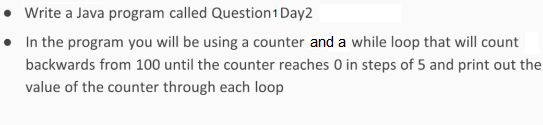
Hands-On Exercises - Java Loops, Strings, Numbers and Dates

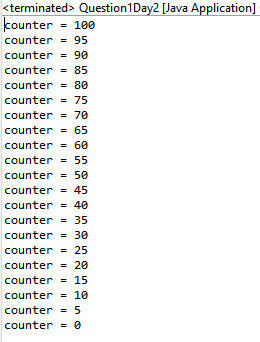
**Setup Instructions:**

1. Create a new Java Project called “JavaFundamentals2”.
2. Create your classes in any package of your choosing.
3. Define your classes as given below for each assignment.

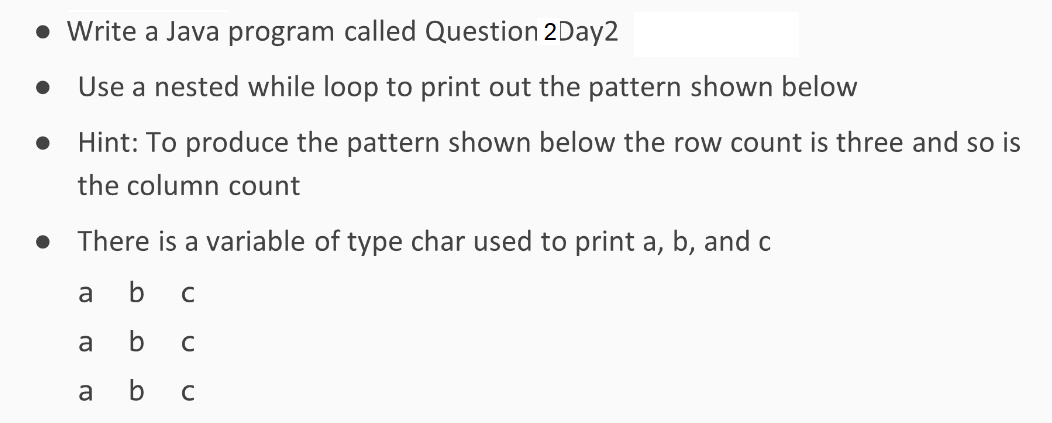
Assignment 01



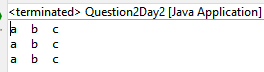
Output



Assignment 02



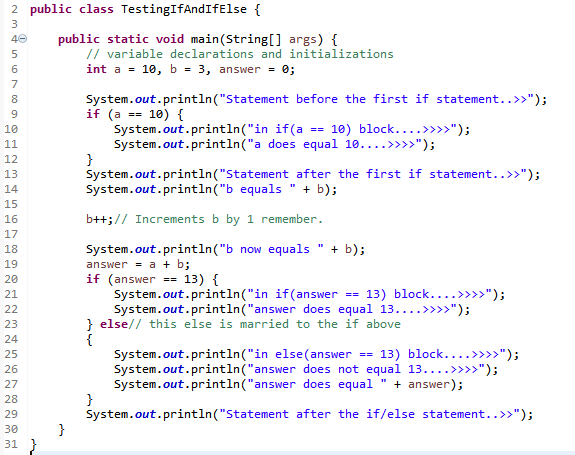
Output



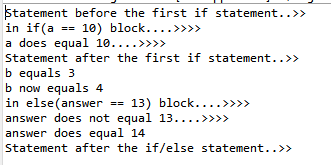
Assignment 03

**TestingIfAndIfElse.java**

**Create the folowing class to test If and Else Statements in java:**



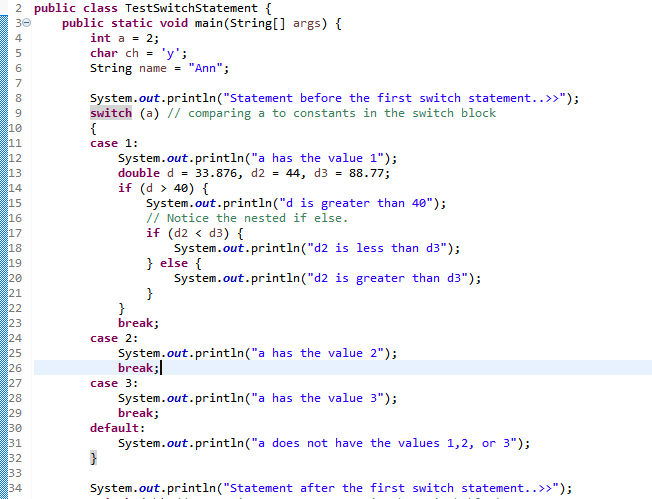
Output

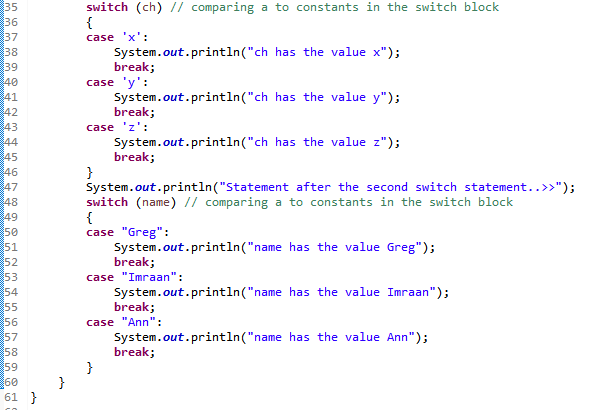


Assignment 04

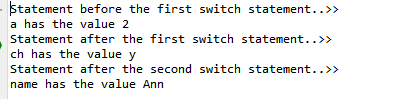
**TestSwitchStatement.java**

**Create the folowing class to test Switch Statements in java:**

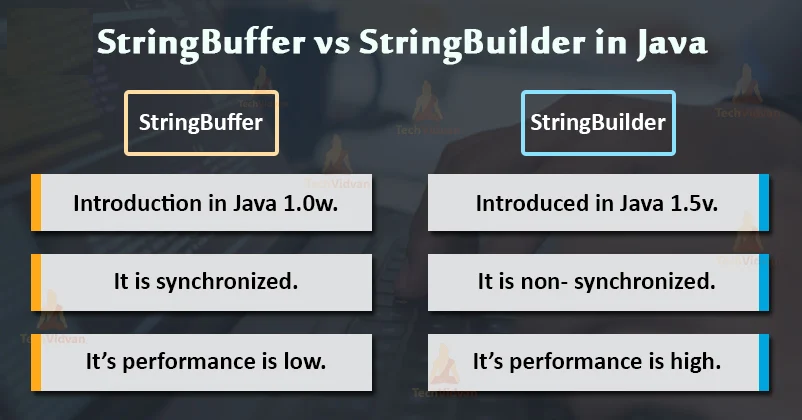




Output



Assignment 05



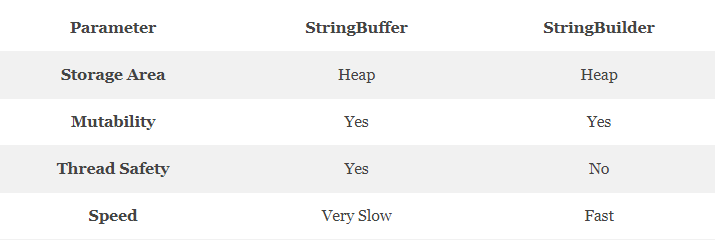
### What is a StringBuilder Class in Java?

Java StringBuilder Class eliminates the StringBuffer only because of the multithreading purpose.

The StringBuilder class was introduced in Java 1.5v after StringBuffer class having an additional functionality of non-synchronized methods so that multiple threads can be allowed to use StringBuilder objects.

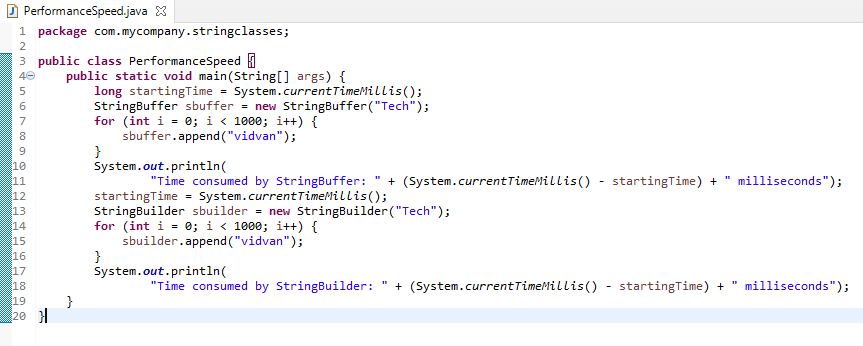
This results in the risk of Thread safety if the user runs the same method at many places at the same time. But, the StringBuilder class is more preferable to use as it works efficiently in a multithreaded environment.

Below are some points which some are common for both the classes and some are different.

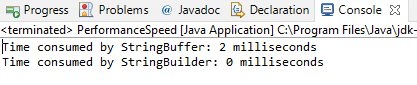


**PerformanceSpeed.java**

**Create the following class to test performance of the String Builder and String Buffer classes in java:**



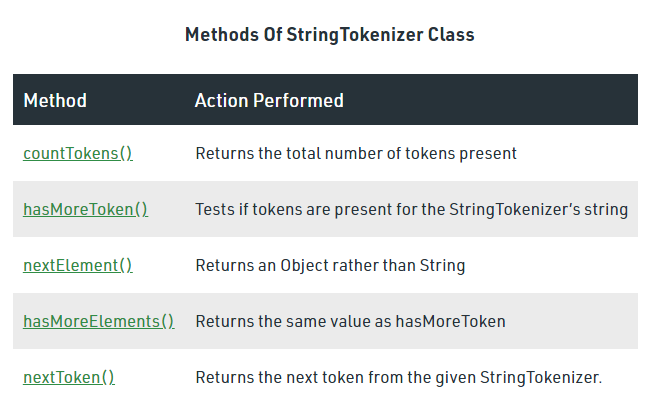
Output



From the above output, we can easily conclude that the StringBuffer takes more time as compared to the StringBuilder class.

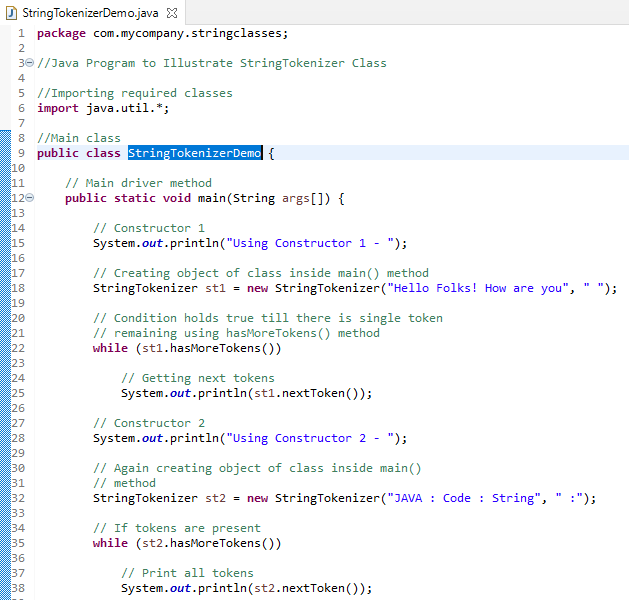
This output may vary from machine to machine and even the output may be different if you compile it several times.

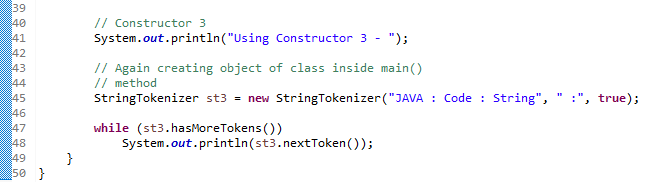
Assignment 05



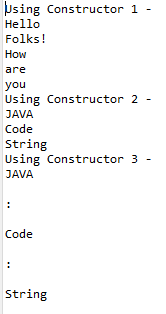
**StringTokenizerDemo.java**

**Create the folowing class to illustrate String Tokenizing in java:**



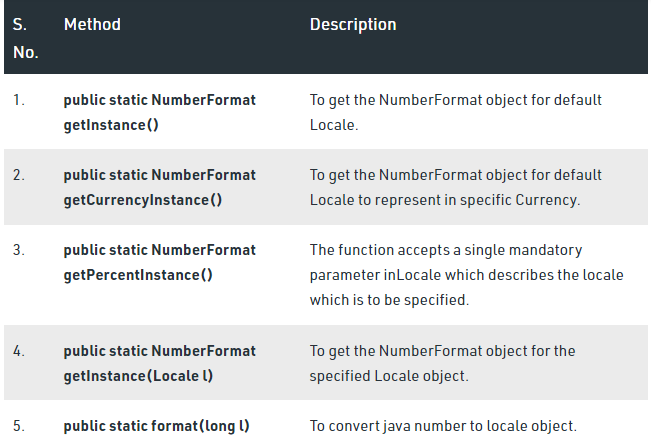


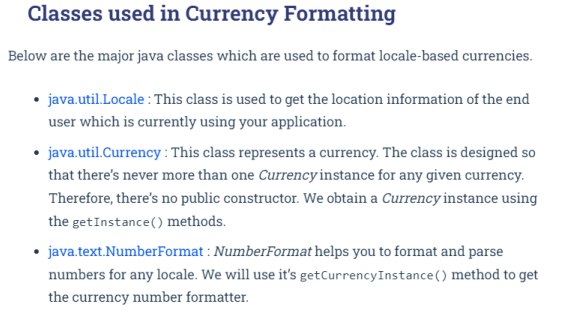
Output



Assignment 06

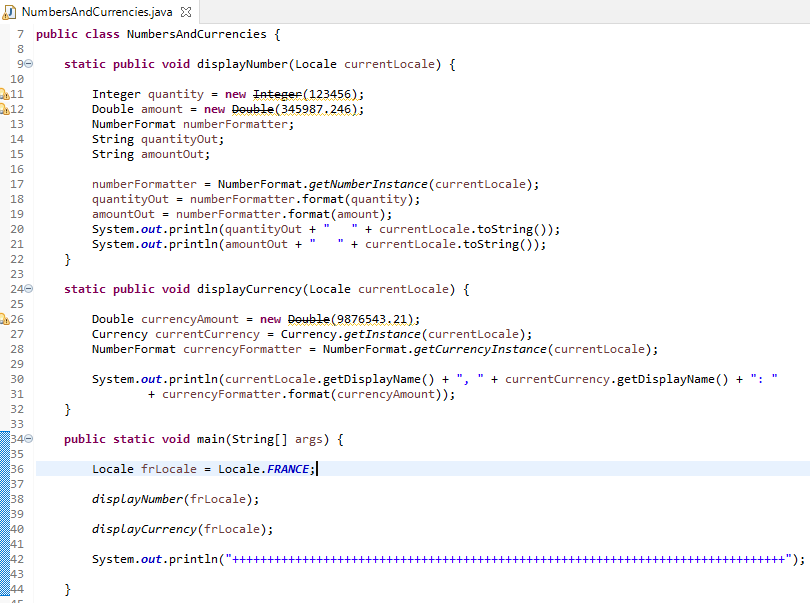
This exercise will demonstrate formatting numbers and currencies in a locale-specific manner by using predefined formats in the NumberFormat class.





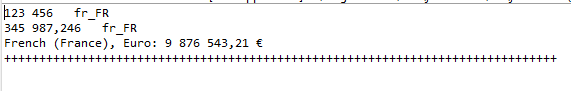
**NumbersAndCurrencies.java**

**Create the following class to demonstrate Number and Currency in java:**





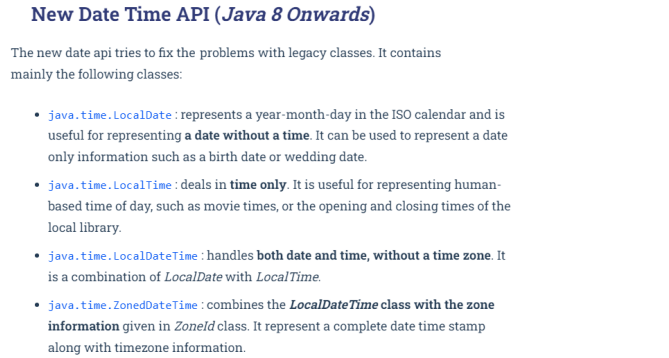
Output

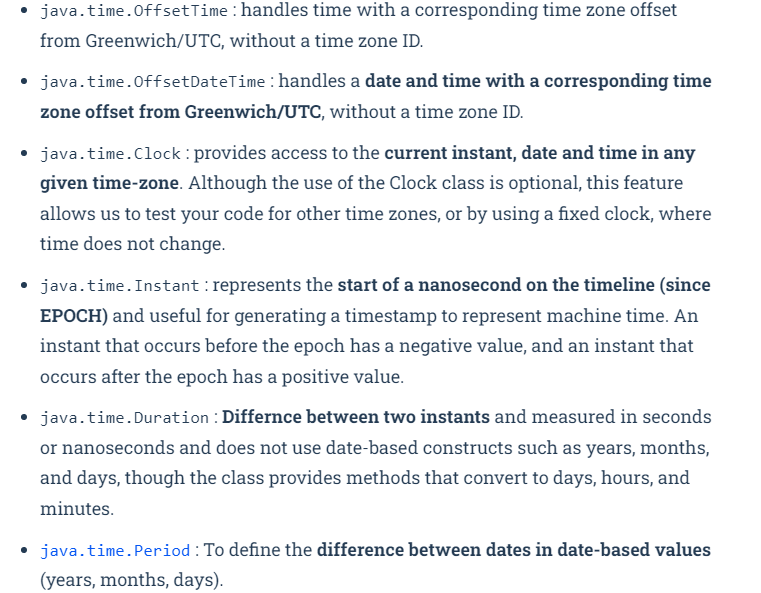


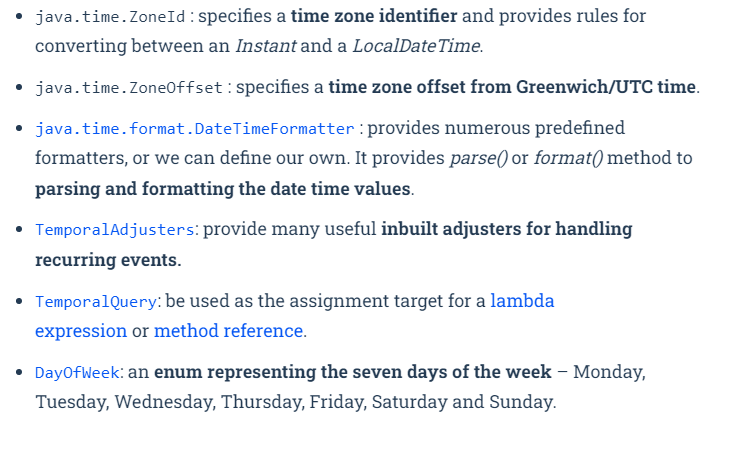
Try out othe locales like Germany, US, India etc.

Assignment 06

This exercise will demonstrate usage of the new Date Time API in Java 8.

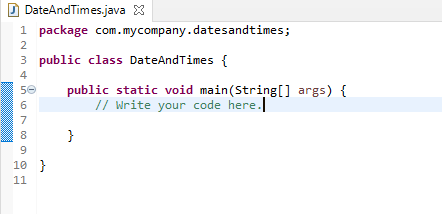






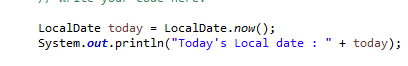
**DateAndTimeAPI.java**

**Create the following class to demonstrate some of the new features introduced into the Date And Time API in Java 8. The code snippets that follow need to be placed within the main method of the following class:**



### 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.

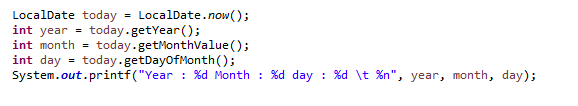


Output



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.

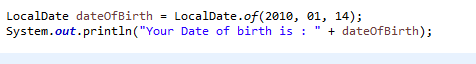
### Example 2 - How to get a current day, month, and year in Java 8

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 :  
  


Output



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

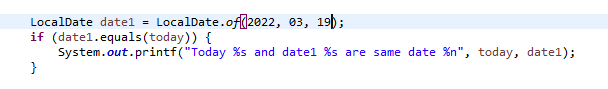
In the first example, we have seen that creating today's date was very easy because of the static factory method now(), but you can also create a date from any arbitrary date by using another useful factory method called LocalDate.of(), this takes a year, month and date and return an equivalent LocalDate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


Output



### You can see that as expected the date created is exactly the same as written and represents 14th January 2014. Example 4 - How to check if two dates are equal in Java 8

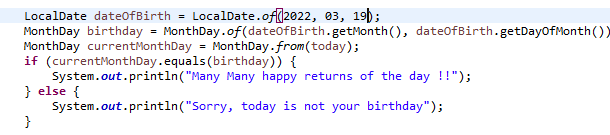
Talking about real world date time tasks, one of them is to check whether two dates are same or not. Many times you would like to check whether today is that special day, your birthday, anniversary or a trading holiday or not. Sometimes, you will get an arbitrary date and you need to check against certain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



Output



### 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 :  
  


Output



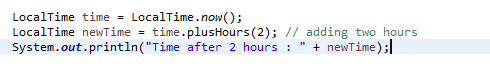
### Since today's date matches the birthday, irrespective of year you have seen the birthday greeting as output. You can run this program by advancing your windows date and time clock and see if it alerts you on your next birthday or not, or you can write a JUnit test with the date of your next year's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


Output



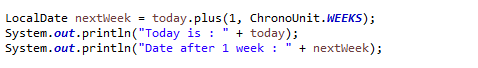
### Example 7 - How to add hours in time

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.  
  


Output



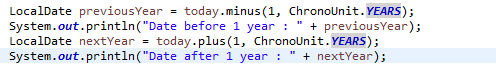
### Example 8 - How to find Date after 1 week

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.  
  
  


Output



### 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

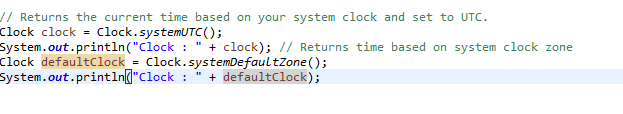
This is a continuation of the previous example. In the last example, we learn how to use plus() method of LocalDate to add days, weeks or months in a date, now we will learn how to use the minus() method to find what was the day before 1 year.  
  


Output



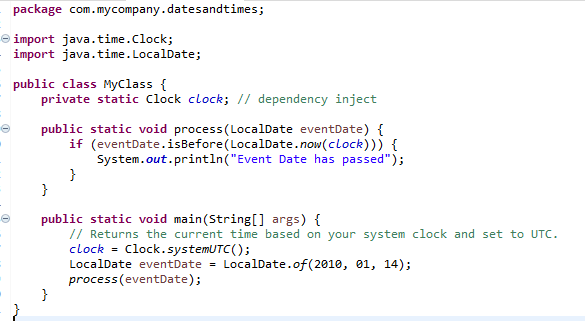
You can see that we now have two years, one is in 2013, and the other is in 2015, the year before and after the current year 2014.  
  
  
**Example 10 - Using Clock in Java 8**

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().

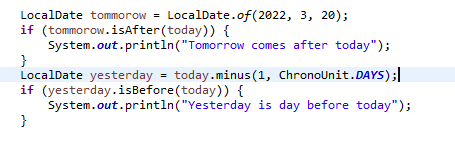


Output



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


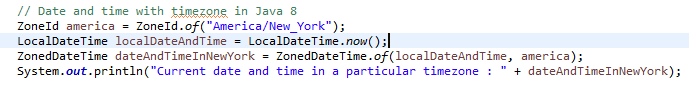
### This could be useful if you want to process dates on a different time zone. Example 11 - How to see if a date is before or after another date in Java

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.  
  


Output



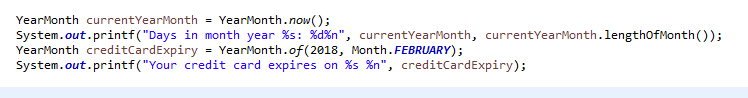
### You can see how easy it is to compare dates in Java 8. You don't need to use another class like Calendar to perform such essential tasks. Example 12 - Dealing with time zones in Java 8

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 :  
  


Output



### 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

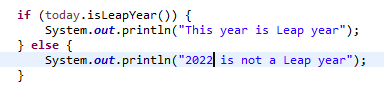
Like our MonthDay example for checking recurring events, YearMonth is another combination class to represent things like credit card expires, FD maturity date, Futures or options expiry dates etc. You can also use this class to find how many days are in the current month, lengthOfMonth() returns a number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


Output



### 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.

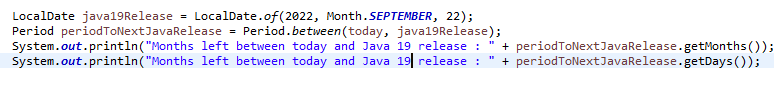


Output



### 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.

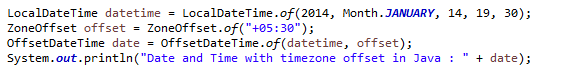


Output



### 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.



Output



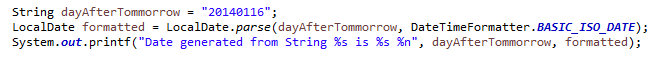
### You can see the timezone attached to date and time now. BTW, OffSetDateTime is meant for machines for human dates prefer ZoneDateTime class. Example 17 - How to get current timestamp in Java 8

The instant class has a static factory method now() which return current timestamp, as shown below :  
  


Output



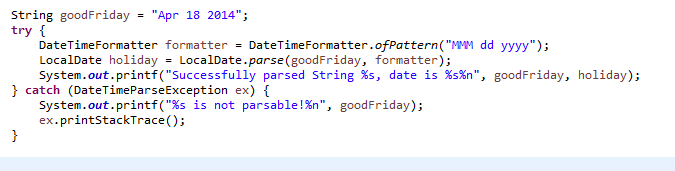
### You can see that current timestamp has both date and time component, much like java.util.Date,  in fact, Instant is your equivalent class of pre-Java 8 Date and you can convert between Date and Instant using respective conversion method added in both of these classes e.g. Date.from(Instant) is used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


Output



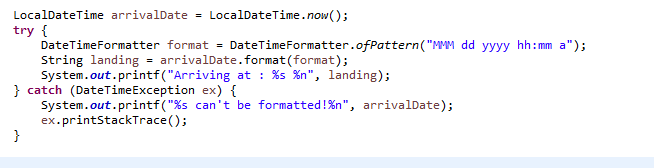
### Example 19 - How to parse the date in Java using custom formatting

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.  


Output



### 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.  
  


Output



\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*