# CIS 3145 Class Notes: Text Chapter 15

## Dates / Time Concepts

**Objectives**

* Create and use GregorianCalendar / Date objects to represent time
* Create and use a Date object
* Use the DateFormat class to format dates and times

**Date / Time Processing: Part A – Prior to Java 8**

Working with dates is a multistep process. First a **GregorianCalendar** object is created. This represents the date. There are limited methods that can be applied to this **GregorianCalendar** object.

Next a **Date** object is created from the **GregorianCalendar** object. This **Date** object can then be used in calculations because it can return the date as represented by the milliseconds since Jan 1st, 1970. Also the **DateFormat** class can use a **Date** object, but NOT a **GregorianCalendar** object to format a date. The **DateFormat** is just like the **NumberFormat** used in previous chapters.

The **GregorianCalendar** class is in the java.util.GregorianCalendar package.

Special fields used with the **GregorianCalendar** are in the java.util.Calendar class.

The **Date** class is in the java.util.Date class.

The **DateFormat** class is in the java.text.DateFormat class.

GregorianCalendar Object 🡪 Date Object 🡪 DateFormat Object applied to a Date Object

1. Create the **Gregorian** **Calendar** objects

GregorianCalendar now = new GregorianCalendar ();

GregorianCalendar startDay = new GregorianCalendar (2011,0,30);

//Note that months are zero based and range from 0 to 11

1. Create a **Date objects** based on the **Date** class or the **Gregorian Calendar** object

Date nowDateObject = new Date ();

Date startDateObject = startDay.getTime(); //getTime returns a Date object

1. Format the **Date object** with the **format**() method of the DateFormat class

DateFormat formattingObject = DateFormat.getDateTimeInstance();

String nowString = formattingObject.format(nowDateObject);

String StartDateString = formattingObject.format(startDateObject);

**Date / Time Processing: Part B – Java 8**

The new way to process time is with the **java.time** API package. With this system objects are created to represent a date, a time, or a combined date/time. The API also has its own DateTimeFormatter class for controlling how the objects are represented as strings. For date/time processing, there are three main steps we need to follow: **creating an object** to represent the time (input), **manipulating** **the time object** (processing), and **creating a formatted string version** of the time (output). The java.time package is said to have ‘**high cohesion**’ because all of the classes and methods needed to create the time variables, manipulate them and create the formatted output are in a single place and the package does not do any other types of processing.

The java.time API uses three classes to represent time only (**LocalTime**), days (**LocalDate**), or a detailed day and time (**LocalDateTime**).

All three classes use one of three methods (‘**now**’, ‘**of**’, or ‘**parse**’) to create an object.

1. The **now()** method creates the object based on the system clock.

LocalDateTime nowDateTime = LocalDateTime.now();

1. The **of()** method creates the object based on integer parameters for the year, month, day, minute, second, and nanosecond. Month can be based on a Month enumeration.

LocalTime **storeOpeningTime** = LocalTime.of(9,0,0); // hour, minute, sec.

LocalDate **birthDayDate** = LocalDate.of(2017, Month.MAY, 30);

1. The **parse()** method creates the object using a string as the argument for the method.

LocalTime **storeClosingTime** = LocalTime.of("21:30:00"); // 9:30:00 PM

LocalDate **schoolStartDate** = LocalDate.of("2017-08-21");

Once an object is created various get methods can be used to get the individual parts of the object such as the year, month, day hour, minute or second value of the object. There is also a getDayOfWeek() method that will determine if the date is a Monday through Sunday, and a getDayOfYear() method that returns the ordinal position of the day as an integer.

Comparison methods of the object take a second object as an argument to compare the object’s value. The **isBefore**(), **isAfter**(), **isEqual**(), and **compareTo**() methods return a Boolean and can be used as conditions in if statements and loops.

LocalDate currentDate = LocalDate.now();

LocalDate dueDate = LocalDate.of(2017, Month.MAY, 20);

if (currentDate.isAfter(dueDate))

System.out.println("The Book is late");

A series of ‘**with’** methods will create a new time object based on the current object but this one element changed. For example, the following code creates a new object that is a year later that the current object using the **withYear()** method.

LocalDate currentDate = LocalDate.now();

int oneYearLater = currentDate.getYear() + 1;

LocalDate futureDate = currentDate.**withYear**(oneYearLater);

The **plus**() and **minus**() methods take two **arguments**, a **long** and an enumerated **chronological unit** (ChronoUnit), to create a new object that is after or before the current date.

LocalDate currentDate = LocalDate.now();

LocalDate dueDate = currentDate.plus(1,ChronoUnit.WEEKS);

System.out.println("The book is due " + dueDate.toString());

Alternate versions of the **plus**() and **minus**() methods are the **plusDays**(), **plusWeeks**(), **plusMonths**(), and **plusYears**() methods and corresponding minus versions, which only need a single long argument.

The **LocalDate** object has a **toEpochDay**() method that returns the number of days since January 1st, 1970, as a long value. This allows us to calculate the elapsed time between two dates. The number of days since the epoch day (Jan. 1st, 1970) is calculated for each day and the difference is the elapsed time.

long currentDay = currentDate.toEpochDay();

long dueDay = dueDate.toEpochDay();

long elapsedDays = dueDay - currentDay;

System.out.println("The book is due in " + elapsedDays + " days.");

The **LocalTime** object has a .toSecondOfDay() method that returns the number of seconds since the start of the day, as an integer.

The **LocalDateTime** object has a .toInstant(zoneOffset.UTC).getEpochSecond() method that returns the number of seconds since Jan. 1st, 1970, as a long data type.

The **DateTimeFormatter** class is similar to the NumberFormat class introduced in chapter 3. First an object must be created, then this object uses the format() method to create a string version of the time object.

* The **ofLocalizedDate()** method creates an object that formats **LocalDate** objects.
* The **ofLocalizedTime()** method creates an object that formats **LocalTime** objects.
* The **ofLocalizedDateTime()** method creates an object that formats **LocalDateTime** objects.

Each method takes the input of an enumerated formatting style. **Dates** can have a SHORT, MEDIUM, LONG, or FULL style, while **times** have only the SHORT or MEDIUM styles.

DateTimeFormatter dtf = DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL);

String currentString = dtf.format(currentDate);

String futureString = dtf.format(futureDate);

System.out.println("The current Date is " + currentString);

System.out.println("The future Date is " + futureString);

It is important to use these formatting methods to create formatted output. One reason is that it is easier to change the formatting style by simply changing the enumeration in the “ofLocalized” method. Another reason is that when the program is used in countries which represent times and dates in different formats these methods will automatically generate the appropriate formatting. This is considered a **best practice** and is better than writing your own code to generate the required formatting.