Java 8 Date & Time API

Overview

- Java 8 introduced new APIs for *Date* and *Time* to address the shortcomings of the older *java.util.Date* and *java.util.Calendar*.

Issues with the Existing Date/Time APIs

- **Thread Safety** – The Date and Calendar classes are not thread safe, leaving developers to deal with the headache of hard to debug concurrency issues and to write additional code to handle thread safety. On the contrary the new Date and Time APIs introduced in Java 8 are immutable and thread safe, thus taking that concurrency headache away from developers.
- **APIs Design and Ease of Understanding** – The Date and Calendar APIs are poorly designed with inadequate methods to perform day-to-day operations. The new Date/Time APIs is ISO centric and follows consistent domain models for date, time, duration and periods. There are a wide variety of utility methods that support the commonest operations.
- **ZonedDate and Time** – Developers had to write additional logic to handle timezone logic with the old APIs, whereas with the new APIs, handling of timezone can be done with Local and ZonedDate/Time APIs.

LocalDate, LocalTime and LocalDateTime

- LocalDate
  - The LocalDate represents a date in ISO format (yyyy-MM-dd) without time.
  - To create instance of current date from the system clock:
    - LocalDate localDate = LocalDate.now();
  - The LocalDate representing a specific day, month and year can be obtained using the "of" method or by using the "parse" method.
    - LocalDate.of(2015, 02, 20);
    - LocalDate.parse("2015-02-20");

```
LocalTime sixThirty = LocalTime.of(6, 30);
```

LocalTime

- LocalTime represents time without a date.
- To create instance of LocalTime from system clock or by using “parse” and “of” method.
  - LocalTime now = LocalTime.now();
  - LocalTime sixThirty = LocalTime.parse("06:30");
  - LocalTime sixThirty = LocalTime.of(6, 30);

LocalDateAndTime

- LocalDateTime is used to represent a combination of date and time.
- To create instance of LocalTime from system clock
  - LocalDateTime.now();
- The LocalDateAndTime representing a specific day, month ,year and time can be obtained using the “of” method or by using the “parse” method.
  - LocalDateTime.of(2015, Month.FEBRUARY, 20, 06, 30);
  - LocalDateTime.parse("2015-02-20T06:30:00");

```
LocalDate date = LocalDate.of(2014, 2, 15); // 2014-06-15
boolean isBefore = LocalDate.now().isBefore(date); // false
// information about the month
Month february = date.getMonth(); // FEBRUARY
int februaryIntValue = february.getValue(); // 2
int minLength = february.minLength(); // 28
int maxLength = february.maxLength(); // 29
Month firstMonthOfQuarter = february.firstMonthOfQuarter(); // JANUARY
// information about the year
int year = date.getYear(); // 2014
int dayOfYear = date.getDayOfYear(); // 46
int lengthOfYear = date.lengthOfYear(); // 365
boolean isLeapYear = date.isLeapYear(); // false
DayOfWeek dayOfWeek = date.getDayOfWeek();
int dayOfWeekIntValue = dayOfWeek.getValue(); // 6
String dayOfWeekName = dayOfWeek.name(); // SATURDAY
int dayOfMonth = date.getDayOfMonth(); // 15
```

```
LocalDateTime startOfDay = date.atStartOfDay(); // 2014-02-15 00:00
// time information
LocalTime time = LocalTime.of(15, 30); // 15:30:00
int hour = time.getHour(); // 15
int second = time.getSecond(); // 0
int minute = time.getMinute(); // 30
int secondOfDay = time.toSecondOfDay(); // 55800
```

- **Use the Year class to get information about a specific year:**

```
Year currentYear = Year.now();
Year twoThousand = Year.of(2000);
boolean isLeap = currentYear.isLeap(); // false
int length = currentYear.length(); // 365
// sixtyFourth day of 2014 (2014-03-05)
LocalDate date = Year.of(2014).atDay(64);
```

- **plus and minus methods to add or subtract specific amounts of time.**

```
LocalDate tomorrow = LocalDate.now().plusDays(1);
// before 5 houres and 30 minutes
LocalDateTime dateTime = LocalDateTime.now().minusHours(5).minusMinutes(30);
```

- TemporalAdjusters are another way for date manipulation.
- TemporalAdjuster is a single method interface that is used to separate the process of adjustment from actual date/time objects.
- A set of common TemporalAdjusters can be accessed using static methods of the TemporalAdjusters class.

```
LocalDate date = LocalDate.of(2014, Month.FEBRUARY, 25); // 2014-02-25
// first day of february 2014 (2014-02-01)
LocalDate firstDayOfMonth = date.with(TemporalAdjusters.firstDayOfMonth());
// last day of february 2014 (2014-02-28)
LocalDate lastDayOfMonth = date.with(TemporalAdjusters.lastDayOfMonth());
```

ZonedDateAndTime

- ZonedDateTime is used to deal with time zone specific date and time.
- **ZoneId** identifier is used to represent different zones.
- There are about 40 different time zones.

```
//to obtain zone id of paris
ZoneId zoneId = ZoneId.of("Europe/Paris");
//to obtain set of all zone ids
Set<String> allZoneIds = ZoneId.getAvailableZoneIds();
//LocalDateTime can be converted to a specific zone:
ZonedDateTime zonedDateTime = ZonedDateTime.of(localDateTime, zoneId);
```

ZonedDateTime provides *parse* method to get time zone specific date time

```
ZonedDateTime.parse("2015-05-03T10:15:30+01:00[Europe/Paris]");
```

```
ZoneId losAngeles = ZoneId.of("America/Los_Angeles");
ZoneId berlin = ZoneId.of("Europe/Berlin");
// 2014-02-20 12:00
LocalDateTime dateTime = LocalDateTime.of(2014, 02, 20, 12, 0);
// 2014-02-20 12:00, Europe/Berlin (+01:00)
ZonedDateTime berlinDateTime = ZonedDateTime.of(dateTime, berlin);
// 2014-02-20 03:00, America/Los_Angeles (-08:00)
ZonedDateTime losAngelesDateTime = berlinDateTime.withZoneSameInstant(losAngeles);
int offsetInSeconds = losAngelesDateTime.getOffset().getTotalSeconds(); // -28800
// a collection of all available zones
Set<String> allZoneIds = ZoneId.getAvailableZoneIds();
// using offsets
LocalDateTime date = LocalDateTime.of(2013, Month.JULY, 20, 3, 30);
ZoneOffset offset = ZoneOffset.of("+05:00");
// 2013-07-20 03:30 +05:00
OffsetDateTime plusFive = OffsetDateTime.of(date, offset);
// 2013-07-19 20:30 -02:00
OffsetDateTime minusTwo = plusFive.withOffsetSameInstant(ZoneOffset.ofHours(-2));
```

Using Period and Duration

- ***Period*** class represents a quantity of time in terms of years, months and days

```
LocalDate finalDate = initialDate.plus(Period.ofDays(5));

int five = Period.between(finalDate, initialDate).getDays();
```

- ***Duration*** class represents a quantity of time in terms of seconds and nano seconds.

```
LocalTime initialTime = LocalTime.of(6, 30, 0);
LocalTime finalTime = initialTime.plus(Duration.ofSeconds(30));

int thirty = Duration.between(finalTime, initialTime).getSeconds();
```

```
// periods
LocalDate firstDate = LocalDate.of(2010, 5, 17); // 2010-05-17
LocalDate secondDate = LocalDate.of(2015, 3, 7); // 2015-03-07
Period period = Period.between(firstDate, secondDate);
int days = period.getDays(); // 18
int months = period.getMonths(); // 9
int years = period.getYears(); // 4
boolean isNegative = period.isNegative(); // false
Period twoMonthsAndFiveDays = Period.ofMonths(2).plusDays(5);
LocalDate sixthOfJanuary = LocalDate.of(2014, 1, 6);
// add two months and five days to 2014-01-06, result is 2014-03-11
LocalDate eleventhOfMarch = sixthOfJanuary.plus(twoMonthsAndFiveDays);
// durations
Instant firstInstant= Instant.ofEpochSecond( 1294881180 ); // 2011-01-13 01:13
Instant secondInstant = Instant.ofEpochSecond(1294708260); // 2011-01-11 01:11
Duration between = Duration.between(firstInstant, secondInstant);
// negative because firstInstant is after secondInstant (-172920)
long seconds = between.getSeconds();
// get absolute result in minutes (2882)
long absoluteResult = between.abs().toMinutes();
// two hours in seconds (7200)
long twoHoursInSeconds = Duration.ofHours(2).getSeconds();
```

Compatibility with Date and Calender

- Java 8 has added the toInstant() method which helps to convert existing Date and Calendar instance to new Date Time API.

```
LocalDateTime.ofInstant(date.toInstant(), ZoneId.systemDefault());
LocalDateTime.ofInstant(calendar.toInstant(), ZoneId.systemDefault());
```

Temporal Adjusters

- TemporalAdjuster is used to perform the date mathematics.
- For example, get the "Next Tuesday".

```
//Get the current date
     LocalDate date1 = LocalDate.now();
     System.out.println("Current date: " + date1);

     //get the next tuesday
     LocalDate nextTuesday = date1.with(TemporalAdjusters.next(DayOfWeek.TUESDAY));
     System.out.println("Next Tuesday on : " + nextTuesday);
```

Output:

```
Current date: 2014-12-10
Next Tuesday on : 2014-12-16
```