Java 8 Date Time - 20 Examples of LocalDate, LocalTime, LocalDateTime

Along with lambda expressions, streams, and several minor goodies, Java 8 has also introduced brand new Date and Time API, and in this tutorial, we will learn how to use Java 8 Date Time API with simple how-to-do task examples. Java's handling of Date, Calendar, and Time is long been criticized by the community, which is not helped by Java's decision of making `java.util.Date` mutable and SimpleDateFormat not thread-safe. It seems, Java has realized a need for better Date and time support, which is good for a community which already used to of Joda Date and Time API.

One of the many good things about new Date and Time API is that now it defines principle date-time concepts e.g. instants, duration, dates, times, timezones, and periods. It also follows good things from the Joda library about keeping human and machine interpretation of date-time separated.

hey are also based on the ISO Calendar system and unlike their predecessor, class in `java.time` packages are both immutable and thread-safe. New Date and time API is located inside `java.time` package and some of the key classes are the following :

- Instant - It represents a timestamp
- LocalDate - a date without time e.g. 2014-01-14. It can be used to store birthday, anniversary, date of joining etc.
- LocalTime - represents time without a date
- LocalDateTime - is used to combine date and time, but still without any offset or time-zone
- ZonedDateTime - a complete date-time with time-zone and resolved offset from UTC/Greenwich

They are also coming with better time zone support with `ZoneOffSet` and `ZoneId`. Parsing and Formatting of Dates are also revamped with the new `DateTimeFormatter` class. By the way, just remember that I wrote this article almost a year ago when Java was about to launch, so you will find examples that have dates of the previous year. When you will run those samples, it will surely return the correct values.

Btw, if you are not familiar with the new Date and Time API added on Java 8 then I suggest you first go through a comprehensive and up-to-date Java course like **The Complete Java MasterClass** on Udemy. It's also very affordable and you can buy in just $10 on Udemy sales which happen every now and then.

How to do Date and Time in Java 8

Someone asked me what is the best way to learn a new library? My answer was, use that library as if you are using it for your real project. There are too many real requirements in a real project, which prompts a developer to explore and learn a new library. In short, it's a task that motivates you to explore and learn new APIs. Java 8's new date and time API is no different.

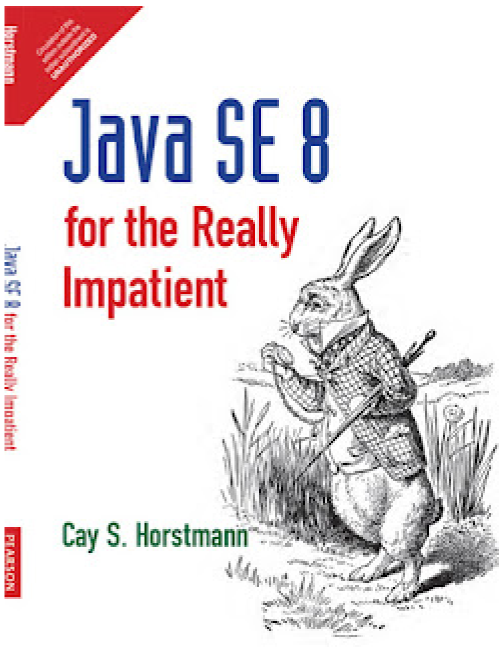
I have created a list of *20 task-based examples* to learn this new gem from Java 8. We will start with a simple task e.g. how to represent today's date using the Java 8 Date Time library then move forward to create a date with time and time zone, exploring how to do more real-world tasks like creating a reminder application how to find a number of days to important dates e.g. birthday, anniversary, next bill date, next premium date, your credit card expiry, etc.

Example 1 - How to get today's date in Java 8

Java 8 has a class called `LocalDate` which can be used to represent today's date. This class is little different than `java.util.Date` because it only contains the date, no time part. So anytime if you just to represent date without time, use this class.

```
LocalDate today = LocalDate.now();
System.out.println("Today's Local date : " + today);

Output
Today's Local date : 2014-01-14
```

You can see that it has created today's date without any time information. It also prints the date in a nicely formatted way, unlike the previous `Date` class which prints data non-formatted. You can also see [Java SE 8 for Really Impatient]() to learn about different ways to create LocalDate in Java 8.



Example 2 - How to get a current day, month, and year in Java 8

The `LocalDate` class has a convenient method to extract the year, month, day of the month, and several other dates attributes from an instance of `LocalDate` class. By using these methods, you can get whatever property of date you want, no need to use a supporting class like `java.util.Calendar`:

```
LocalDate today = LocalDate.now();
int year = today.getYear();
int month = today.getMonthValue();
int day = today.getDayOfMonth();
```

```
System.out.printf("Year : %d  Month : %d  day : %d \t %n", year, month, day);

Output
Today's Local date : 2014-01-14
Year : 2014  Month : 1  day : 14
```

You can see how easy it is to get a year or month from a date in Java 8, just use the corresponding getter method, nothing to remember, very intuitive. Compare this with the older way of getting the current date, month, and year in Java.

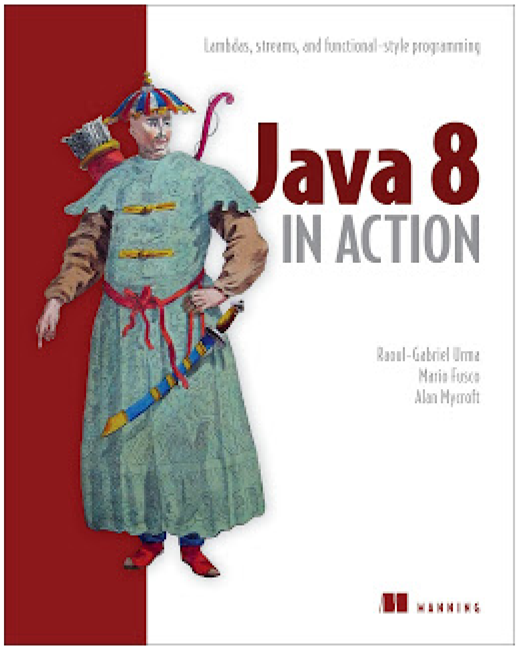
Example 3 - How to get a particular date in Java 8

In the first example, we have seen that creating today's date was very easy because of the static factory method `now()`, but you can also create a date from any arbitrary date by using another useful factory method called `LocalDate.of()`, this takes a year, month and date and return an equivalent `LocalDate` instance.

The good thing about this method is that it has not repeated mistakes done in previous API e.g. year started from 1900, months starting from zero, etc. Here dates are represented in the way you write it e.g. in the following example it will represent 14th January, nothing is hidden about it.

```
LocalDate dateOfBirth = LocalDate.of(2010, 01, 14);
System.out.println("Your Date of birth is : " + dateOfBirth);

Output : Your Date of birth is : 2010-01-14
```

You can see that as expected the date created is exactly the same as written and represents 14th January 2014. See these Java 8 courses and books to learn more about the difference between LocalDate and java.util.Date in Java 8.



Example 4 - How to check if two dates are equal in Java 8

Talking about real world date time tasks, one of them is to check whether two dates are same or not. Many times you would like to check whether today is that special day, your birthday, anniversary or a trading holiday or not. Sometimes, you will get an arbitrary date and you need to check against certain date e.g. holidays to confirm whether given date is a holiday or not.

This example will help you to accomplish those task in Java 8. Just like you thought, `LocalDate` has overridden equal method to provide date equality, as shown in the following example :

```
LocalDate date1 = LocalDate.of(2014, 01, 14);
if(date1.equals(today)){
    System.out.printf("Today %s and date1 %s are same date %n", today, date1);
}

Output
today 2014-01-14 and date1 2014-01-14 are same date
```

In this example the two dates we compared are equal. BTW, If you get a formatted date String in your code, you will have to parse that into a date before checking equality. Just compare this with the older way of comparing dates in Java, you will find it a fresh breeze.

Example 5 - How to check for recurring events e.g. birthday in Java 8

Another practical task related to date and time in Java is checking for recurring events e.g. monthly bills, wedding anniversary, EMI date or yearly insurance premium dates. If you are working for an E-commerce site, you would definitely have a module which sends birthday wishes to your customer and seasons greetings on every major holiday e.g. Christmas, Thanksgiving date or Deepawali in India.

How do you check for holidays or any other recurring event in Java? By using `MonthDay` class. This class is a combination of month and date without a year, which means you can use it for events that occur every year.

There are similar classes exists for other combination as well e.g. `YearMonth`. Like other classes in the new date and time API, this is also immutable and thread-safe and it is also a value class. Now let's see an example of how to use `MonthDay` class for checking recurring date time events :

```
LocalDate dateOfBirth = LocalDate.of(2010, 01, 14);
MonthDay birthday = MonthDay.of(dateOfBirth.getMonth(),
                                        dateOfBirth.getDayOfMonth());
MonthDay currentMonthDay = MonthDay.from(today);

if(currentMonthDay.equals(birthday)){
   System.out.println("Many Many happy returns of the day !!");
}else{
   System.out.println("Sorry, today is not your birthday");
}

Output:
Many Many happy returns of the day !!
```

Since today's date matches the birthday, irrespective of year you have seen the birthday greeting as output. You can run this program by advancing your windows date and time clock and see if it alerts you on your next birthday or not, or you can write a JUnit test with the date of your next year's birthday and see if your code runs properly or not.

Example 6 - How to get current Time in Java 8

This is very similar to our first example of getting the current date in Java 8. This time, we will use a class called `LocalTime`, which is the time without date and a close cousin of `LocalDate` class. Here also you can use the static factory method `now()` to get the current time. The default format is `hh:mm:ss:nnn` where nnn is nanoseconds. BTW, compare this how to get current time before Java 8.

```
LocalTime time = LocalTime.now();
System.out.println("local time now : " + time);

Output
local time now : 16:33:33.369  // in hour, minutes, seconds, nano seconds
```

You can see that the current time has no date attached to it because LocalTime is just time, no date.

Example 7 - How to add hours in time

On many occasions, we would like to add hours, minutes or seconds to calculate time in the future. Java 8 has not only helped with Immutable and thread-safe classes but also provided better methods e.g. `plusHours()` instead of `add()`, there is no conflict. BTW, remember that these methods return a reference to new `LocalTime` instance because `LocalTime` is immutable, so don't forget to store them back.

```
LocalTime time = LocalTime.now();
LocalTime newTime = time.plusHours(2); // adding two hours
System.out.println("Time after 2 hours : " +  newTime);

Output :
Time after 2 hours : 18:33:33.369
```

You can see that the new time is 2 hours ahead of the current time which is 16:33:33.369. Now, try to compare this with older ways of adding and subtracting hours from a date in Java. Let us know which one is better.

Example 8 - How to find Date after 1 week

This is similar to the previous example, there we learned how to find time after 2 hours, and here we will learn how to find a date after 1 week. `LocalDate` is used to represent date without the time and it got a `plus()` method which is used to add days, weeks, or months, `ChronoUnit` is used to specify that unit. Since `LocalDate` is also immutable any mutable operation will result in a new instance, so don't forget to store it back.

```
LocalDate nextWeek = today.plus(1, ChronoUnit.WEEKS);
System.out.println("Today is : " + today);
System.out.println("Date after 1 week : " + nextWeek);

Output:
Today is : 2014-01-14
Date after 1 week : 2014-01-21
```

You can see that new date is 7 days away from the current date, which is equal to 1 week. You can use the same method to add 1 month, 1 year, 1 hour, 1 minute and even 1 decade, check out `ChronoUnit` class from Java 8 API for more options.

Example 9 - Date before and after 1 year

This is a continuation of the previous example. In the last example, we learn how to use `plus()` method of `LocalDate` to add days, weeks or months in a date, now we will learn how to use the `minus()` method to find what was the day before 1 year.

```
LocalDate previousYear = today.minus(1, ChronoUnit.YEARS);
System.out.println("Date before 1 year : " + previousYear);

LocalDate nextYear = today.plus(1, YEARS);
System.out.println("Date after 1 year : " + nextYear);
```

```
Output:
Date before 1 year : 2013-01-14
Date after 1 year : 2015-01-14
```

You can see that we now have two years, one is in 2013, and the other is in 2015, the year before and after the current year 2014.

Example 10 - Using Clock in Java 8

Java 8 comes with a Clock, which can be used to get current instant, date and time using time zone. You can use Clock in place of `System.currentTimeInMillis()` and `TimeZone.getDefault()`.

```
// Returns the current time based on your system clock and set to UTC.
Clock clock = Clock.systemUTC();
System.out.println("Clock : " + clock);

// Returns time based on system clock zone
Clock defaultClock = Clock.systemDefaultZone();
System.out.println("Clock : " + clock);

Output:
Clock : SystemClock[Z]
Clock : SystemClock[Z]
```

You can check given date against this clock, as shown below :

```
public class MyClass {
    private Clock clock;  // dependency inject
    ...
    public void process(LocalDate eventDate) {
      if (eventDate.isBefore(LocalDate.now(clock)) {
        ...
      }
    }
}
```

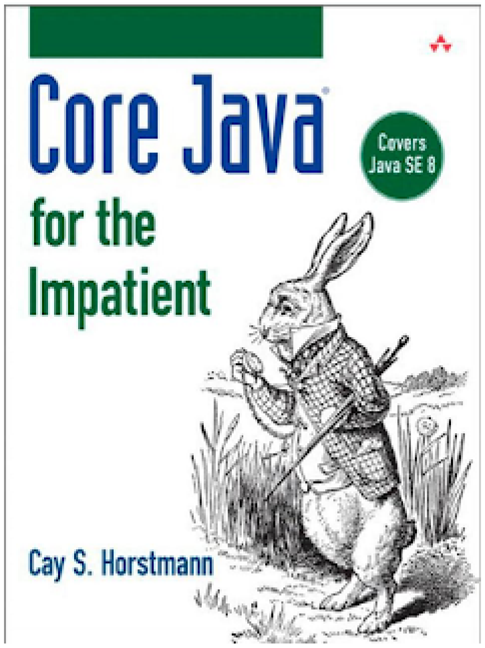
This could be useful if you want to process dates on the different time zone.

Example 11 - How to see if a date is before or after another date in Java

This is another very common task in an actual project. How do you find if a given date is before, or after the current date or just another date? In Java 8, `LocalDate` class has got methods like `isBefore()` and `isAfter()` which can be used to compare two dates in Java. `isBefore()` method return `true` if given date comes before the date on which this method is called.

```
LocalDate tomorrow = LocalDate.of(2014, 1, 15);

if(tommorow.isAfter(today)){
    System.out.println("Tomorrow comes after today");
}

LocalDate yesterday = today.minus(1, DAYS);

if(yesterday.isBefore(today)){
    System.out.println("Yesterday is day before today");
}

Output:
Tomorrow comes after today
Yesterday is day before today
```

You can see how easy it is to compare dates in Java 8. You don't need to use another class like Calendar to perform such essential tasks. You can see Core Java for the Impatient to learn more ways to compare Dates in Java.



Example 12 - Dealing with time zones in Java 8

Java 8 has not only separated date and time but also timezone. You now have a separate set of classes related to timezone e.g. `ZonId` to represent a particular timezone and `ZonedDateTime` class to represent a date-time with timezone. It's equivalent of the GregorianCalendar class in pre-Java 8 world. By using this class, you can convert local time to equivalent time in another time zone as shown in the following example :

```
// Date and time with timezone in Java 8
ZoneId america = ZoneId.of("America/New_York");
LocalDateTime localtDateAndTime = LocalDateTime.now();
ZonedDateTime dateAndTimeInNewYork  = ZonedDateTime.of(localtDateAndTime,
                                                        america );
System.out.println("Current date and time in a particular timezone : "
                         + dateAndTimeInNewYork);

Output :
Current date and time in a particular timezone :
               2014-01-14T16:33:33.373-05:00[America/New_York]
```

Compare this with the older way of converting local time to GMT. By the way, just like before Java 8, don't forget to use the correct text for time zones, otherwise, you would be greeted with the following exception :

```
Exception in thread "main" java.time.zone.ZoneRulesException: Unknown time-zone ID: ASIA/Tokyo
        at java.time.zone.ZoneRulesProvider.getProvider(ZoneRulesProvider.java:272)
        at java.time.zone.ZoneRulesProvider.getRules(ZoneRulesProvider.java:227)
        at java.time.ZoneRegion.ofId(ZoneRegion.java:120)
        at java.time.ZoneId.of(ZoneId.java:403)
        at java.time.ZoneId.of(ZoneId.java:351)
```

Example 13 - How to represent fixed date e.g. credit card expiry, YearMonth

Like our `MonthDay` example for checking recurring events, `YearMonth` is another combination class to represent things like credit card expires, FD maturity date, Futures or options expiry dates etc. You can also use this class to find how many days are in the current month, `lengthOfMonth()` returns a number of days in the current `YearMonth` instance, useful for checking whether February has 28 or 29 days.

```
YearMonth currentYearMonth = YearMonth.now();
System.out.printf("Days in month year %s: %d%n", currentYearMonth,
                                 currentYearMonth.lengthOfMonth());
YearMonth creditCardExpiry = YearMonth.of(2018, Month.FEBRUARY);
System.out.printf("Your credit card expires on %s %n", creditCardExpiry);

Output:
Days in month year 2014-01: 31
Your credit card expires on 2018-02
```

Based on this data, you can now send a reminder to the customer about his credit card expiry, a very useful class in my opinion.

Example 14 - How to check Leap Year in Java 8

Nothing fancy here, `LocalDate` class has `isLeapYear()` method which returns true if the year represented by that `LocalDate` is a leap year. If you still want to reinvent the wheel, check out this code sample, which contains a Java program to find if a given year is leap using pure logic.

```
if(today.isLeapYear()){
   System.out.println("This year is Leap year");
}else {
    System.out.println("2014 is not a Leap year");
}

Output:
2014 is not a Leap year
```

You can further check some more year to see if it correctly identify a leap year or not, better write a JUnit test to check for normal and leap year.

Example 15 - How many days, the month between two dates

One of the common tasks is to calculate the number of days, weeks or months between two given dates. You can use `java.time.Period` class to calculate the number of days, month or year between two dates in Java. In the following example, we have calculated the number of months between the current date and a future date.

```
LocalDate java8Release = LocalDate.of(2014, Month.MARCH, 14);
Period periodToNextJavaRelease = Period.between(today, java8Release);
System.out.println("Months left between today and Java 8 release : "
                                    + periodToNextJavaRelease.getMonths() );
Output:
Months left between today and Java 8 release : 2
```

You can see that the current month is January and Java 8 release is scheduled in March, so 2 months away.

Example 16 - Date and Time with timezone offset

In Java 8, you can use `ZoneOffset` class to represent a time zone, for example, India is GMT or UTC +05:30 and to get a corresponding timezone you can use static method `ZoneOffset.of()` method. Once you get the offset you can create an OffSetDateTime by passing LocalDateTime and offset to it.

```
LocalDateTime datetime = LocalDateTime.of(2014, Month.JANUARY, 14, 19, 30);
ZoneOffset offset = ZoneOffset.of("+05:30");
OffsetDateTime date = OffsetDateTime.of(datetime, offset);
System.out.println("Date and Time with timezone offset in Java : " + date);

Output :
Date and Time with timezone offset in Java : 2014-01-14T19:30+05:30
```

You can see the timezone attached to date and time now. BTW, OffSetDateTime is meant for machines for human dates prefer ZoneDateTime class.

Example 17 - How to get current timestamp in Java 8

If you remember how to get the current timestamp before Java 8 then this would be a breeze. The instant class has a static factory method now() which return current timestamp, as shown below :

```
Instant timestamp = Instant.now();
System.out.println("What is value of this instant " + timestamp);

Output :
What is value of this instant 2014-01-14T08:33:33.379Z
```

You can see that current timestamp has both date and time component, much like java.util.Date, in fact, Instant is your equivalent class of pre-Java 8 Date and you can convert between Date and Instant using respective conversion method added in both of these classes e.g. Date.from(Instant) is used to convert Instant to java.util.Date in Java and Date.toInstant() returns an Instant equivalent of that Date class.

Example 18 - How to parse/format date in Java 8 using predefined formatting

Date and time formatting was very tricky in pre-Java 8 world, our only friend SimpleDateFormat was not threading safe and quite bulky to use as a local variable for formatting and parsing numerous date instances. Thankfully, thread-local variables made it usable in a multi-threaded environment but Java has come a long way from there.

It introduced a brand new date and time formatter which is thread-safe and easy to use. It now comes with some predefined formatter for common date patterns. For example, in this sample code, we are using predefined `BASIC_ISO_DATE` formatter, which uses the format `20140114` for January 14, 214.

```
String dayAfterTommorrow = "20140116";
LocalDate formatted = LocalDate.parse(dayAfterTommorrow,
                        DateTimeFormatter.BASIC_ISO_DATE);
System.out.printf("Date generated from String %s is %s %n",
                    dayAfterTommorrow, formatted);

Output :
Date generated from String 20140116 is 2014-01-16
```

You can clearly see that the generated date has the same value as given String, but with different date patterns.

Example 19 - How to parse the date in Java using custom formatting

In the last example, we have used an inbuilt date and time formatter to parse date strings in Java. Sure, predefined formatters are great but there would be a time when you want to use your own custom date pattern and in that case, you have to create your own custom date-time formatter instances as shown in this example. The following example has a date in the format "`MMM dd yyyy`".

You can create a `DateTimeFormatter` with any arbitrary pattern by using `ofPattern()` static method, it follows the same literals to represent a pattern as before e.g. M is still a month and m is still a minute. An Invalid pattern will throw `DateTimeParseException` but a logically incorrect where you use m instead of M will not be caught.

```
String goodFriday = "Apr 18 2014";
try {
    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("MMM dd yyyy");
    LocalDate holiday = LocalDate.parse(goodFriday, formatter);
    System.out.printf("Successfully parsed String %s, date is %s%n",
                      goodFriday, holiday);
} catch (DateTimeParseException ex) {
    System.out.printf("%s is not parsable!%n", goodFriday);
    ex.printStackTrace();
}
Output :
Successfully parsed String Apr 18 2014, date is 2014-04-18
```

You can see that the value of Date is the same as the String passed, just they are formatted differently.

Example 20 - How to convert Date to String in Java 8, formatting dates

In the last two examples, though we have been using `DateTimeFormatter` class we are mainly parsing a formatted date String. In this example, we will do the exact opposite. Here we have a date, instance of `LocalDateTime` class and we will convert into a formatted date String. This is by far the simplest and easiest way to convert Date to String in Java.

The following example will return the formatted String in place of Date. Similar to the previous example, we still need to create a `DateTimeFormatter` instance with the given pattern but now instead of calling `parse()` method of LocalDate class, we will call `format()` method.

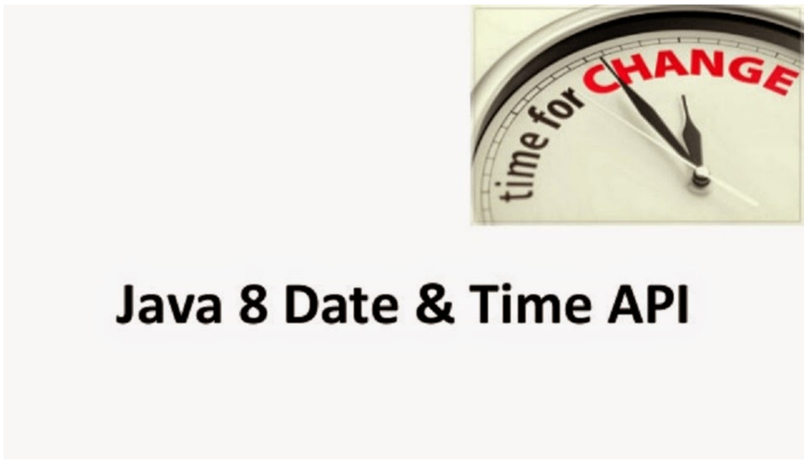
This method returns a String that represents a date in a pattern represented bypassed DateTimeFormatter instance.

```
LocalDateTime arrivalDate  = LocalDateTime.now();
try {
    DateTimeFormatter format = DateTimeFormatter
                  .ofPattern("MMM dd yyyy  hh:mm a");
    String landing = arrivalDate.format(format);
    System.out.printf("Arriving at :  %s %n", landing);
} catch (DateTimeException ex) {
    System.out.printf("%s can't be formatted!%n", arrivalDate);
    ex.printStackTrace();
}
Output : Arriving at :  Jan 14 2014  04:33 PM
```

You can see that current time is represented in given "`MMM dd yyyy hh:mm a`" pattern which includes three-letter month representation followed by time with AM and PM literals.

Important points about Date and Time API of Java 8

After going through all these examples, I am sure you have learned a lot of new things about Java 8 Date and Time API. Now let's take a step back and recall some important points about this beautiful API.



1) Provides `javax.time.ZoneId` to deal with timezones.

2) Provides `LocalDate` and `LocalTime` classes

3) All classes of new Date and Time API of Java 8 are Immutable and thread-safe, as opposed to existing Date and Calendar API, where key classes e.g. `java.util.Date` or SimpleDateFormat is not thread-safe.

4) The key thing to new Date and Time API is that it defines principle date-time concepts e.g. instants, durations, dates, times, timezones, and periods. They are also based on the ISO calendar system.

3) Every Java developer should at least know following five classes from new Date and Time API :

- Instant - It represents a time-stamp e.g. `2014-01-14T02:20:13.592Z` and can be obtained from java.time.Clock class as shown below :
  Instant current = Clock.system(ZoneId.of("Asia/Tokyo")).instant();
- LocalDate - represents a date without a time e.g. 2014-01-14. It can be used to store birthday, anniversary, date of joining etc.
- LocalTime - represents time without a date
- LocalDateTime - is used to combine date and time, but still without any offset or time-zone
- ZonedDateTime - a complete date-time with time-zone and resolved offset from UTC/Greenwich

4) The main package is `java.time`, which contains classes to represent dates, times, instants, and durations. Two sub-packages are `java.time.format` for obvious reasons and `java.time.temporal` for a lower level access to fields.

5) A time zone is a region of the earth where the same standard time is used. Each time zone is described by an identifier and usually has the format region/city (Asia/Tokyo) and an offset from Greenwich/UTC time. For example, the offset for Tokyo is `+09:00`.

6) The `OffsetDateTime` class, in effect, combines the `LocalDateTime` class with the `ZoneOffset` class. It is used to represent a full date (year, month, day) and time (hour, minute, second, nanosecond) with an offset from Greenwich/UTC time (+/-hours:minutes, such as `+06:00` or `-08:00`).

7) The `DateTimeFormatter` class is used to format and parse dates in Java. Unlike SimpleDateFormat, this is both immutable and thread-safe and it can (and should) be assigned to a static constant where appropriate. `DateFormatter` class provides numerous predefined formatters and also allows you to define your own.

Following the convention, it also has `parse()` to convert String to Date in Java and throws `DateTimeParseException`, if any problem occurs during conversion. Similarly, `DateFormatter` has a `format()` method to format dates in Java, and it throws `DateTimeException` error in case of a problem.

8) Just to add, there is subtle difference between date formats "`MMM d yyyy`" and "`MMM dd yyyy`", as former will recognize both "`Jan 2, 2014`" and "`Jan 14, 2014`", but later will throw exception when you pass "`Jan 2, 2014`", because it always expects two characters for day of the month. To fix this, you must always pass single digit days as two digits by padding zero at beginning e.g. "`Jan 2, 2014`" should be passed as "`Jan 02 2014`"