Enough history. Let's look at dates and times in Java!

Early Java (before version 8)

- Had the following classes for dates:
  - Date
  - Calendar
  - SimpleDateFormat
- Problems:
  - Not thread safe
  - The classes didn't correspond to the domain very well

The Date class was particularly bad

- No time zones
- Months are 0-indexed (this made it far too easy to introduce off-by-1 errors)
- Uses the local system timezone in too many places
- Doesn’t correspond well to the java.sql.Date class
- It’s not really a “date” - it’s more of an “instant”
- Confusing methods: getDate() returns the day of the month; getDay() returns the day of the week. And it had some rough edges - you could specify February 1 as January 32 (!).

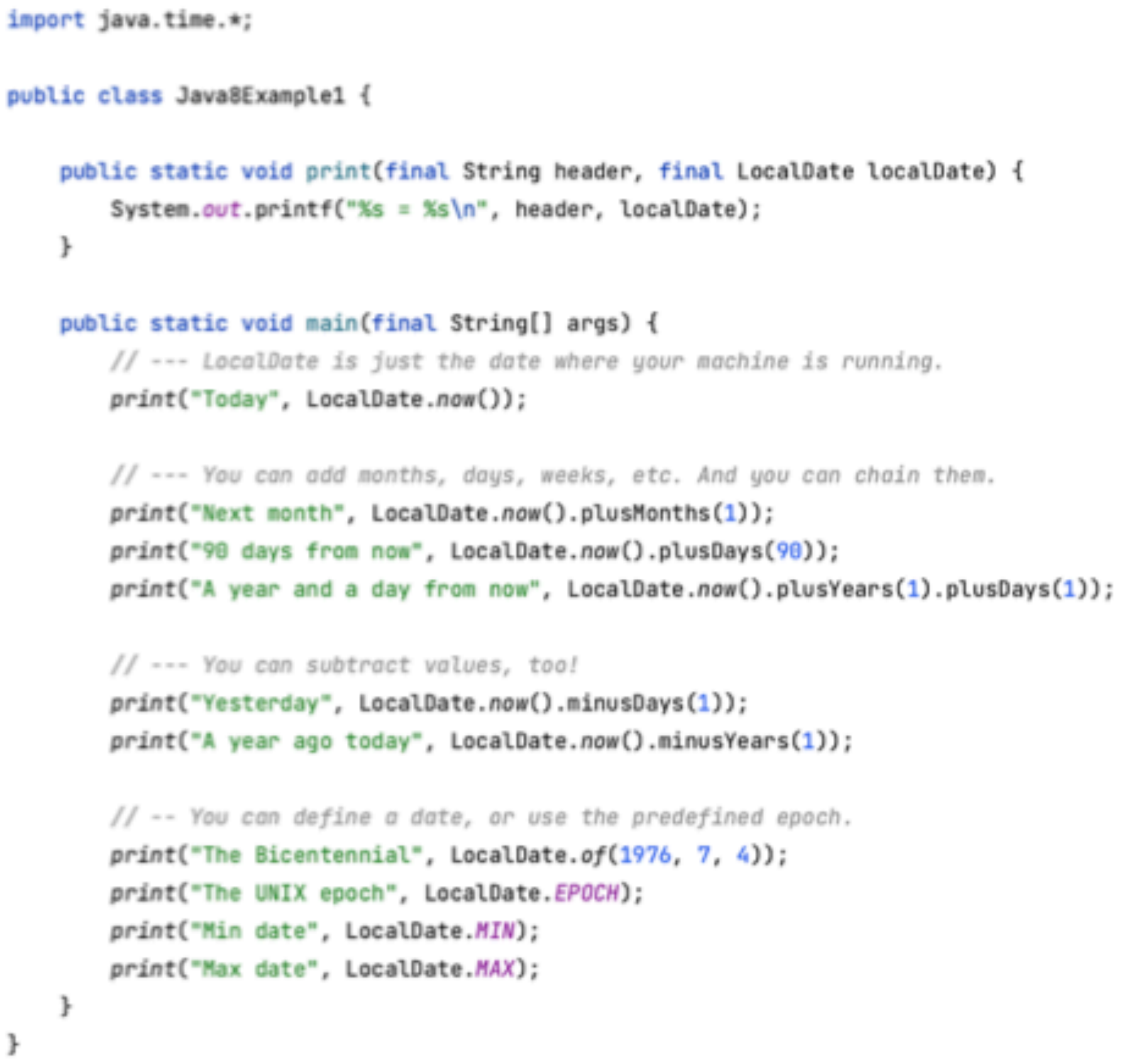
JodaTime

- Multiple calendar systems (Gregorian, Julian, etc.) with ISO 8601 as the default.
- Better API. Immutable classes, fluent API.
- Better performance.
- Less confusing to use.
- It is the de facto standard date and time library for Java versions earlier than 8.
- However, the designer (Stephen Colebourne) says it has some “design flaws”. It’s not “broken”, but it could have been designed better.

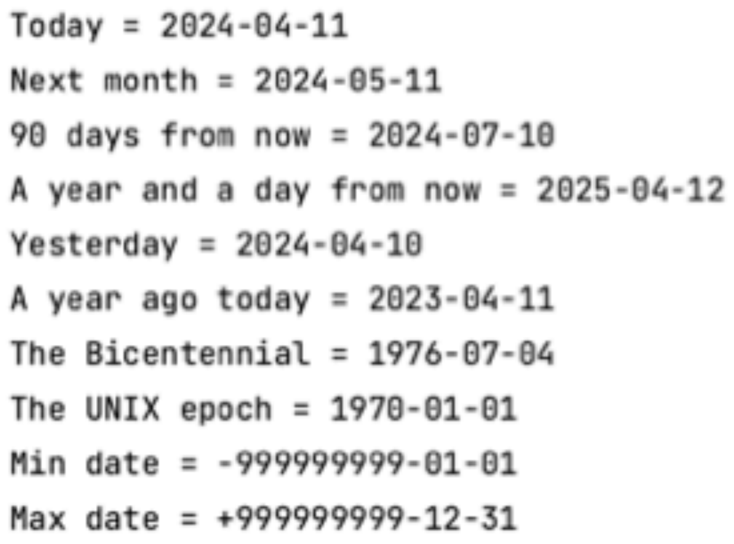
The Java 8 Date And Time APIs

- Stephen Colebourne (Joda Time) was again a primary author
- He decided to address the design flaws in Joda Time by implementing a new API in Java itself.
- Java 8 is inspired by Joda Time, but with a better design.
- The APIs are in the java.time package.

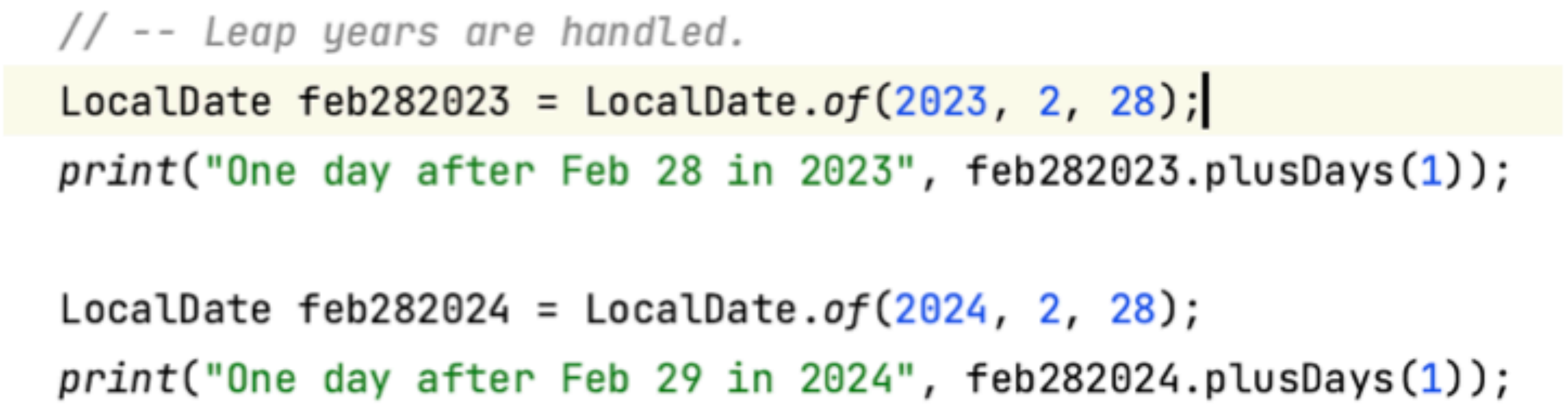
LocalDate



```java
import java.time.*;

public class Java8Example1 {

    public static void print(final String header, final LocalDate localDate) {
        System.out.printf("%s = %s\n", header, localDate);
    }

    public static void main(final String[] args) {
        // --- LocalDate is just the date where your machine is running.
        print("Today", LocalDate.now());

        // --- You can add months, days, weeks, etc. And you can chain them.
        print("Next month", LocalDate.now().plusMonths(1));
        print("90 days from now", LocalDate.now().plusDays(90));
        print("A year and a day from now", LocalDate.now().plusYears(1).plusDays(1));

        // --- You can subtract values, too!
        print("Yesterday", LocalDate.now().minusDays(1));
        print("A year ago today", LocalDate.now().minusYears(1));

        // -- You can define a date, or use the predefined epoch.
        print("The Bicentennial", LocalDate.of(1976, 7, 4));
        print("The UNIX epoch", LocalDate.EPOCH);
        print("Min date", LocalDate.MIN);
        print("Max date", LocalDate.MAX);
    }
}
```



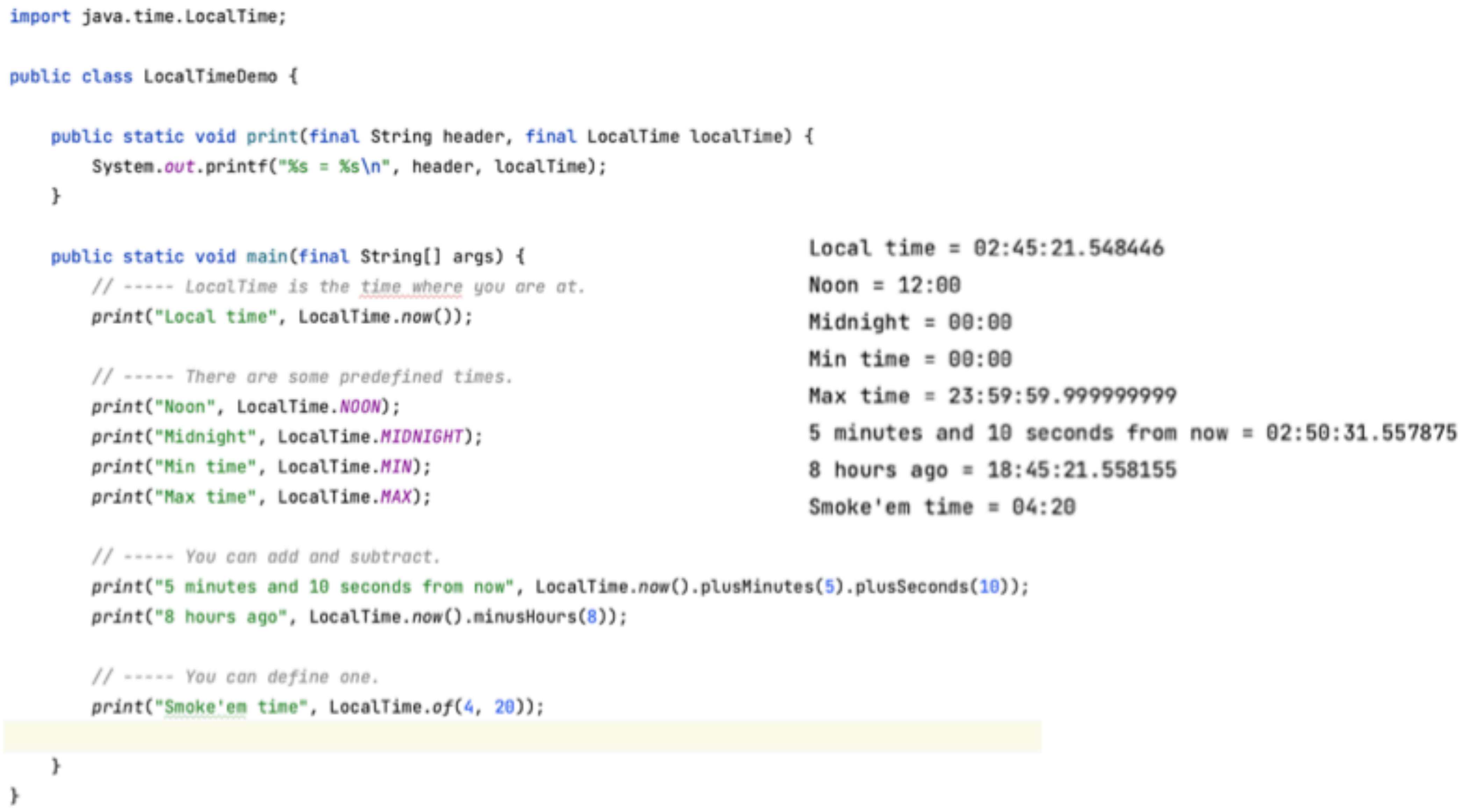
```
Today = 2024-04-11
Next month = 2024-05-11
90 days from now = 2024-07-10
A year and a day from now = 2025-04-12
Yesterday = 2024-04-10
A year ago today = 2023-04-11
The Bicentennial = 1976-07-04
The UNIX epoch = 1970-01-01
Min date = -999999999-01-01
Max date = +999999999-12-31
```



```java
// -- Leap years are handled.
LocalDate feb282023 = LocalDate.of(2023, 2, 28);
print("One day after Feb 28 in 2023", feb282023.plusDays(1));

LocalDate feb282024 = LocalDate.of(2024, 2, 28);
print("One day after Feb 29 in 2024", feb282024.plusDays(1));
```

```
One day after Feb 28 in 2023 = 2023-03-01
One day after Feb 29 in 2024 = 2024-02-29
```

LocalTime



```java
import java.time.LocalTime;

public class LocalTimeDemo {

    public static void print(final String header, final LocalTime localTime) {
        System.out.printf("%s = %s\n", header, localTime);
    }

    public static void main(final String[] args) {
        // ----- LocalTime is the time where you are at.
        print("Local time", LocalTime.now());

        // ----- There are some predefined times.
        print("Noon", LocalTime.NOON);
        print("Midnight", LocalTime.MIDNIGHT);
        print("Min time", LocalTime.MIN);
        print("Max time", LocalTime.MAX);

        // ----- You can add and subtract.
        print("5 minutes and 10 seconds from now", LocalTime.now().plusMinutes(5).plusSeconds(10));
        print("8 hours ago", LocalTime.now().minusHours(8));

        // ----- You can define one.
        print("Smoke'em time", LocalTime.of(4, 20));

    }
}
```

```
Local time = 02:45:21.548446
Noon = 12:00
Midnight = 00:00
Min time = 00:00
Max time = 23:59:59.999999999
5 minutes and 10 seconds from now = 02:50:31.557875
8 hours ago = 18:45:21.558155
Smoke'em time = 04:20
```

LocalDateTime

```java
// ----- The local date and time.
print("Now", LocalDateTime.now());

// ----- And you can manipulate it.
print("Yesterday", LocalDateTime.now().minusDays(1));
print("A week from now", LocalDateTime.now().plusWeeks(1));
print("3:27 from now", LocalDateTime.now().plusMinutes(3).plusSeconds(27));
```

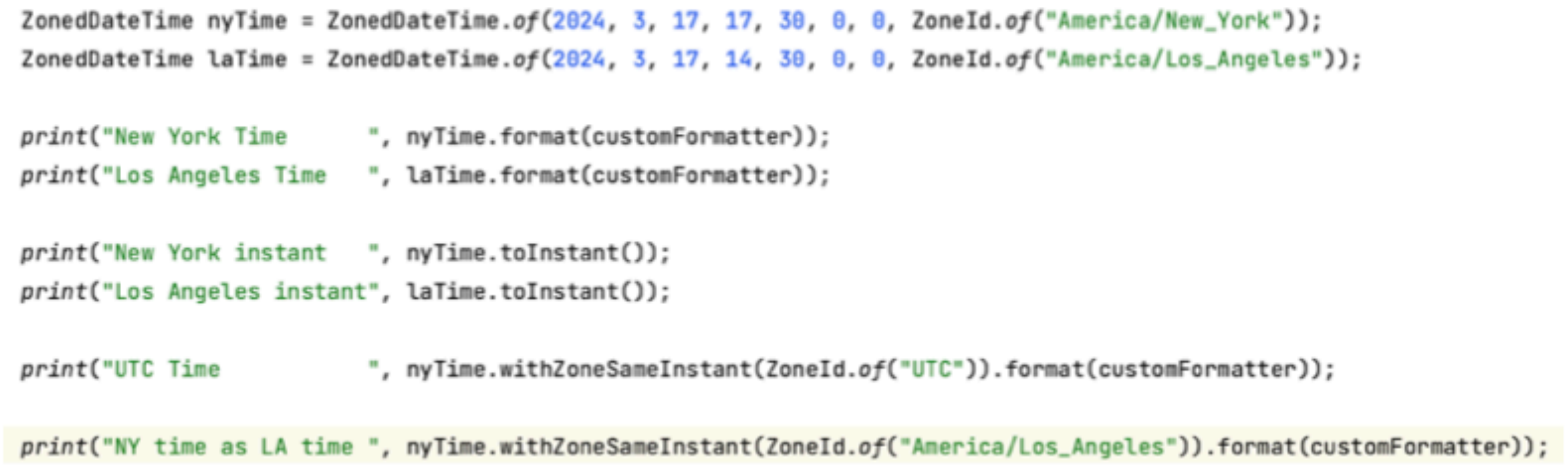
```
Now = 2024-04-11T02:54:33.565676
Yesterday = 2024-04-10T02:54:33.571665
A week from now = 2024-04-18T02:54:33.571808
3:27 from now = 2024-04-11T02:58:00.571928
```

ZonedDateTime

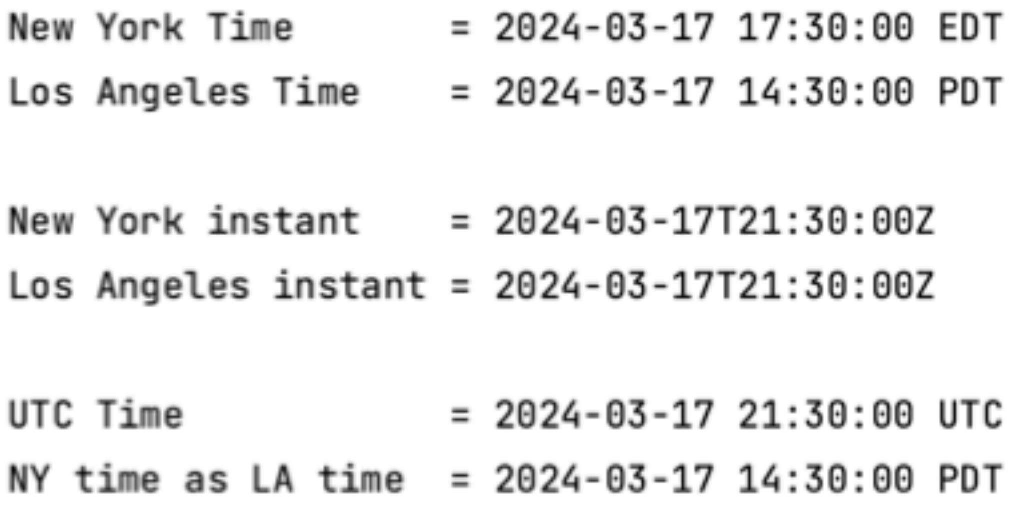
```java
// ----- DateTimeFormatter lets us format a ZonedDateTime
DateTimeFormatter customFormatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss z");

// ----- ZonedDateTime is like LocalDateTime with a time zone
ZonedDateTime now = ZonedDateTime.now();
print("now                ", now.format(customFormatter));
print("ISO_ZONED_DATE_TIME", now.format(DateTimeFormatter.ISO_ZONED_DATE_TIME));
print("ISO_DATE_TIME      ", now.format(DateTimeFormatter.ISO_DATE_TIME));
print("ISO_WEEK_DATE      ", now.format(DateTimeFormatter.ISO_WEEK_DATE));
print("ISO_LOCAL_DATE_TIME", now.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));

print("default time zone", now.getZone());
print("zone offset", now.getOffset());
```

```
now                 = 2024-04-11 03:31:10 EDT
ISO_ZONED_DATE_TIME = 2024-04-11T03:31:10.953531-04:00[America/New_York]
ISO_DATE_TIME       = 2024-04-11T03:31:10.953531-04:00[America/New_York]
ISO_WEEK_DATE       = 2024-W15-4-04:00
ISO_LOCAL_DATE_TIME = 2024-04-11T03:31:10.953531
default time zone = America/New_York
zone offset = -04:00
```

Instants



```java
ZonedDateTime nyTime = ZonedDateTime.of(2024, 3, 17, 17, 30, 0, 0, ZoneId.of("America/New_York"));
ZonedDateTime laTime = ZonedDateTime.of(2024, 3, 17, 14, 30, 0, 0, ZoneId.of("America/Los_Angeles"));

print("New York Time      ", nyTime.format(customFormatter));
print("Los Angeles Time   ", laTime.format(customFormatter));

print("New York instant   ", nyTime.toInstant());
print("Los Angeles instant", laTime.toInstant());

print("UTC Time           ", nyTime.withZoneSameInstant(ZoneId.of("UTC")).format(customFormatter));

print("NY time as LA time ", nyTime.withZoneSameInstant(ZoneId.of("America/Los_Angeles")).format(customFormatter));
```

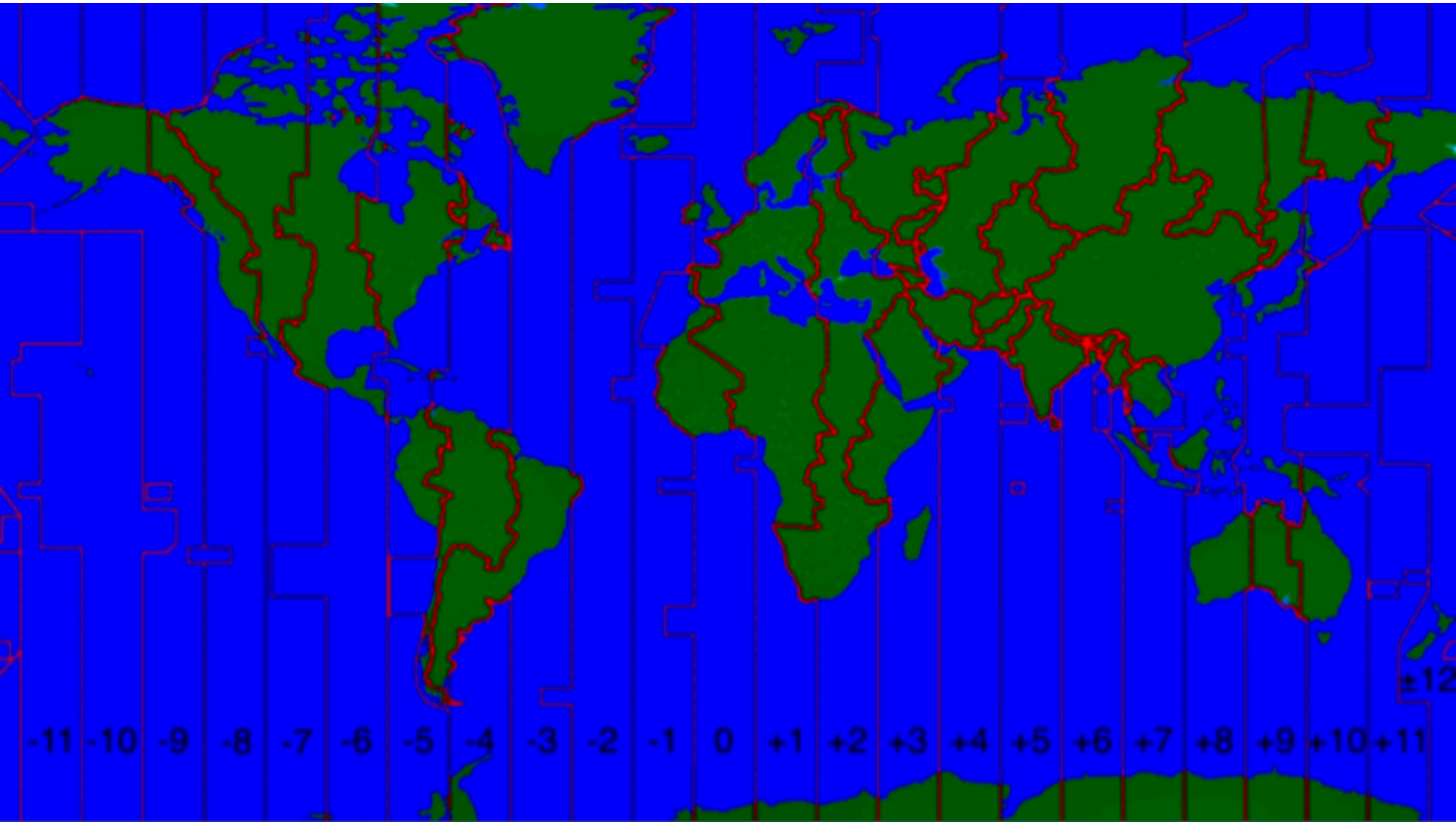


```
New York Time       = 2024-03-17 17:30:00 EDT
Los Angeles Time    = 2024-03-17 14:30:00 PDT

New York instant    = 2024-03-17T21:30:00Z
Los Angeles instant = 2024-03-17T21:30:00Z

UTC Time            = 2024-03-17 21:30:00 UTC
NY time as LA time  = 2024-03-17 14:30:00 PDT
```

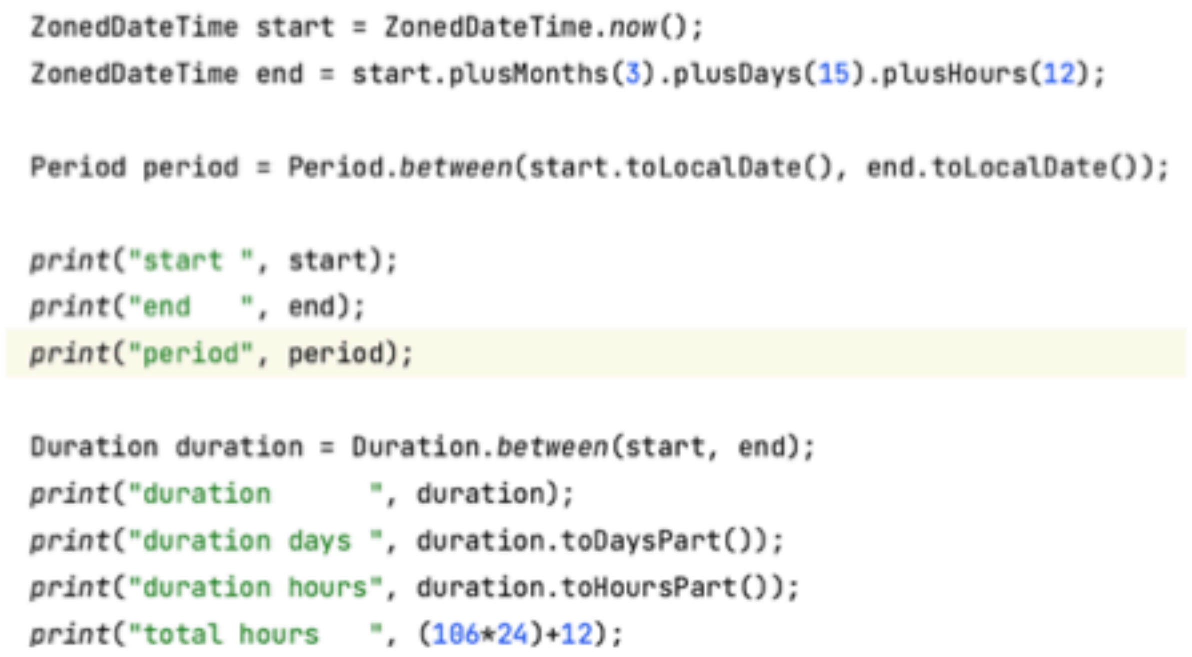
OffsetDateTime

```java
// A ZonedDateTime has a Date, a Time, a ZoneId, and a Zone offset.
ZonedDateTime now = ZonedDateTime.now();

// OffsetDateTime doesn't have a ZoneId.
// So, it can't handle Daylight Savings Time, etc.
OffsetDateTime nowAsOffset = now.toOffsetDateTime();

print("now         ", now);
print("now's offset", now.getOffset());
print("now's ZoneId", now.getZone());
print("nowAsOffset ", nowAsOffset);
```

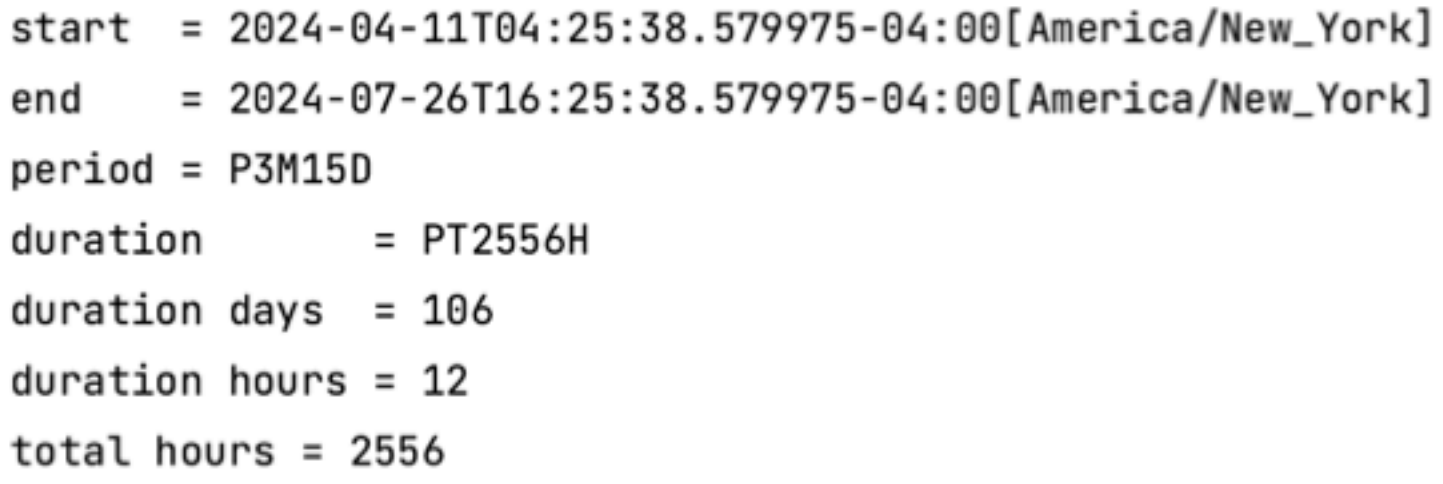
```
now          = 2024-04-11T04:14:42.160005-04:00[America/New_York]
now's offset = -04:00
now's ZoneId = America/New_York

nowAsOffset  = 2024-04-11T04:14:42.160005-04:00
```


±12
-11
-10
-9
-8
-7
-6
-5
-4
-3
-2
-1
0
+1
+2
+3
+4
+5
+6
+7
+8
+9
+10
+11

Periods and Durations



```java
ZonedDateTime start = ZonedDateTime.now();
ZonedDateTime end = start.plusMonths(3).plusDays(15).plusHours(12);

Period period = Period.between(start.toLocalDate(), end.toLocalDate());

print("start ", start);
print("end   ", end);
print("period", period);

Duration duration = Duration.between(start, end);
print("duration      ", duration);
print("duration days ", duration.toDaysPart());
print("duration hours", duration.toHoursPart());
print("total hours   ", (106*24)+12);
```



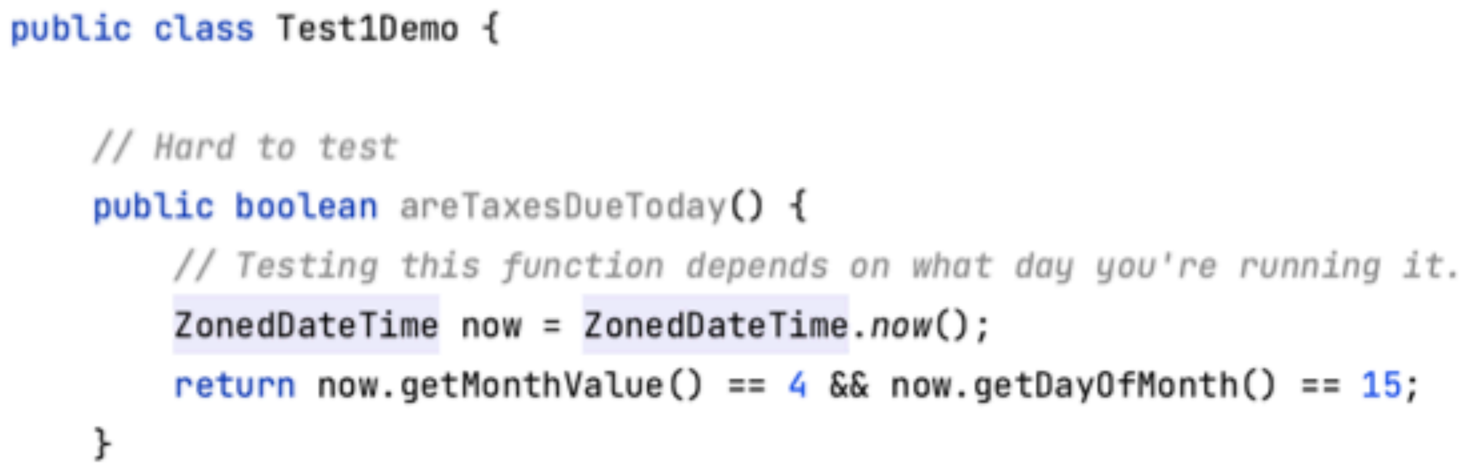
```
start  = 2024-04-11T04:25:38.579975-04:00[America/New_York]
end    = 2024-07-26T16:25:38.579975-04:00[America/New_York]
period = P3M15D
duration       = PT2556H
duration days  = 106
duration hours = 12
total hours = 2556
```

Summary

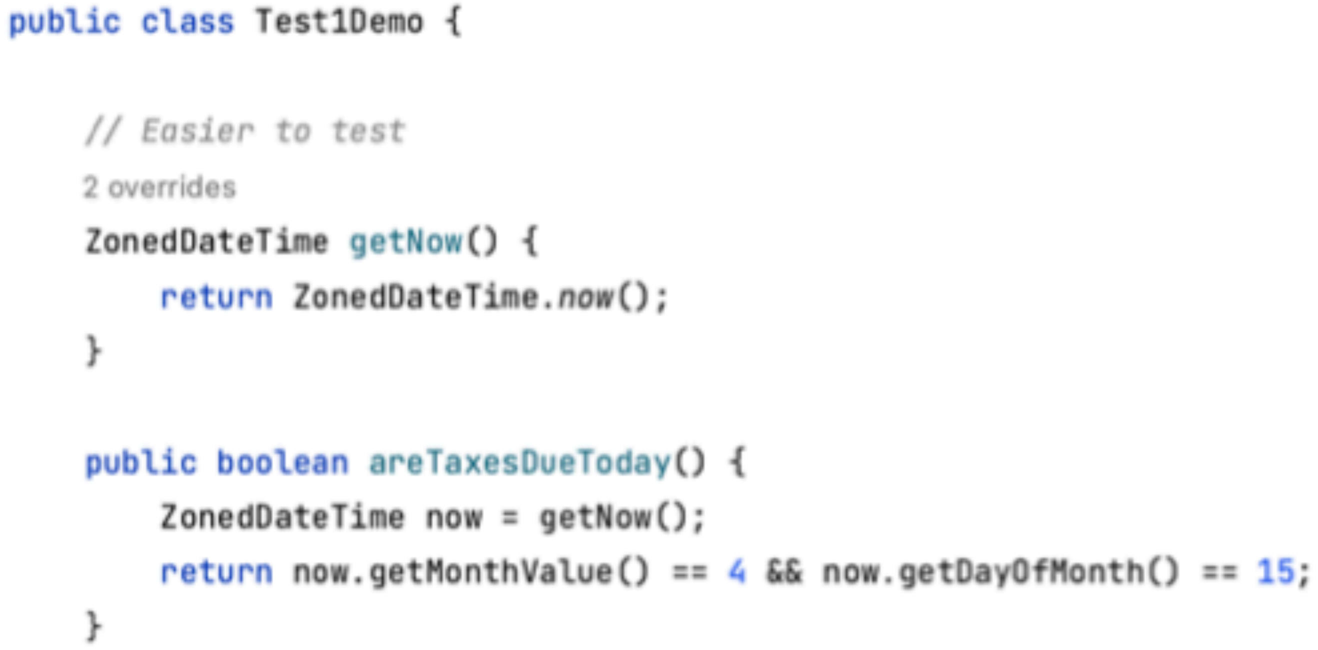
- Use ZonedDateTime. It has a date, time, ZoneId, and ZoneOffset.
  - The ZoneId lets it deal with Daylight Savings Time, etc.
- You can use DateFormatter to format it to your preferred format.
- Period and Duration are useful for calculating the difference between two dates.
- ZonedDateTime's .withZoneSameInstant(...zoneID...) is useful for seeing what time it is right now in a different ZoneID. For example, if it's 9:03pm in Atlanta, Georgia, what time is it in London, England right now? Or Sydney, Australia?

Unit test considerations

- Use ZonedDateTime.of(....) to construct specific date/times instead of using ZonedDateTime.now().
- You can write a function that gets the current ZonedDateTime so that it's easy to override or mock in a unit test.
- Or, you can write a TimeFactory to inject in your code that will let you mock dates and times for testing.



```java
public class Test1Demo {

    // Hard to test
    public boolean areTaxesDueToday() {
        // Testing this function depends on what day you're running it.
        ZonedDateTime now = ZonedDateTime.now();
        return now.getMonthValue() == 4 && now.getDayOfMonth() == 15;
    }
```

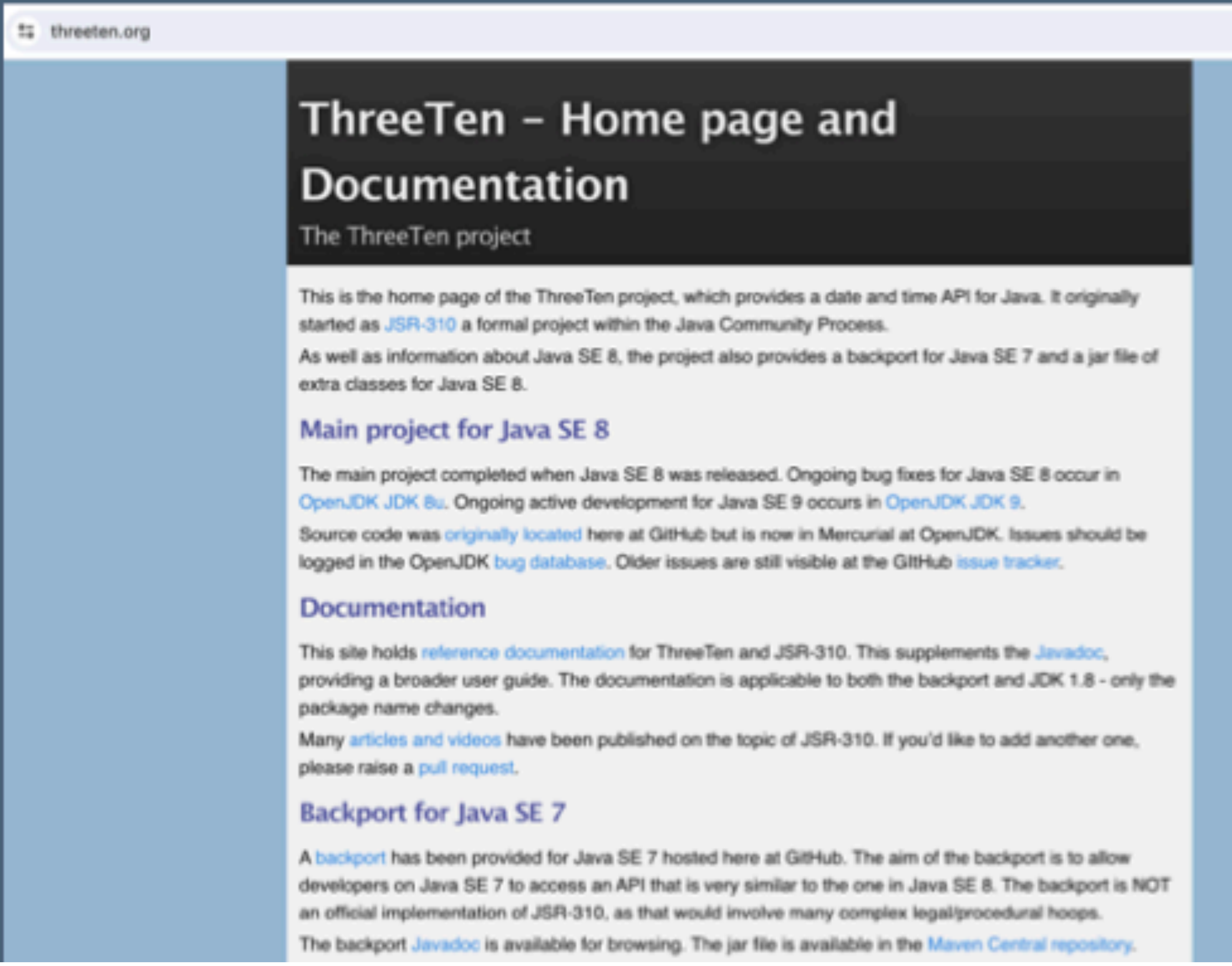


```java
public class Test1Demo {

    // Easier to test
    2 overrides
    ZonedDateTime getNow() {
        return ZonedDateTime.now();
    }

    public boolean areTaxesDueToday() {
        ZonedDateTime now = getNow();
        return now.getMonthValue() == 4 && now.getDayOfMonth() == 15;
    }
    public static void main(final String[] args) {
        Test1Demo demo1 = new Test1Demo() {
            ZonedDateTime getNow() {
                return ZonedDateTime.of(2024, 03, 17, 0, 0, 0, 0, ZoneId.of("America/New_York"));
            }
        };

        Test1Demo demo2 = new Test1Demo() {
            ZonedDateTime getNow() {
                return ZonedDateTime.of(2024, 04, 15, 0, 0, 0, 0, ZoneId.of("America/New_York"));
            }
        };

        System.out.println("Are taxes due today? " + demo1.areTaxesDueToday());
        System.out.println("Are taxes due today? " + demo2.areTaxesDueToday());

    }
}
```

```
Are taxes due today? false
Are taxes due today? true
```

ThreeTen

ThreeTen - An incubator



threeten.org

ThreeTen – Home page and Documentation

The ThreeTen project

This is the home page of the ThreeTen project, which provides a date and time API for Java. It originally started as JSR-310 a formal project within the Java Community Process.

As well as information about Java SE 8, the project also provides a backport for Java SE 7 and a jar file of extra classes for Java SE 8.

Main project for Java SE 8

The main project completed when Java SE 8 was released. Ongoing bug fixes for Java SE 8 occur in OpenJDK JDK 8u. Ongoing active development for Java SE 9 occurs in OpenJDK JDK 9.

Source code was originally located here at GitHub but is now in Mercurial at OpenJDK. Issues should be logged in the OpenJDK bug database. Older issues are still visible at the GitHub issue tracker.

Documentation

This site holds reference documentation for ThreeTen and JSR-310. This supplements the Javadoc, providing a broader user guide. The documentation is applicable to both the backport and JDK 1.8 - only the package name changes.

Many articles and videos have been published on the topic of JSR-310. If you'd like to add another one, please raise a pull request.

Backport for Java SE 7

A backport has been provided for Java SE 7 hosted here at GitHub. The aim of the backport is to allow developers on Java SE 7 to access an API that is very similar to the one in Java SE 8. The backport is NOT an official implementation of JSR-310, as that would involve many complex legal/procedural hoops.

The backport Javadoc is available for browsing. The jar file is available in the Maven Central repository.

Daugherty

BUSINESS SOLUTIONS

kelly.morrison@daugherty.com
kellyivymorrison@gmail.com

Slides on GitHub

Kelly Morrison
Manager / Application Architect at Daugherty Business Solutions

LinkedIn