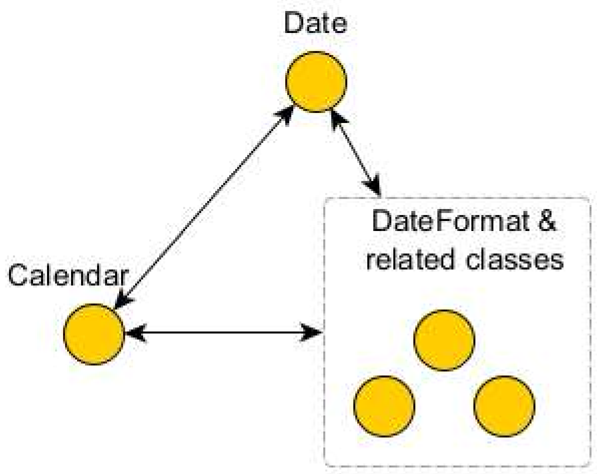
Lecture 18 — Date and Time in Java

Time and date APIs seem to have been a last-minute addition to Java 1; they were not well-designed and seem to mimic the Unix time C library from 1970. Java 8 finally replaced the old API, but many years of code use the original APIs so you need to know these as well. Let's start with the old API (in `java.util`).

Old (pre Java 8) API: Date, Calendar, and DateTimeFormatter

These classes share the following relationships:



Objects of the **`Date`** class in the **`Java.util`** package (`Calendar` is in that package too) can be used to represent specific instant in time, with millisecond precision (take that with a grain of salt!), and has methods to convert and compare `Date`s.

```
Date now = new Date();  // The current date/time
new Date(long);  // Date constructor
```

The second form constructs a `Date` from a *timestamp*, which is a `long` representing the number of milliseconds since Midnight 1/1/1970 GMT. (See also "**`System.currentTimeMillis()`**".) Displaying a `Date` as a string ("`System.out.println(now);`") shows the date in the local time zone if possible. (Note 1 second equals 1,000 milliseconds.)

> Unix-based systems stored time as the number of seconds, in a 32-bit int. That will overflow in 2038 (Monday, January 18 at 10:14:07 PM EST). Like the "Y2K" problem, this one is the Y2K38 problem. It will affect older hardware (such as some bank mainframes and GPS satellites).
>
> Although Java uses a `long` and not an `int` to store timestamps, the `Date` class uses `System.currentTimeMillis` which is platform-dependent; On a 32-bit (virtual) machine the problem still exists for Java. Additionally, many system clocks only update the time to the nearest 10 or so milliseconds.

You can time events (in milliseconds) by subtracting two `Date` timestamps:

```
      long duration = end.getTime() - start.getTime();
```

However, that may not be very (or at all) accurate for durations of less than several seconds. See also **`System.nanoTime()`**, the best way to time durations. (Note 1 second equals 1,000,000,000 (a billion) nanoseconds.)

The timestamp returned by `System.nanoTime` is just a `long` that starts counting from some arbitrary point in time. You cannot use this to determine the real "wall-clock" time. The starting point is set by the JVM when it starts and can vary between JVMs.

`System.nanoTime` is useful for timing durations within a single running JVM. Don't use `System.currentTimeMillis` for that, as it is the best approximation to "wall-clock" time (and known as the "real time clock") and can leap forward or backward by tens of milliseconds. (Technically speaking, `System.nanoTime` is a *monotonic* clock and `System.currentTimeMillis` is not. The `Date` class is based on the latter.)

When timing or comparing timestamps between JVMs (and between networked computers in general), the best you can currently use is the "real time clock" (`Date` or `System.currentTimeMillis`). Don't expect that to have accuracy better than (many) tens of milliseconds.

While not subject to the Y2K38 problem, Both `System.currentTimeMillis and System.nanoTime` use the resolution of the system's underlying clock. (The precision is accurate to a nanosecond but the clock may report the same value without updating (*resolution*) for the same tens of milliseconds as for currentTimeMillis.)

The `Date` class doesn't contain any methods to fetch the parts of a date, such as the year or day or month. Instead these tasks are given to a **`Calendar`** class. This is an abstract class, to support many different calendaring systems. However, in most of the world (the part that uses computers anyway) the calendar system used is *Gregorian*, and there is a `GregorianCalendar` ***concrete*** (that is, not abstract) class.

"Prior to JDK 1.1, the class `Date` had additional functions. It allowed the interpretation of dates as year, month, day, hour, minute, and second values. It also allowed the formatting and parsing of date strings. Unfortunately, the API for these functions was not amenable to internationalization. As of JDK 1.1, the `Calendar` class should be used to convert between dates and time fields and the **`DateFormat`** class (in the `java.text` package) should be used to format and parse date strings. The corresponding methods in `Date` are deprecated."

Java since version 8 includes a much better set of classes for dates, times, and intervals, based on the open source Joda time project. (See JSR-310, "Date and Time API".)

"Although the class `Date` is intended to reflect ***Coordinated Universal Time*** **(UTC)**, it may not do so exactly, depending on the host environment of the JVM."

In all methods of class `Date` (and `Calendar`; most `Date` methods are deprecated!) that accept or return year, month, date, hours, minutes, and seconds values, the following representations are used (documented *only* in `Date`, not `Calendar`):

- A *year* y is represented by the integer y – 1900 (not in `Calendar`).
- A *month* is represented by an integer form 0 to 11; 0 is January, 1 is February, and so forth; thus 11 is December. (True for `Calendar` too.)
- A *date* (*day of month*) is represented by an integer from 1 to 31.
- An *hour* is represented by an integer from 0 to 23. Thus, the hour from midnight to 1 a.m. is hour 0, and the hour from noon to 1 p.m. is hour 12.
- A *minute* is represented by an integer from 0 to 59 in the usual manner.
- A *second* is represented by an integer from 0 to 61; the values 60 and 61 occur only for leap seconds and even then only in Java implementations that actually track leap seconds correctly. Because of the manner in which leap seconds are currently introduced, it is extremely unlikely that two leap seconds will occur in the same minute, but this specification follows the date and time conventions for ISO C.

In all cases, arguments given to methods for these purposes need not fall within the indicated ranges; for example, a date may be specified as January 32 and is interpreted as meaning February 1.

To format a date for the current Locale, use one of the static factory methods (formatting classes are in the `java.text` package):

```
myString=DateFormat.getDateInstance().format(myDate);
```

```
myString=DateFormat.getDateInstance(style).format(myDate);
```

You can use a `DateFormat` to parse also:

```
     DateFormat df = ...;
    myDate = df.parse( myString );
```

Use **`getDateInstance`** to get the normal date format for that country. Other static factory methods: **`getTimeInstance`** to get the time format for that country, and **`getDateTimeInstance`** to get a date and time format. You can pass in different options to these factory methods to control the length of the result; `SHORT`, `MEDIUM`, `LONG`, or `FULL`. The result depends on the *locale*, but generally:

> **SHORT** is completely numeric, such as *12.13.52 or 3:30 PM*
> **MEDIUM** is longer, such as *Jan 12, 1952*
> **LONG** is longer, such as *January 12, 1952 or 3:30:32 PM*
> **FULL** is pretty completely specified, such as
> *Tuesday, April 12, 1952 AD or 3:30:42 PM PST.*

Note: Don't confuse these classes with those in `java.sql` package!

`java.util.`**`SimpleDateFormat`** **class allows you to specify an arbitrary pattern for formatting dates**. A number of formats are used, such as for email (RFC-5322), and most other uses such as log files (ISO-8601, and the related RFC-3339 used for HTTP dates). Here's how you can format a `Date` as an RFC-5322 (standard date format, formally known as RFC-822) string:

```
SimpleDateFormat fmt = new SimpleDateFormat(
    "EEE, dd MMM yyyy HH:mm:ss Z (zzz)" );
System.out.print( fmt.format(new Date() ) );
```

(Show **DateTime.java**, **CalendarDemo.java**.)

`java.time` — New API since Java 8

The old Calendar and Date classes suffer from many short-comings. They fail to use the international standard for working with dates and times, ISO-8601 (see also this ISO-8601 notation summary). They don't use standard Unicode Common Locale Data Repository (Unicode CLDR) for locale data. They don't use the standard time zone database (iana.org TZDB). The old API didn't have units of time, durations, etc. The `Date` class was mutable. The new API handles all that and more.

The new API has a large number of classes, in five packages. But don't let that convince you it is more complex than the old classes; it isn't. The most used packages are

- `java.time` — contains the most used classes.
- `java.time.format` — contains the formatting and parsing classes.
- `java.time.temporal` — classes to convert dates and times, and to make adjustments to them (such as adding 30 days to today).

The other two packages are `java.time.chrono` (used for non-standard calendars) and `java.time.zone` (contains classes for defining time zone rules).

All the classes in these packages are immutable, so there are no "set" methods in any class. They do use a common method naming scheme, to make working with so many classes and methods easier. See the chart on the Java Tutorial's Method Naming Conventions page. Most of these method prefixes are obvious: "`of`" creates new objects from its arguments, which are validated. The "`from`" methods create new objects by converting from other objects (a `ZonedDate` from a `LocalDate`, etc.) (Note such methods may need to "fake" missing data, or drop extra data, for example when converting `LocalDateTime` to/from `LocalDate`). Some examples:

```
LocalDate date = LocalDate.of( 2015, Month.JULY, 4 );
LocalDateTime dt = LocalDateTime.from(Instant.now());
LocalDateTime dt = LocalDateTime.now();
```

The "`to`" methods convert to other types (similar to "from"), "`get`" methods returns a part of a date-time object, "`parse`" and "`format`" methods convert from/to Strings, "`is`" methods query an object (example "`isLeapYear`"), and "`plus`" and "`minus`" methods add/subtract time. (In addition to the `format` methods of these classes, you can use `String.format` to convert date and time objects into Strings.)

The two method names (prefixes really) that weren't obvious to me were "`with`" methods (returns a new object with one part changed) and "`at`" methods, which combine date/time objects together. (*Show API of some classes*.)

The most common classes include

- **`LocalDate`** — Objects represent a date, but without any time or time zone. (There are also classes to represent years, months, days of months, etc.)
- **`LocalTime`** — Objects represent time, but without any date or time zone data.
- **`LocalDateTime`** — Obvious. Since most applications use a single (local) time zone in the application, these three classes are the most commonly used.
- **`ZoneId`** — Objects represent time zones, and have methods to convert UTC time to/from local time.
- **`ZonedDateTime`** — In essence, a combination of `LocalDateTime` and `ZoneID`. Use when you need to mix dates and times from different time zones, or to allow users to select a zone other than the local zone (useful for web applications, where the server is often in a different time zone from the clients).
- **`Instant`** — Similar to the old `Date` class, objects represent a specific nanosecond in time, using Midnight, January 1st 1970 GMT (UTC) as the zero point. Unlike the `Date` class (and the `System.currentTimeMillis` method), All `Instant`s are monotonic since 1972. Whenever you need to work with timestamps or durations, use this class. An example:

  ```
  Instant later = Instant.now().plusHours(1);
  ```

  Monotonic clocks never are adjusted backwards. Instead they lengthen or shorten seconds (and milliseconds) until the clock catches up.

  For very accurate timing of durations, a non-standard clock must be used. In the future, it is expected that more OS versions and computer hardware will provide such a clock.

  For now, be aware that time is an unsolved problem when using multiple JVMs (each has their own notion of time) and of network time in general. Clocks used from inside running virtual machines may appear to leap ahead when the VM is paused, which happens frequently.

  Only use monotonic clocks for timing durations, and even then, be aware the time may leap forward by tens of milliseconds or may adjust timestamps by ±0.1% (to make the clock better match Solar time).

  For timing between JVMs (especially across a network), the best you can do is the “real time” clock that is not monotonic, and don’t expect an accuracy better than many tens of milliseconds.

- **`DateTimeFormatter`** — Provides standard format and parse methods for the new date and time classes. In addition, the old formatting methods (including `printf`) work on the new objects as well:

  ```
  System.out.printf( "%tl:%1$tM %1$Tp%n",
     LocalDateTime.now() );  // Output: 3:06 PM
  ```

- **`TemporalAdjuster`** — Provides numerous static methods to use with the various “`with`” methods of the other classes. Some examples (assume appropriate static imports):

```
LocalDate date = LocalDate.now();
LocalDate first = date.with( firstDayOfMonth() );
LocalDate monday = date.with( firstInMonth(MONDAY) );
```

(“`MONDAY`” is a constant of the class `DayOfWeek`.) You can also create custom adjusters.

- **`Duration`** — Used most often with the `Instant` class, specifies a number of nanoseconds. You can use these with `plus` and `minus` methods. Note, the duration is independent of time zones or other calendar changes.
- **`Period`** — Similar to `Duration` but uses days, months, and years to represent the duration of time; adding a day takes into account (for example) daylight savings time switches.

See the Java Tutorial Date Time Trail for more information, as well as the Java docs for the API. (*Show JavaTimeDemo*.)