date and Time in UTC	2014-07-07T07:01Z
Combined Date and Time in MDT	2014-07-07T07:38:51.716-06:00
Date With Week Number	2014-W27-3
Ordinal Date	2014-188

Duration	P3Y6M4DT12H30M5S
Finite Interval	2014-03-01T13:00:00Z/2015-05-11T15:30:00Z
Finite Start with Duration	2014-03-01T13:00:00Z/P1Y2M10DT2H30M
Duration with with Finite End	P1Y2M10DT2H30M/2015-05-11T15:30:00Z

Kxcd ISO-8601

GMT v. UTC

GMT

- Greenwich Mean Time
- Mean solar time at the Royal Observatory in Greenwich, London.

- Britain had been the world’s time keeper since Victorian Age.
- It is commonly used in practice to refer to UTC (not accurate)
- Side Note: GMT used in winter, British Summer Time in summer

---

UTC

- Coordinated Universal Time
- Backronym: "Universal Time, Coordinated"
- Time Standard based on International Atomic Time (TAI)
- Universally accepted
- UTC replaced GMT as the basis for the main reference time scale or civil time in various regions on 1 January 1972.
- GMT doesn’t address sub-second levels of time.
- Backed with Atomic Time

Life and Times Java

`java.util.Date`

- Introduced millisecond resolution
- java.util.Date
- What was wrong with it?
  - Constructors that accept year arguments require offsets from 1900, which has been a source of bugs.
  - January is represented by 0 instead of 1, also a source of bugs.
  - Date doesn’t describe a date but describes a date-time combination.

- Date's mutability makes it unsafe to use in multithreaded scenarios without external synchronization.
- Date isn't amenable to internationalization.

Source: http://www.javaworld.com/article/2078757/java-se/java-101-the-next-generation-it-s-time-for-a-change.html (http://www.javaworld.com/article/2078757/java-se/java-101-the-next-generation-it-s-time-for-a-change.html)

`java.util.Calendar`

- Introduced in Java 1.1
- What is wrong with it?
  - It isn't possible to format a calendar.
  - January is represented by 0 instead of 1, a source of bugs.
  - Calendar isn't type-safe; for example, you must pass an int-based constant to the get(int field) method. (In fairness, enums weren't available when Calendar was released.)
  - Calendar's mutability makes it unsafe to use in multithreaded scenarios without external synchronization. (The companion java.util.TimeZone and java.text.DateFormat classes share this problem.)
  - Calendar stores its state internally in two different ways — as a millisecond offset from the epoch and as a set of fields — resulting in many bugs and performance issues.

Source: http://www.javaworld.com/article/2078757/java-se/java-101-the-next-generation-it-s-time-for-a-change.html (http://www.javaworld.com/article/2078757/java-se/java-101-the-next-generation-it-s-time-for-a-change.html)

Of course then there is this:

```
> new java.util.GregorianCalendar

java.util.GregorianCalendar =
java.util.GregorianCalendar[time=1393764079082,areFieldsSet=true,ar
eAllFieldsSet=true,lenient=true,zone=sun.util.calendar.ZoneInfo[id=
"America/New_York",offset=-18000000,dstSaving=3600000,useDaylight=t
rue,transitions=235,lastRule=java.util.SimpleTimeZone[id=America/Ne
w_York,offset=-18000000,dstSaving=3600000,useDaylight=true,startYea
r=0,startMode=3,startMonth=2,startDay=8,startDayOfWeek=1,startTime=
7200000,startTimeMode=0,endMode=3,endMonth=10,endDay=1,endDayOfWeek
=1,endTime=7200000,endTimeMode=0]],firstDayOfWeek=1,minimalDaysInFi
rstWeek=1,ERA=1,YEAR=2014,MONTH=2,WEEK_OF_YEAR=10,WEEK_OF_MONTH=2,D
AY_OF_MONTH=2,DAY_OF_YEAR=61,DAY_OF_WEEK=1,DAY_OF_WEEK_IN_MONTH=1,A
M_PM=0,HOUR=7,HOUR_OF_DAY=7,MINUTE=41,SECOND=19,MILLISECOND=82,ZONE
_OFFSET=-18000000,DST_...
```

What was cool about Joda Time

- Straight-forward instantiation and methods
- UTC/ISO 8601 Based, not Gregorian Calendar
- Has support for other calendar systems if you need it (Julian, Gregorian-Julian, Coptic, Buddhist)
- Includes classes for date times, dates without times, times without dates, intervals and time periods.
- Advanced formatting
- Well documented and well tested
- Immutable!
- Months are 1 based

About the Java 8 Date Time API

- Authored by the same team as Joda Time

- Immutable & Threadsafe
- Learned from previous mistakes made in Joda Time
- There are no *constructors* (Dude what?)
- Nanosecond Resolution

The Java Date Time Packaging

- `java.time` - Base package for managing date time
- `java.time.chrono` - Package that handles alternative calendering and chronology systems
- `java.time.format` - Package that handles formatting of dates and times
- `java.time.temporal` - Package that allows us to query dates and times

Date Time Conventions

- `of` - static factory usually validating input parameters not converting them
- `from` - static factory that converts to an instance of a target class
- `parse` - static factory that parses an input string
- `format` - uses a specified formatter to format the date
- `get` - Returns part of the state of the target object
- `is` - Queries the state of the object
- `with` - Returns a copy of the object with one element changed, this is the immutable equivalent
- `plus` - Returns a copy of the target object with the amount of time added
- `minus` - Returns a copy of the target object with the amount of time subtracted
- `to` - Converts this object to another object type
- `at` - Combines the object with another

Instant

- Single point in time
- Time since the Unix/Java Epoch `1970-01-01T00:00:00Z`
- Differs from the `java.util.Date` and `long` representation
- Contains two states:
  - `long` of seconds since the Unix Epoch
  - `int` of nano seconds within one second

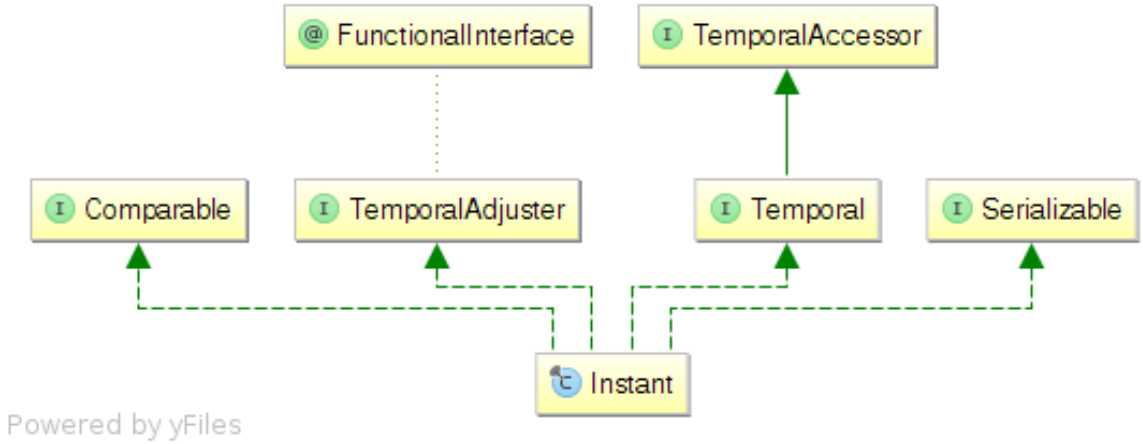
That a lot of resolution!

An `Instant` can be resolved as $1.844674407 \times 10^{19}$ seconds or 584542046090 years!

Some of the basic features of `Instant`

```
Instant now = Instant.now();
System.out.println(now.getEpochSecond());
System.out.println(now.getNano());
System.out.println(Instant.parse("2014-02-20T20:21:20.432Z"));
```

UML Diagram of an `Instant`



Enums

`Month` and `DayOfWeek`

- The Java Date/Time API contains `enum` classes to describe our months and days
  - `Month`
  - `DayOfWeek`

`Month` and `DayOfWeek` Exemplified

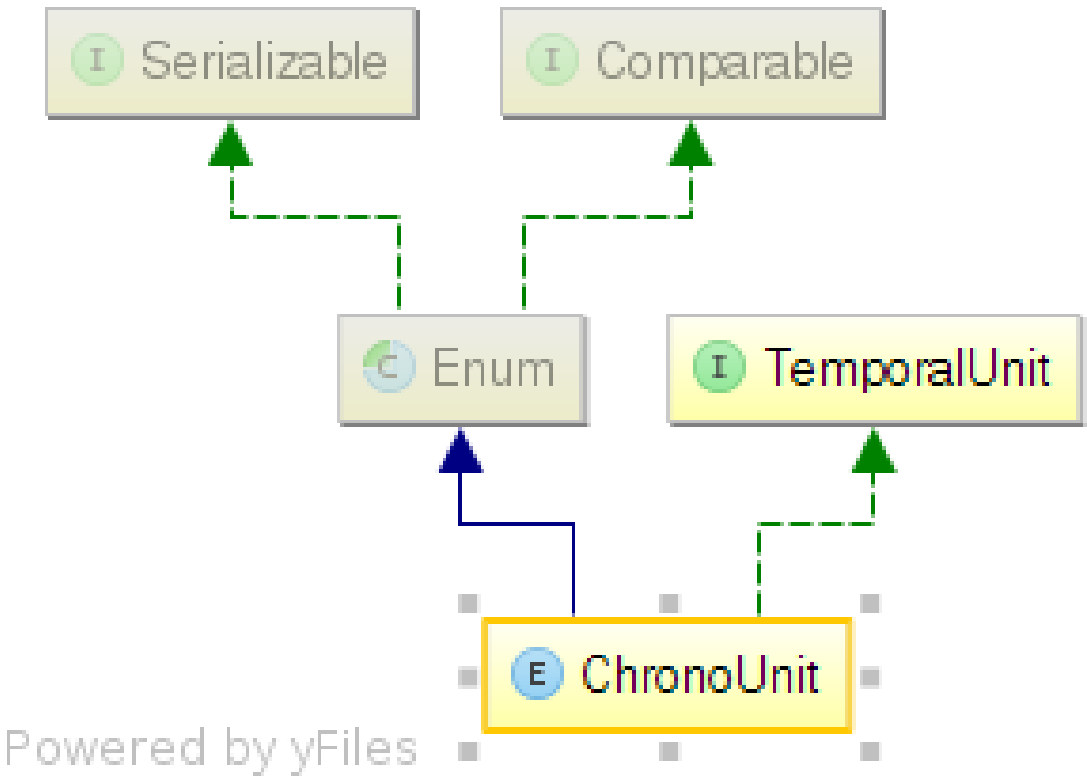
```
DayOfWeek.SUNDAY
DayOfWeek.FRIDAY
```

```
Month.JANUARY
Month.JULY
Month.DECEMBER
```

`ChronoUnit`

- `enum` to represent a unit of time for a scalar
- `implements TemporalUnit`
- `ChronoUnit` is meant to be general enough for various calendars

UML Diagram of `ChronoUnit`



`ChronoUnit` Exemplified

```
ChronoUnit.DAYS
ChronoUnit.CENTURIES
ChronoUnit.ERAS
ChronoUnit.MINUTES
ChronoUnit.MONTHS
ChronoUnit.SECONDS
ChronoUnit.FOREVER
```

```
Instant.now().plus(19, ChronoUnit.DAYS)
```

`ChronoField`

- Represents a field in a date
- Given: `2010-10-22T12:00:13` has six fields
  - The year: `2010`
  - The month: `10`
  - The day of the month: `22`
  - The hour of the day: `12`
  - The minute: `0`
  - The seconds: `13`
- `implements TemporalField`
- `ChronoField` is also meant to be general enough for various calendars

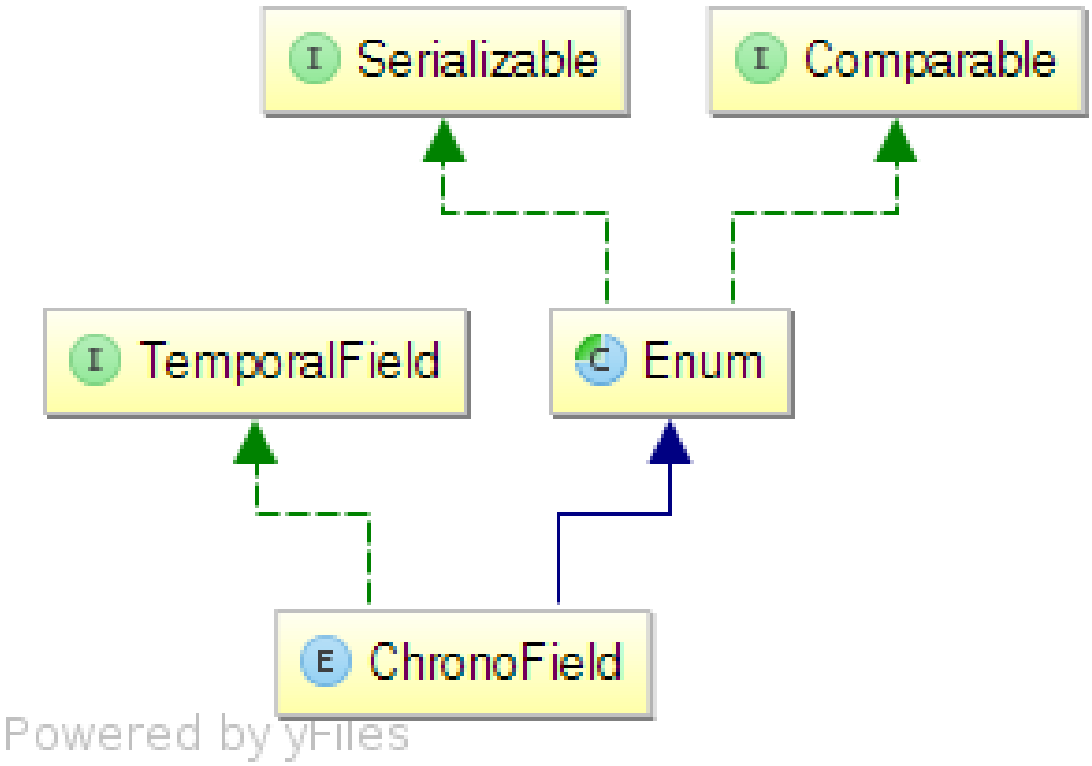
---

`ChronoField` Exemplified

```
ChronoField.MONTH_OF_YEAR
ChronoField.DAY_OF_MONTH
ChronoField.HOUR_OF_DAY
ChronoField.SECOND_OF_MINUTE
ChronoField.SECOND_OF_DAY
ChronoField.MINUTE_OF_DAY
ChronoField.MINUTE_OF_HOUR
```

```
Instant.now.get(ChronoField.HOUR_OF_DAY);
```

UML Diagram of `ChronoField`



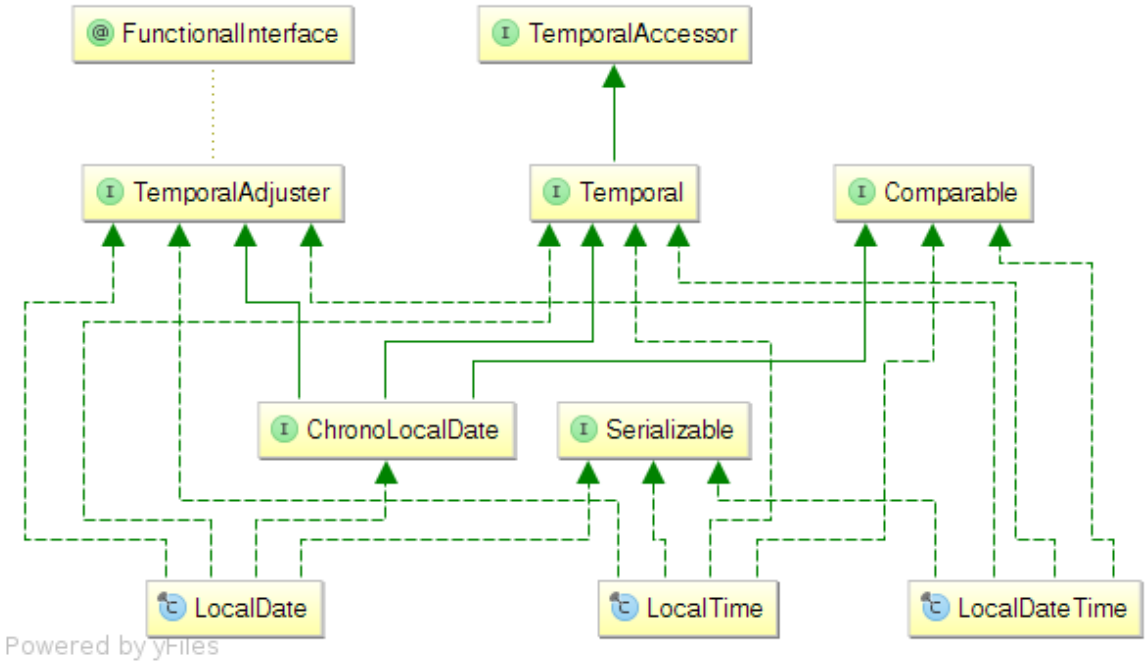
Local Dates and Times

- `LocalDate` - An ISO 8601 date representation without timezone and time
- `LocalTime` - An ISO 8601 time representation without timezone and date

- `LocalDateTime` - An ISO 8601 date and time representation without time zone

---

UML Diagram of `LocalDate`, `LocalTime`, `LocalDateTime`



`LocalDate` exemplified

```
LocalDate february20th = LocalDate.of(2014, Month.FEBRUARY, 20);
february20th;                                                  //2014-02-
20
LocalDate.from(february20th.plus(15, ChronoUnit.YEARS)); //2029-02-
20
LocalDate.parse("2014-11-22");                                 //2014-11-
22
```

`LocalTime` exemplified

```
LocalTime.MIDNIGHT;                                  //00:00
LocalTime.NOON;                                      //12:00
LocalTime.of(23, 12, 30, 500);
//23:12:30.000000500
LocalTime.now();                                     //00:40:34.110
LocalTime.ofSecondOfDay(11 * 60 * 60);               //11:00
LocalTime.from(LocalTime.MIDNIGHT.plusHours(4)); //04:00
```

`LocalDateTime` exemplified

```
LocalDateTime.of(2014, 2, 15, 12, 30, 50, 200); //2014-02-
15T12:30:50.000000200
LocalDateTime.now();                                 //2014-02-
28T17:28:21.002
LocalDateTime.from(
   LocalDateTime.of
      (2014, 2, 15, 12, 30, 40, 500)
        .plusHours(19)));                            //2014-02-
16T07:30:40.000000500
LocalDateTime.MIN;                                   //-999999999-01-
01T00:00
LocalDateTime.MAX;                                   //+999999999-12-
31T23:59:59.999999999
```

`ZonedDateTime`

- Specifies a complete date and time in a particular time zone
- Contains methods that can convert from `LocalDate`, `LocalTime`, and `LocalDateTime` to `ZonedDateTime`

But first, `ZoneId`

- `ZoneId` represents the IANA Time Zone Entry
- http://www.iana.org/time-zones (http://www.iana.org/time-zones)
- Download tar.gz file, locate the region file (e.g. northamerica)
- TimeZone names are divided by region

```
# Monaco
# Shanks & Pottenger give 0:09:20 for Paris Mean Time; go with
Howse's
# more precise 0:09:21.
# Zone  NAME            GMTOFF  RULES   FORMAT  [UNTIL]
Zone    Europe/Monaco   0:29:32 -       LMT     1891 Mar 15
                        0:09:21 -       PMT     1911 Mar 11    #
Paris Mean Time
                        0:00    France  WE%sT   1945 Sep 16 3:00
                        1:00    France  CE%sT   1977
                        1:00    EU      CE%sT
```

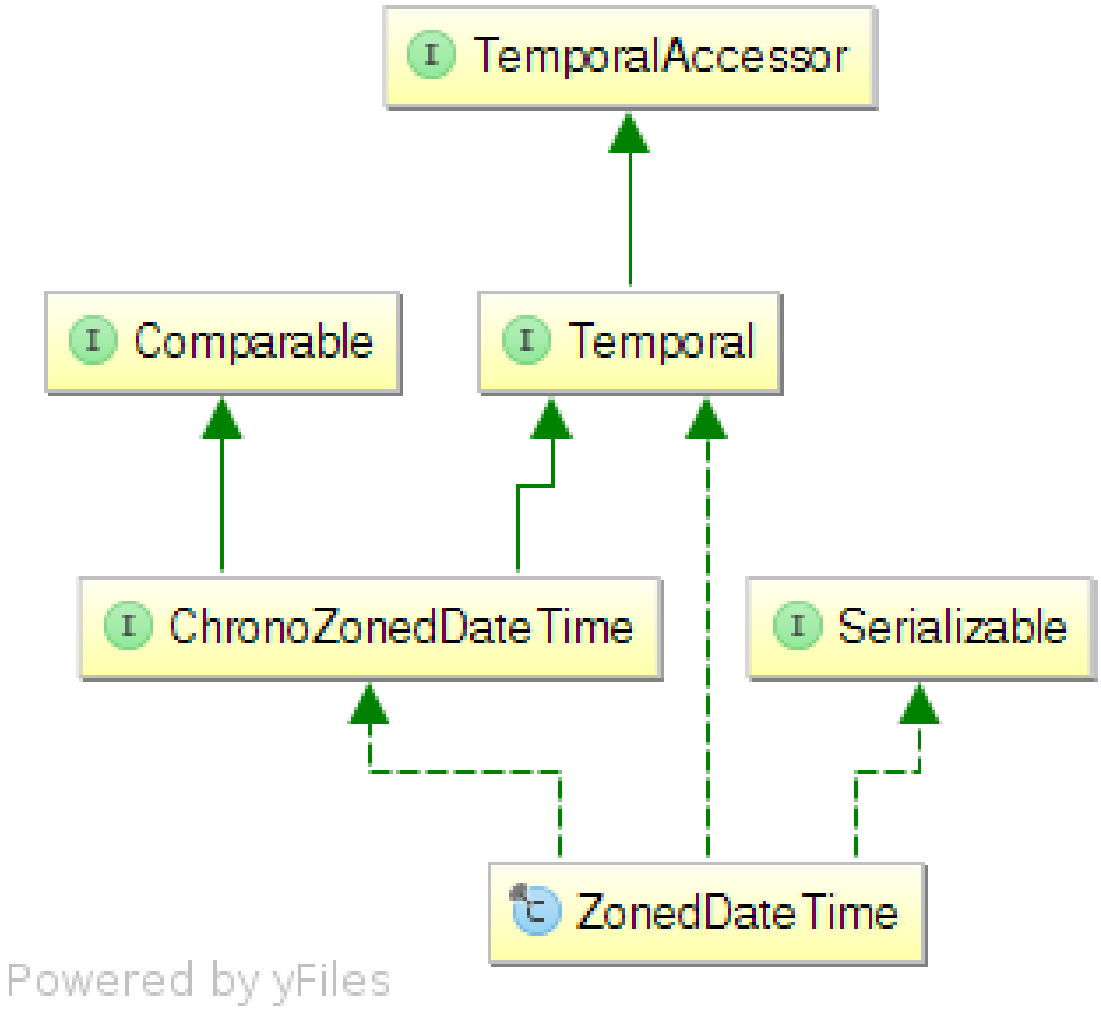
Creating the `ZoneId`

```
ZoneId.of("America/Denver");
ZoneId.of("Asia/Jakarta");
ZoneId.of("America/Los_Angeles");
ZoneId.ofOffset("UTC", ZoneOffset.ofHours(-6));
```

Creating your `ZoneId` naming

```
Map<String, String> map = new HashMap<String, String>();
map.put("Pacific", "America/Los_Angeles");
map.put("Mountain", "America/Denver");
map.put("Central", "America/Chicago");
map.put("Eastern", "America/New_York");
ZoneId.of("Mountain", map);  // Same as "America/Denver"
```

UML Diagram of `ZonedDateTime`



`ZonedDateTime` exemplified

```
ZonedDateTime.now();  //Current Date Time with Zone

ZonedDateTime myZonedDateTime = ZonedDateTime.of(2014, 1, 31, 11,
20, 30, 93020122, ZoneId.systemDefault());

ZonedDateTime nowInAthens =
ZonedDateTime.now(ZoneId.of("Europe/Athens"));

LocalDate localDate = LocalDate.of(2013, 11, 12);
LocalTime localTime = LocalTime.of(23, 10, 44, 12882);
ZoneId chicago = ZoneId.of("America/Chicago");
ZonedDateTime chicagoTime = ZonedDateTime.of(localDate, localTime,
chicago);


LocalDateTime localDateTime = LocalDateTime.of(1982, Month.APRIL,
17, 14, 11);
ZonedDateTime jakartaTime = ZonedDateTime.of(localDateTime,
ZoneId.of("Asia/Jakarta"));
```

Daylight Saving Time Begins

- In the summer
  - In the case of a gap, when clocks jump forward, there is no valid offset.
  - Local date-time is adjusted to be later by the length of the gap
  - For a typical one hour daylight savings change, the local date-time will be moved one hour later into the offset typically corresponding to "summer"

Daylight Saving Time Exemplified

```
LocalDateTime date = LocalDateTime.of(2012, 11, 12, 13, 11, 12);
date.atZone(ZoneId.of("America/Los_Angeles")) //2012-11-
12T13:11:12-08:00[America/Los_Angeles]

LocalDateTime daylightSavingTime = LocalDateTime.of(2014, 3, 9, 2,
0, 0, 0);
daylightSavingTime.atZone(ZoneId.of("America/Denver")); //2014-03-
09T03:00-06:00[America/Denver]

LocalDateTime daylightSavingTime2 = LocalDateTime.of(2014, 3, 9, 2,
30, 0, 0);
daylightSavingTime2.atZone(ZoneId.of("America/New_York")); //2014-
03-09T03:30-04:00[America/New_York]

LocalDateTime daylightSavingTime3 = LocalDateTime.of(2014, 3, 9, 2,
0, 0, 0);
daylightSavingTime3.atZone(ZoneId.of("America/Phoenix")); //2014-
03-09T03:59:59.999999999-05:00[America/Chicago]

LocalDateTime daylightSavingTime4 = LocalDateTime.of(2014, 3, 9, 2,
59, 59, 999999999);
daylightSavingTime4.atZone(ZoneId.of("America/Chicago")); //2014-
03-09T02:00-07:00[America/Phoenix]
```

Daylight Saving Time Ends

- In the winter
  - In the case of an overlap, when clocks are set back, there are two valid offsets.
  - This method uses the earlier offset typically corresponding to "summer".

Standard Time Exemplified

```
LocalDateTime date2 = LocalDateTime.of(2012, 11, 12, 13, 11, 12);
date2.atZone(ZoneId.of("America/Los_Angeles"))); //2012-11-
12T13:11:12-08:00[America/Los_Angeles]

LocalDateTime standardTime = LocalDateTime.of(2014, 11, 2, 2, 0, 0,
0);
standardTime.atZone(ZoneId.of("America/Denver")); //2014-11-
02T02:00-07:00[America/Denver]

LocalDateTime standardTime2 = LocalDateTime.of(2014, 11, 2, 2, 30,
0, 0);
standardTime2.atZone(ZoneId.of("America/New_York")); //2014-11-
02T02:30-05:00[America/New_York]

LocalDateTime standardTime3 = LocalDateTime.of(2014, 11, 2, 2, 0,
0, 0);
standardTime3.atZone(ZoneId.of("America/Phoenix")); //2014-11-
02T02:00-07:00[America/Phoenix]

LocalDateTime standardTime4 = LocalDateTime.of(2014, 11, 2, 2, 59,
59, 999999999);
standardTime4.atZone(ZoneId.of("America/Chicago")); //2014-11-
02T02:59:59.999999999-06:00[America/Chicago]
```

Shifting Time

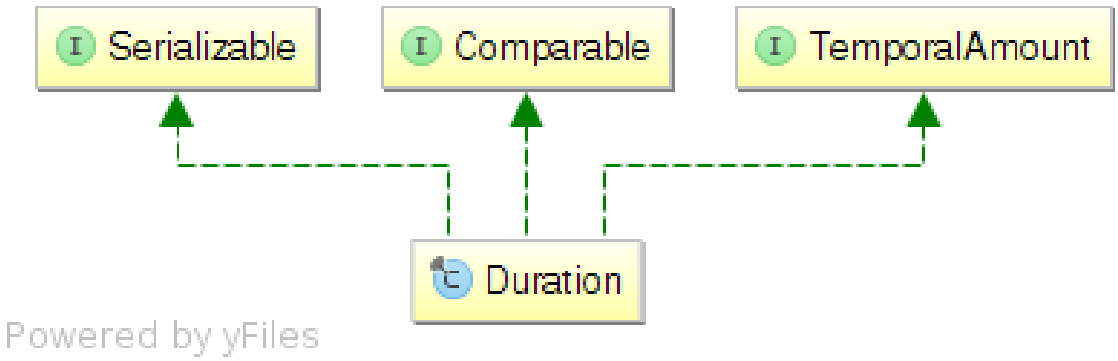
Durations and Periods

- To model a span of time (e.g. 10 days) you have two choices
  - `Duration` - a span of time in seconds and nanoseconds
  - `Period` -a span of time in years, months and days
- Both implement `TemporalAmount`

More about `Duration`

- Spans only seconds and nanoseconds
- Meant to adjust `LocalTime` (assumes no dates are involved)
- `static` method calls include construction for:
  - days
  - hours
  - milliseconds
  - nanoseconds
- Can have a side effect depending on which API calls you make

UML Diagram of `Duration`



`Duration` Exemplified

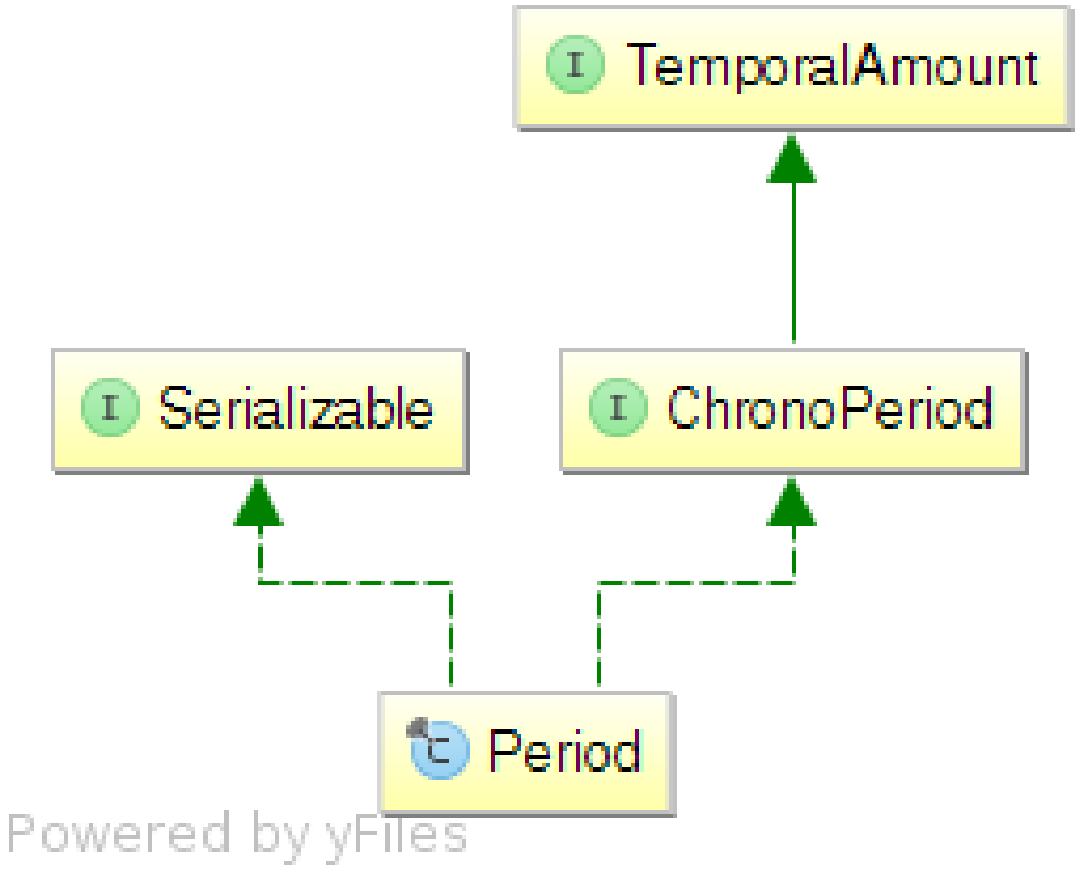
```
Duration duration =  Duration.ofDays(33); //seconds or nanos
Duration duration1 = Duration.ofHours(33); //seconds or nanos
Duration duration2 = Duration.ofMillis(33); //seconds or nanos
Duration duration3 = Duration.ofMinutes(33); //seconds or nanos
Duration duration4 = Duration.ofNanos(33); //seconds or nanos
Duration duration5 = Duration.ofSeconds(33); //seconds or nanos
Duration duration6 = Duration.between(LocalDate.of(2012, 11, 11),
LocalDate.of(2013, 1, 1));
```

More about `Period`

- Spans years, months, weeks and days
- Meant to adjust `LocalDate` (assumes no times are involved)
- `static` method calls include construction for:
  - days
  - months
  - weeks
  - years
- Can also have a side effect depending on which API call you make

---

UML Diagram of `Period`



`Period` Exemplified

```
Period p = Period.ofDays(30);
Period p1 = Period.ofMonths(12);
Period p2 = Period.ofWeeks(11);
Period p3 = Period.ofYears(50);
```

Shifting Dates and Time

- Any class that derives from `Temporal` has the ability to add or remove any time using methods:
  - `plus`
  - `minus`
- “Changing” any one implementation of a `Temporal` will provide a copy!

Shifting `LocalDate`

- A shift of `LocalDate` can be done with:
  - a `TemporalAmount` (`Period`)
  - a `long` with `TemporalUnit` (`ChronoUnit`)

```
LocalDate localDate = LocalDate.of(2012, 11, 23);
localDate.plus(3, ChronoUnit.DAYS); //2012-11-26
localDate.plus(Period.ofDays(3)); //2012-11-26
try {
   localDate.plus(Duration.ofDays(3));  //2012-11-26
} catch (UnsupportedTemporalTypeException e) {
   e.printStackTrace();
}
```

Shifting `LocalTime`

- A shift of `LocalTime` can be done with:
  - a `TemporalAmount` (`Duration`)
  - a `long` with `TemporalUnit` (`ChronoUnit`)

```
LocalTime localTime = LocalTime.of(11, 20, 50);
localTime.plus(3, ChronoUnit.HOURS); //14:20:50
localTime.plus(Duration.ofDays(3)); //11:20:50
try {
  localTime.plus(Period.ofDays(3));
} catch (UnsupportedTemporalTypeException e) {
  e.printStackTrace();
}
```

Temporal Adjusters

- New construct
- `interface` that can be implemented to specialize a time shift
- Use Case - An object that shifts time based on external factors

```
@FunctionalInterface
public interface TemporalAdjuster {
   Temporal adjustInto(Temporal temporal);
}
```

Overly Simplified Temporal Adjuster

```
TemporalAdjuster fourMinutesFromNow = new TemporalAdjuster() {
   @Override
   public Temporal adjustInto(Temporal temporal) {
       return temporal.plus(4, ChronoUnit.MINUTES);
   }
};

LocalTime localTime = LocalTime.of(12, 0, 0);
localTime.with(fourMinutesFromNow)); //12:04
```

But, wait there's more!

Remember this?

```
@FunctionalInterface
public interface TemporalAdjuster {
      Temporal adjustInto(Temporal temporal);
}
```

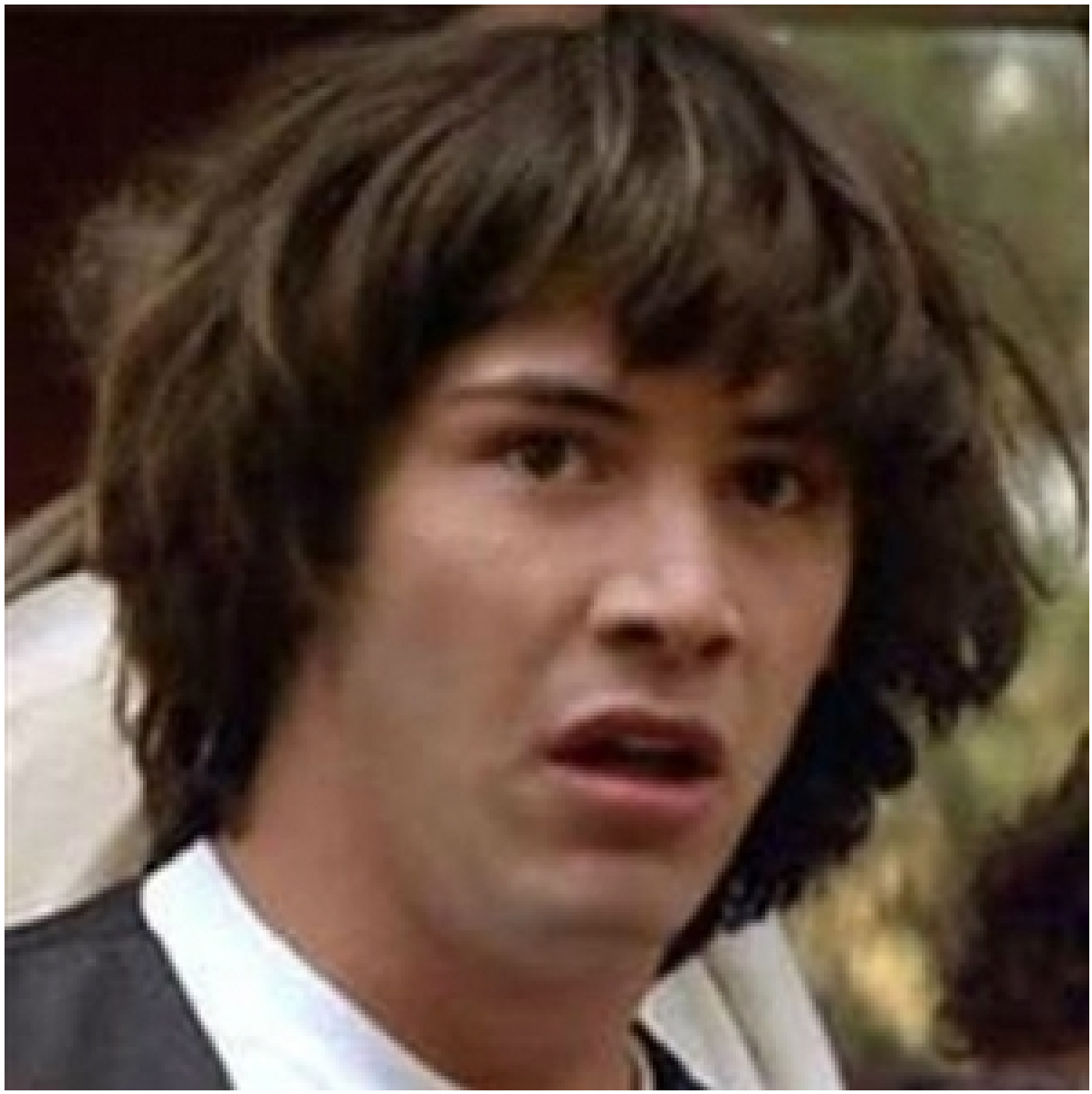
That's a Java 8 Lambda! Therefore `fourMinutesFromNow` can now be:

```
TemporalAdjuster fourMinutesFromNow = temporal -> temporal.plus(4,
ChronoUnit.MINUTES);
LocalTime localTime = LocalTime.of(12, 0, 0);
localTime.with(fourMinutesFromNow)); //12:04
```

Refactoring and inlining

```
LocalTime.of(12, 0, 0).with(temporal -> temporal.plus(4,
ChronoUnit.MINUTES));
```

Simple, isn't it?



Parsing and Formatting

- Converting dates and times from a `String` is always important
- `java.time.format.DateFormatter`
- Immutable and Threadsafe

Formatting `LocalDate`

```
DateTimeFormatter dateFormatter =
DateTimeFormatter.ofLocalizedDate(FormatStyle.MEDIUM);

dateFormatter.format(LocalDate.now()); // Jan. 19, 2014
```

Formatting `LocalTime`

```
DateTimeFormatter timeFormatter =
   DateTimeFormatter.ofLocalizedTime(FormatStyle.MEDIUM);

timeFormatter.format(LocalTime.now())); //3:01:48 PM
```

Formatting `LocalDateTime`

```
DateTimeFormatter dateTimeFormatter =
   DateTimeFormatter.ofLocalizedDateTime(FormatStyle.MEDIUM,
FormatStyle.SHORT);

dateTimeFormatter.format(LocalDateTime.now())); // Jan. 19, 2014
3:01 PM
```

Formatting Customized Patterns

```
DateTimeFormatter obscurePattern =
   DateTimeFormatter.ofPattern("MMMM dd, yyyy '(In Time Zone:
'VV')'");
ZonedDateTime zonedNow = ZonedDateTime.now();

obscurePattern.format(zonedNow); //January 19, 2014 (In Time Zone:
America/Denver)
```

Formatting with Localization

- Localization using `java.util.Locale` is available for:
  - `ofLocalizedDate`
  - `ofLocalizedTime`
  - `ofLocalizedDateTime`

```
ZonedDateTime zonedDateTime =
ZonedDateTime.now(ZoneId.of("Europe/Paris"));

DateTimeFormatter longDateTimeFormatter =
DateTimeFormatter.ofLocalizedDateTime(FormatStyle.FULL,
FormatStyle.FULL).withLocale(Locale.FRENCH);
longDateTimeFormatter.getLocale(); //fr
longDateTimeFormatter.format(zonedDateTime); //samedi 19 janvier
2014 00 h 00 CET
```

Shifting Time Zones

```
LocalDateTime localDateTime = LocalDateTime.of(1982, Month.APRIL,
17, 14, 11);
ZonedDateTime jakartaTime = ZonedDateTime.of(localDateTime,
ZoneId.of("Asia/Jakarta"));
jakartaTime.withZoneSameInstant(ZoneId.of("America/Los_Angeles")));
//1982-04-16T23:11-08:00[America/Los_Angeles]
jakartaTime.withZoneSameLocal(ZoneId.of("America/New_York")));
//1982-04-17T14:11-05:00[America/New_York]
```

Temporal Querying

- Process of asking information about a `TemporalAccessor`
  - `LocalDate`
  - `LocalTime`

- `LocalDateTime`
- `ZonedDateTime`

```
@FunctionalInterface
public interface TemporalQuery<R> {
    R queryFrom(TemporalAccessor temporal);
}
```

A Festive Example

```
TemporalQuery<Integer> daysBeforeChristmas = new
TemporalQuery<Integer>() {
    public int daysTilChristmas (int acc, Temporal temporal) {
        int month = temporal.get(ChronoField.MONTH_OF_YEAR);
        int day = temporal.get(ChronoField.DAY_OF_MONTH);
        int max = Month.of(month).maxLength();
        if (month == 12 && day <= 25) return acc + (25 - day);
        return daysTilChristmas(acc + (max - day + 1),
           temporal.with(TemporalAdjusters.firstDayOfNextMonth()));
    }


    @Override
    public Integer queryFrom(TemporalAccessor temporal) {
        if (!(temporal instanceof Temporal))
            throw new RuntimeException("Temporal accessor must be
of type Temporal");
        return daysTilChristmas(0, (Temporal) temporal);
    }
};

LocalDate.of(2013, 12, 26).query(daysBeforeChristmas); //364
LocalDate.of(2013, 12, 23).query(daysBeforeChristmas); //2
LocalDate.of(2013, 12, 25).query(daysBeforeChristmas); //0
ZonedDateTime.of(2013, 12, 1, 11, 0, 13, 938282,
  ZoneId.of("America/Los_Angeles"))
  .query(daysBeforeChristmas)); //24
```

Let's come back to parsing

- We discussed `format` but not `parse`
- There is more to it

Simple Parsing

LOLWUT?

```
DateTimeFormatter dateFormatter =
DateTimeFormatter.ofLocalizedDate(FormatStyle.MEDIUM);
dateFormatter.parse("Jan 19, 2014")); // {}, ISO resolved to 2014-
01-19
```

- Parses to `java.time.format.Parsed` which is rather useless
- The more effective call is to `parse(CharSequence, TemporalQuery)`

First Attempt

```
TemporalQuery<LocalDate> localDateTemporalQuery = new
TemporalQuery<LocalDate>() {
   @Override
   public LocalDate queryFrom(TemporalAccessor temporal) {
       return LocalDate.from(temporal);
   }
};

dateFormatter.parse("Jan 19, 2014", localDateTemporalQuery);
//2014-01-19
```

Second Attempt

```
dateFormatter.parse("Jan 19, 2014", temporal ->
LocalDate.from(temporal)); //2014-01-19
```

About Java 8 Instance Method References

Given

```
@FunctionalInterface
public interface Predicate<T> {
   public boolean test(T t);
}
```

And

```
static class IntPredicates {
public static boolean isOdd(Integer n) { return n % 2 != 0; }
public static boolean isEven(Integer n) { return n % 2 == 0; }
public static boolean isPositive(Integer n) { return n >= 0; }
}
```

About Java 8 Instance Method References (Continued)

Using the `Predicate`

```
Predicate<Integer> isOdd = n -> IntPredicates.isOdd(n);
Predicate<Integer> isEven = n -> IntPredicates.isEven(n);
```

We can always refer to those as:

```
Predicate<Integer> isOdd = IntPredicates::isOdd;
Predicate<Integer> isEven = IntPredicate::isEven;
```

source: http://java.dzone.com/articles/java-lambda-expressions-vs (http://java.dzone.com/articles/java-lambda-expressions-vs)

What does that mean to us?

Inside of LocalDate.java

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

Therefore, our last attempt

```
dateFormatter.parse("Jan 19, 2014", LocalDate::from); // Jan 19,
2014
```

Interoperabilility with Legacy Code

- `calendar.toInstant()` - converts the Calendar object to an Instant.
- `gregorianCalendar.toZonedDateTime()` - converts a GregorianCalendar instance to a ZonedDateTime.

- `gregorianCalendar.from(ZonedDateTime)` - creates a GregorianCalendar object using the default locale from a ZonedDateTime instance.
- `date.from(Instant)` - creates a Date object from an Instant.
- `date.toInstant()` - converts a Date object to an Instant.
- `timeZone.toZoneId()` - converts a TimeZone object to a ZoneId.

```
GregorianCalendar gregorianCalendar = new GregorianCalendar();
gregorianCalendar.toZonedDateTime();
```

Conclusion

- Lot to play with and enjoy
- Immutablity rules!
- The only learning curve is knowing the difference between `LocalDate`, `LocalTime`; `Period`, and `Duration`
- Tight integration with Java 8 lambdas
- Nanosecond Resolution, not millseconds
- Should be considered with strong consideration

Questions?

Thank You

- Email: dhinojosa@evolutionnext.com (mailto:dhinojosa@evolutionnext.com)
- Twitter: @dhinojosa
- Google Plus: gplus.to/dhinojosa
- Linked In: www.linkedin.com/in/dhevolutionnext

Last updated 2014-04-23 10:30:47 MDT