HowToDoInJava

Java 8 – Date and Time Examples

Last Updated: December 26, 2020 By: Lokesh Gupta Java 8 Java 8, Java Date Time, Lambda Expression

A big part of developer community has been complaining about **Date and Calendar classes**. Reasons were many such as hard to understand, hard to use and not flexible. Date class has even become obsolete and java docs suggest to use `Calendar` class instead of `Date` class. And on top of all, **Date comparison** is buggy and I have also faced such issue in past.

Moving forward, JAVA 8 (Lambda) is expected to release the new Date and Time APIs/classes (**JSR-310**), also called as **ThreeTen**, which will simply change the way you have been doing till date. This A key part of this is providing a new API that is dramatically easier to use and less error prone.

It will provide some highly demanded features such as:

- All the key public classes are immutable and thread-safe
- Defined terminology and behavior that other areas in computing can adopt

I wrote this post on 15th May 2013. Now today on 18th Mar 2014, java 8 is finally released and available for early access. I have re-validated and verified all the outputs in post examples. They work like charm as they did in May last year. Only change encountered was in `TemporalAdjuster.java`. Previously it was a class, now it is a `@FunctionalInterface`. So, I have corrected the related example and used the class "`TemporalAdjusters.java`".

Table of Contents

New classes to represent local date and timezone

The new classes intended to replace Date class are `LocalDate`, `LocalTime` and `LocalDateTime`.

LocalDate

The LocalDate class represents a date. There is no representation of a time or time-zone.

```
LocalDate localDate = LocalDate.now();
System.out.println(localDate.toString());                //2013-05-15
System.out.println(localDate.getDayOfWeek().toString()); //WEDNESDAY
System.out.println(localDate.getDayOfMonth());           //15
System.out.println(localDate.getDayOfYear());            //135
System.out.println(localDate.isLeapYear());              //false
System.out.println(localDate.plusDays(12).toString());   //2013-05-27
```

LocalTime

The LocalTime class represents a time. There is no representation of a date or time-zone.

```
//LocalTime localTime = LocalTime.now();     //toString() in format 09:57:59.744
LocalTime localTime = LocalTime.of(12, 20);
System.out.println(localTime.toString());    //12:20
System.out.println(localTime.getHour());     //12
System.out.println(localTime.getMinute());   //20
System.out.println(localTime.getSecond());   //0
System.out.println(localTime.MIDNIGHT);      //00:00
System.out.println(localTime.NOON);          //12:00
```

LocalDateTime

The LocalDateTime class represents a date-time. There is no representation of a time-zone.

```
LocalDateTime localDateTime = LocalDateTime.now();
System.out.println(localDateTime.toString());      //2013-05-15T10:01:14.911
System.out.println(localDateTime.getDayOfMonth()); //15
System.out.println(localDateTime.getHour());       //10
System.out.println(localDateTime.getNano());       //911000000
```

If you want to use the date functionality with zone information, then Lambda provide you extra 3 classes similar to above one i.e. `OffsetDate`, `OffsetTime` and `OffsetDateTime`. Timezone offset can be represented in "+05:30" or "Europe/Paris" formats. This is done via using another class i.e. `ZoneId`.

```
OffsetDateTime offsetDateTime = OffsetDateTime.now();
System.out.println(offsetDateTime.toString());              //2013-05-15T10:10:37.257+05:30

offsetDateTime = OffsetDateTime.now(ZoneId.of(&quot;+05:30&quot;));
System.out.println(offsetDateTime.toString());              //2013-05-15T10:10:37.258+05:30
```

```
offsetDateTime = OffsetDateTime.now(ZoneId.of(&quot;-06:30&quot;));
System.out.println(offsetDateTime.toString());              //2013-05-14T22:10:37.258-06:30

ZonedDateTime zonedDateTime =
        ZonedDateTime.now(ZoneId.of(&quot;Europe/Paris&quot;));
System.out.println(zonedDateTime.toString());           //2013-05-15T06:45:45.290+02:00[Europe/Paris]
```

New classes to represent timestamp and duration

Instant

For representing the specific timestamp ant any moment, the class needs to be used is Instant. The `Instant` class represents an instant in time to an accuracy of nanoseconds. Operations on an `Instant` include comparison to another `Instant` and adding or subtracting a duration.

```
Instant instant = Instant.now();
System.out.println(instant.toString());                                 //2013-05-15T05:20:08.145Z
System.out.println(instant.plus(Duration.ofMillis(5000)).toString());   //2013-05-15T05:20:13.145Z
System.out.println(instant.minus(Duration.ofMillis(5000)).toString());  //2013-05-15T05:20:03.145Z
System.out.println(instant.minusSeconds(10).toString());        //2013-05-15T05:19:58.145Z
```

Duration

Duration class is a whole new concept brought first time in java language. It represents the time difference between two time stamps.

```
Duration duration = Duration.ofMillis(5000);
System.out.println(duration.toString());     //PT5S

duration = Duration.ofSeconds(60);
System.out.println(duration.toString());     //PT1M

duration = Duration.ofMinutes(10);
System.out.println(duration.toString());     //PT10M

duration = Duration.ofHours(2);
System.out.println(duration.toString());     //PT2H

duration = Duration.between(Instant.now(), Instant.now().plus(Duration.ofMinutes(10)));
System.out.println(duration.toString());  //PT10M
```

`Duration` deals with small unit of time such as milliseconds, seconds, minutes and hour. They are more suitable for interacting with application code.

Period

To interact with human, you need to get bigger durations which are presented with Period class.

```
Period period = Period.ofDays(6);
System.out.println(period.toString());    //P6D

period = Period.ofMonths(6);
System.out.println(period.toString());    //P6M
```

```
period = Period.between(LocalDate.now(),
      LocalDate.now().plusDays(60));
System.out.println(period.toString());   //P1M29D
```

Added utility classes over existing enums

The current Java SE platform uses int constants for months, day-of-week and am-pm etc. Now a lot of extra utility classes have been added which work on top of these enums. I am taking an example such a class DayOfWeek. This class is a wrapper of day enums and can be used consistently with other classes also.

DayOfWeek

```
//day-of-week to represent, from 1 (Monday) to 7 (Sunday)
System.out.println(DayOfWeek.of(2));              //TUESDAY

DayOfWeek day = DayOfWeek.FRIDAY;
System.out.println(day.getValue());               //5

LocalDate localDate = LocalDate.now();
System.out.println(localDate.with(DayOfWeek.MONDAY));  //2013-05-13  i.e. when was monday in current week ?
```

Other such classes are `Month`, `MonthDay`, `Year`, `YearMonth` and many more.

Date adjusters

Date adjusters are another beautiful and useful addition in date handling tools. It easily solves the problems like : How do you *find last day of the month*? Or the *next working day*? Or a week on Tuesday?

Lets see in code.

```
LocalDate date = LocalDate.of(2013, Month.MAY, 15);           //Today

LocalDate endOfMonth = date.with(TemporalAdjusters.lastDayOfMonth());
System.out.println(endOfMonth.toString());                //2013-05-31

LocalDate nextTue = date.with(TemporalAdjusters.next(DayOfWeek.TUESDAY));
System.out.println(nextTue.toString());                 //2013-05-21
```

Creating date objects

Creating date objects now can be done using builder pattern also. The builder pattern allows the object you want to be built up using individual parts. This is achieved using the methods prefixed by "at".

```
//Builder pattern used to make date object
 OffsetDateTime date1 = Year.of(2013)
            .atMonth(Month.MAY).atDay(15)
            .atTime(0, 0)
            .atOffset(ZoneOffset.of(&quot;+03:00&quot;));
 System.out.println(date1);                    //2013-05-15T00:00+03:00
```

```
//factory method used to make date object
OffsetDateTime date2 = OffsetDateTime.
            of(2013, 5, 15, 0, 0, 0, 0, ZoneOffset.of(&quot;+03:00&quot;));
System.out.println(date2);                    //2013-05-15T00:00+03:00
```

New class to simulate system/machine clock

A new class Clock is proposed in new release. This **simulates the system clock functionality**. I loved this feature most of all others. The reason is while doing unit testing. you are often required to test a API in future date. For this we had been forwarding the system clock for next date, and then again restart the server and test the application.

Now, no need to do this. Use `Clock` class to simulate this scenario.

```
Clock clock = Clock.systemDefaultZone();
System.out.println(clock);             //SystemClock[Asia/Calcutta]
System.out.println(clock.instant().toString()); //2013-05-15T06:36:33.837Z
System.out.println(clock.getZone());      //Asia/Calcutta

Clock anotherClock = Clock.system(ZoneId.of(&quot;Europe/Tiraspol&quot;));
System.out.println(anotherClock);           //SystemClock[Europe/Tiraspol]
System.out.println(anotherClock.instant().toString());  //2013-05-15T06:36:33.857Z
System.out.println(anotherClock.getZone());       //Europe/Tiraspol

Clock forwardedClock  = Clock.tick(anotherClock, Duration.ofSeconds(600));
System.out.println(forwardedClock.instant().toString());  //2013-05-15T06:30Z
```

Timezone Changes

Timezone related handling is done by 3 major classes. These are ZoneOffset, TimeZone, ZoneRules.

- The `ZoneOffset` class represents a fixed offset from UTC in seconds. This is normally represented as a string of the format "±hh:mm".
- The `TimeZone` class represents the identifier for a region where specified time zone rules are defined.
- The `ZoneRules` are the actual set of rules that define when the zone-offset changes.

```
//Zone rules
System.out.println(ZoneRules.of(ZoneOffset.of(&quot;+02:00&quot;)).isDaylightSavings(Instant.now()));
System.out.println(ZoneRules.of(ZoneOffset.of(&quot;+02:00&quot;)).isFixedOffset());
```

Date Formatting

Date formatting is supported via two classes mainly i.e. `DateTimeFormatterBuilder` and `DateTimeFormatter`. `DateTimeFormatterBuilder` works on builder pattern to build custom patterns where as `DateTimeFormatter` provides necessary input in doing so.

```
DateTimeFormatterBuilder formatterBuilder = new DateTimeFormatterBuilder();
formatterBuilder.append(DateTimeFormatter.ISO_LOCAL_DATE_TIME)
```

```
        .appendLiteral(&quot;-&quot;)
        .appendZoneOrOffsetId();
DateTimeFormatter formatter = formatterBuilder.toFormatter();
System.out.println(formatter.format(ZonedDateTime.now()));
```

These are major changes which I was able to identify and worked on.

References

- https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html
- http://sourceforge.net/apps/mediawiki/threeten/index.php?title=User_Guide

Happy Learning !!

Was this post helpful?

Let us know if you liked the post. That's the only way we can improve.

Yes

No

Recommended Reading:

1. Java 8 Parse String to Date
2. Java Stream – Get Object with Max Date From a List
3. Java Strict Date Validation – Java SimpleDateFormat setLenient() Method
4. Guide to java.util.Date Class
5. Add or Subtract Days, Months & Years to Date in Java
6. Convert between LocalDate to java.sql.Date
7. Java Strict, Smart and Lenient Date Resolutions
8. Formatting Date to String in Java
9. Introduction to the Java Date/Time API
0. Finding the Day of Week for a Date in Java

Join 7000+ Awesome Developers

Get the latest updates from industry, awesome resources, blog updates and much more.

Email Address

Subscribe

** We do not spam !!*

2 thoughts on "Java 8 – Date and Time Examples"

Chris Milburn

February 6, 2020 at 5:15 pm

Why do the clocks in Asia/Calcutta time zone and Europe/Tiraspol show the 'same' time ?

Reply

Pawan

December 5, 2013 at 9:29 am

thanks for updating

Reply

Leave a Comment

Name *

Email *

Website

☐ Add me to your newsletter and keep me updated whenever you publish new blog posts

Post Comment

Search ...

Promoted by slack.com
Sponsored

A message from our sponsor

HowToDoInJava

A blog about Java and related technologies, the best practices, algorithms, and interview questions.

Meta Links

- About Me
- Contact Us
- Privacy policy
- Advertise
- Guest Posts

Blogs

REST API Tutorial