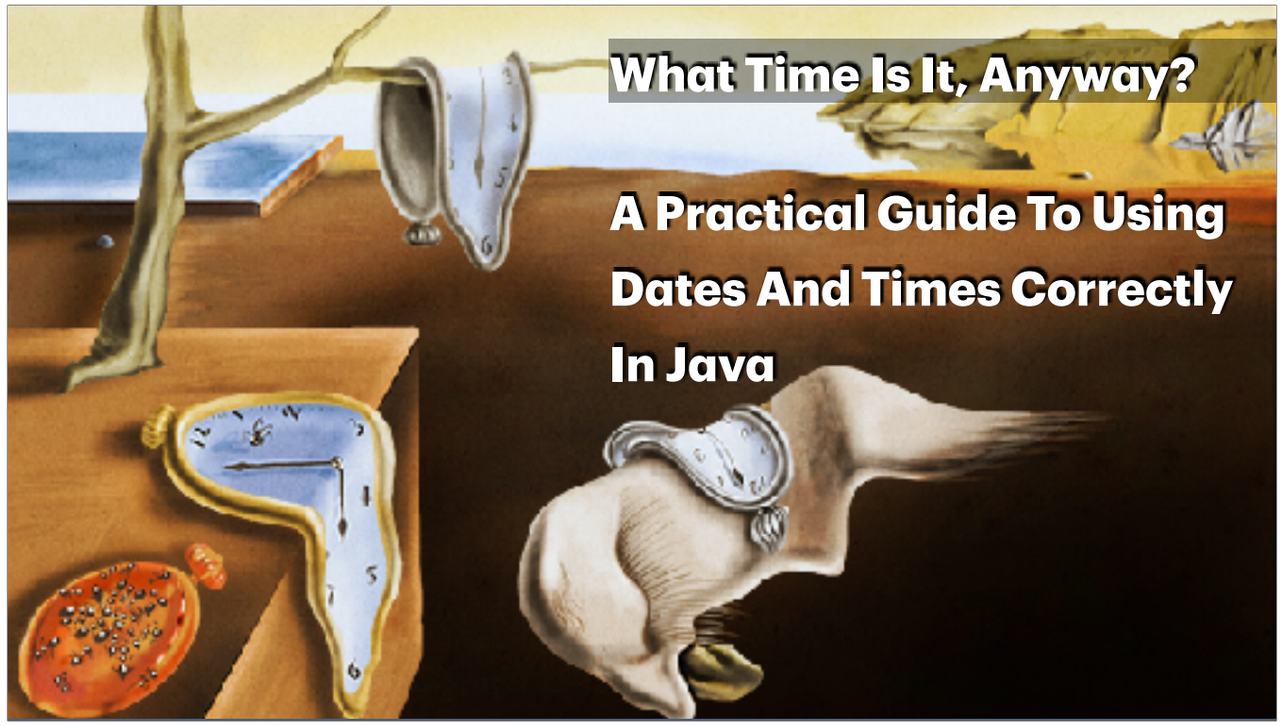


Good afternoon, and welcome to my talk “What Time Is It, Anyway? A Practical Guide To Using Dates And Times Correctly In Java”.

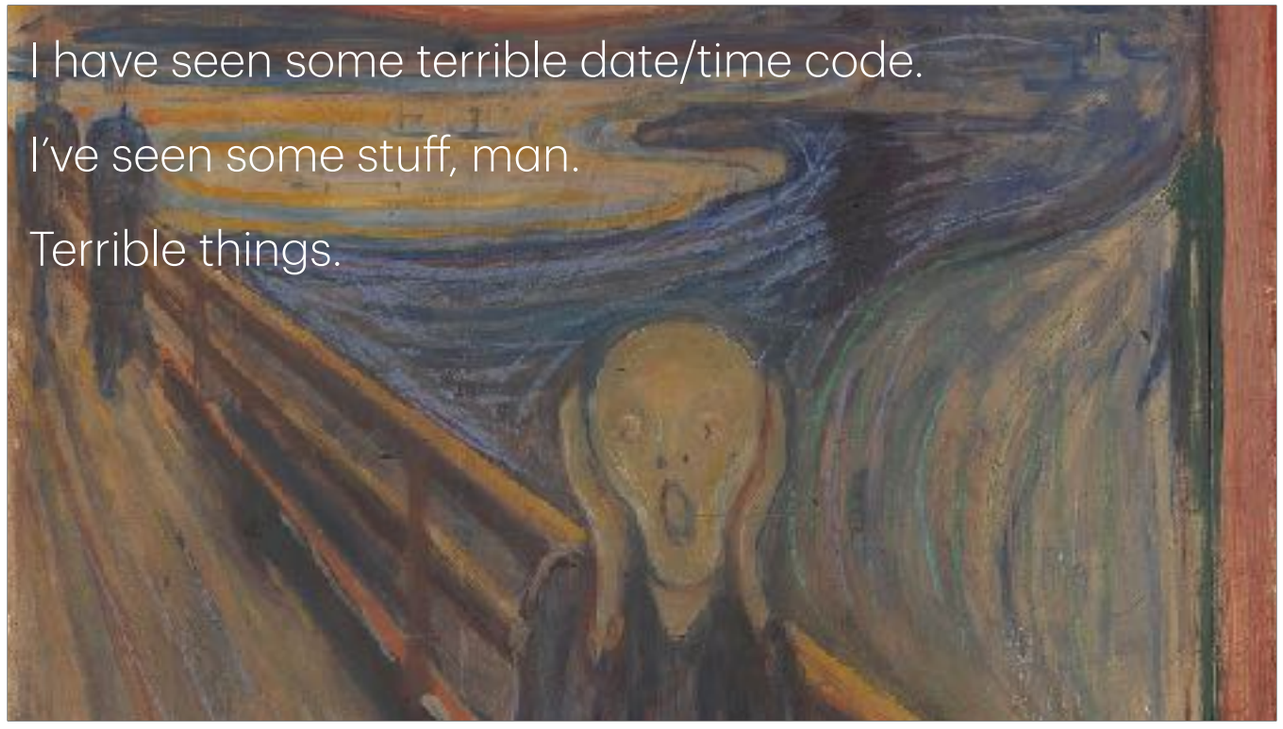
Today's Agenda

- Introduction
- A Brief History Of Time
- UTC - Coordinated Universal Time
- ISO-8601
- Why early Java date/time needed work
- JodaTime
- The Java 8 Date and Time APIs
- ZonedDateTime, LocalDateTime, And More!
- Some Examples
- Suggestions for unit testing
- ThreeTen
- Resources

Since there are so many great talks happening simultaneously at Devnexus, here's what I'll be covering here today. If you're still interested, great! If not, feel free to switch to a different talk - you won't hurt my feelings. Although, please leave a couple of dollars at the door to pay my therapist to treat my abandonment issues. :) Just kidding!

Here’s a little background on myself. My name is Kelly Morrison, and I’m a consultant and solutions architect for Daugherty Business Solutions. I have a bachelor’s degree in mathematics from the University of Georgia (Go Dawgs!) and a doctorate in computer engineering from Auburn University (War Eagle!). I’ve worked for many companies over the years, from start-ups to large, internationally known firms. I’m currently consulting on a project at Edward Jones.

You’re probably wondering why I’m covering a topic like the Java date and time API in the year 2024. Good question.



The reason is that, over the years, the date and time functions in Java are the ones that are least understood and most frequently abused by programmers. I have seen some truly horrendous date and time code in my career. Let me begin my talk by showing you three of the many date time debacles I’ve encountered.

Example #1: “I don’t know Date(), but I know regex!”

The first one came from a programmer who was actually a fairly good coder (he was an expert at regular expressions), but he didn’t understand the original Java Date functions.

```java
import java.util.*;
import java.util.regex.*;

public class Main {

    public static void main(String[] args) {
        Date currentDate = new Date();
        System.out.println("Current date = " + currentDate);

        String timePattern = "(\\d{2}:\\d{2}:\\d{2})"; // Pattern for hours:minutes:seconds
        Pattern pattern = Pattern.compile(timePattern);

        Matcher matcher = pattern.matcher(currentDate.toString());

        if (matcher.find()) {
            String time = matcher.group(1);
            System.out.println("Current time: " + time);
        } else {
            System.out.println("Could not get the time.");
        }
    }
}
```

```
Current date = Wed Apr 10 22:49:08 EDT 2024
Current time: 22:49:08
```

His task was to get the current time, in “hours:minutes:seconds” format. He started out correctly, by creating a new Date object. But then his solution went off the rails. He didn’t know that Java had a SimpleDateFormatter for formatting dates and times. So he converted the Date to a string using the toString() method, then wrote a regular expression to parse out the time, and applied that regular expression to the string. That was an interesting pull request review when he checked that in. :)

Example #2: “Formatting? I’ll do it myself.”

Another programmer at a different company also had the same issue: she didn’t know about the Java date/time formatter.

```java
import java.util.*;

public class Bad2 {

    public static void main(String[] args) {
        Date currentDate = new Date();

        String currentTime = String.format("%02d:%02d:%02d", currentDate.getHours(), currentDate.getMinutes(), currentDate.getSeconds());

        System.out.println("Current time: " + currentTime);
    }
}
```

```
Current time: 22:59:29
```

So, she formatted the date and time herself by pulling the appropriate fields from the Date object. It works, but it’s fairly clunky, when there was a built-in and tested solution available.

Example #3: The Day The Unit Tests Died

- I was working with a team on a project that scheduled appointments.
- One day, the unit tests started failing. Everywhere. Even on machines where nothing had changed.
- The culprit was in the unit tests themselves.
- The tests were checking to see if an appointment was in the summer...

The last example I'll show you caused me some grief on New Year's Day. On January 1, several of our unit tests started failing (and since they were run as part of our build pipeline, it was preventing the app from being deployed). And the tests were failing everywhere, across multiple code branches and environments. They were failing on the build machine, one developer machines, QA machines… it was bizarre. And I got called in early to look at it.

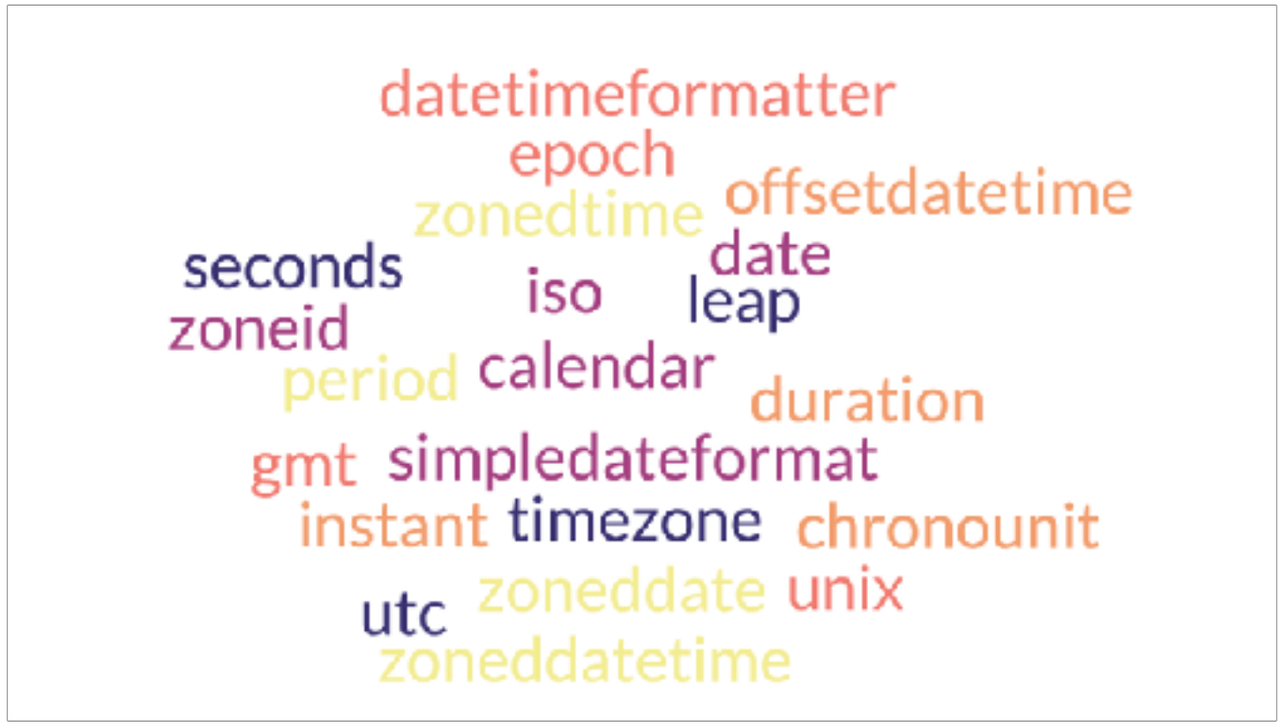
```
Date firstDayOfSummer = new Date();
firstDayOfSummer.setMonth(5);
firstDayOfSummer.setDate(1);
firstDayOfSummer.setDate(2012);

Date today = new Date();

boolean isNotYetSummer = today.before(firstDayOfSummer);
```

The culprit turned out to be some badly written test cases. There was a function that was created to determine whether summer had started yet (in the summer, interns would become available to the project, so it affected the appointment generation we were doing). In the test cases, someone had hard coded the current year (2012) into the test cases as the beginning of summer. And you can see several issues right away with the Date class in Java. It's mutable (a date was declared and then modified three times to set the month, day, and year). The setMonth() function is 0-based, so you need to specify a 5 for "June" instead of the "6" that you would expect. But the biggest problem with the unit test is that the firstDayOfSummer was hard coded with the current year, but the code that gets the current day was dynamic. So, the code worked properly all through 2012, but on the first day of 2013, it failed, because it was comparing the current date with the start of summer from the PREVIOUS year.

So, why are intelligent, otherwise reliable programmers having so much trouble with dates and times in Java?

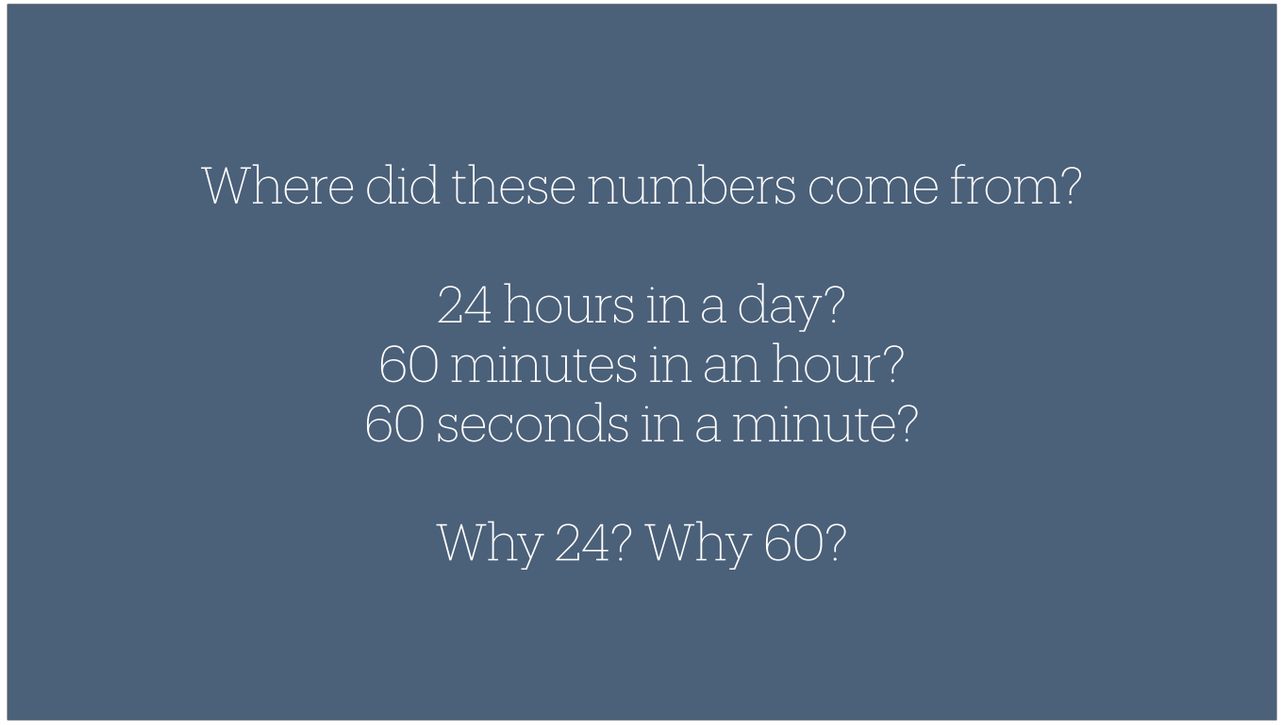
Well… it *is* confusing.

Well, to be honest, the APIs are rather confusing.



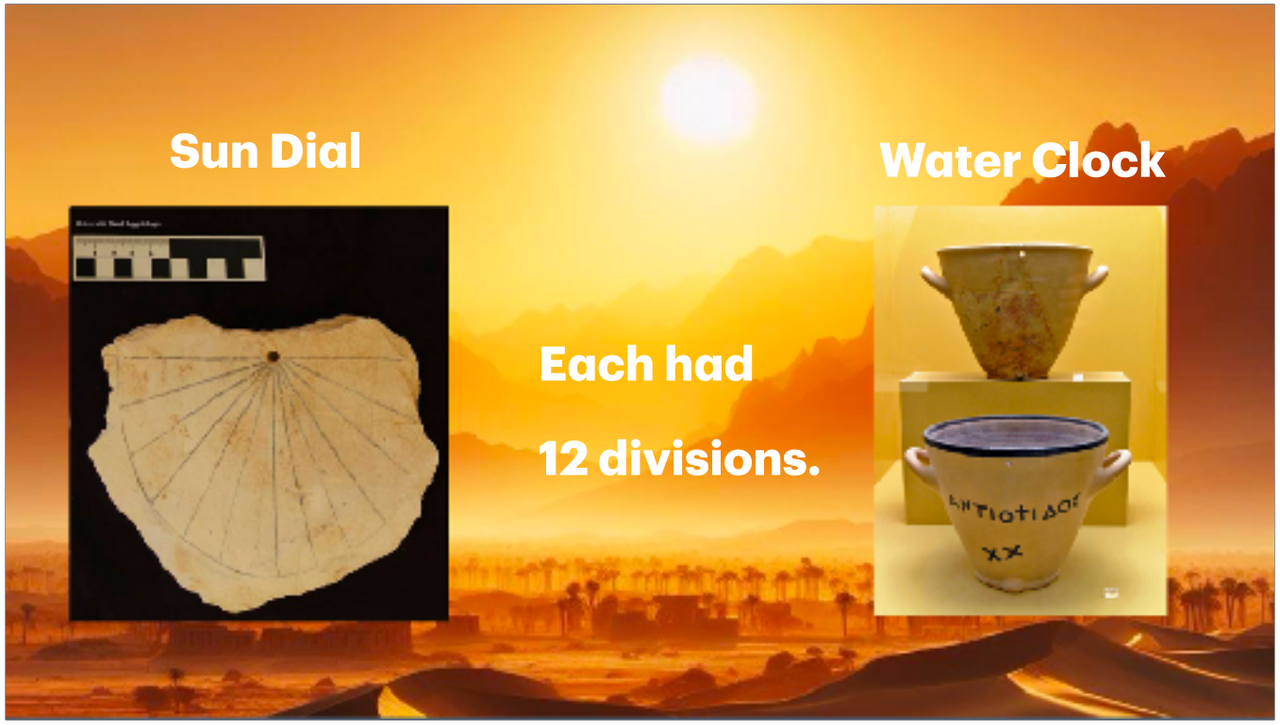
Here’s a word cloud showing some of the terms you come across when looking at the Java date and time functions. It has a lot of terms that are probably incomprehensible to someone who hasn’t been properly trained. ZoneId? GMT? UTC? Chronounit? Epoch? Why isn’t this easier to understand? You would think you would just have some simple “get me the current time and date” functions.

So, let’s look at a little history to explain where some of these terms came from… and why they’re relevant to you when working with times and dates. I’ll try to be a little less annoying than “Miss Minutes” from Disney’s “Loki” miniseries.

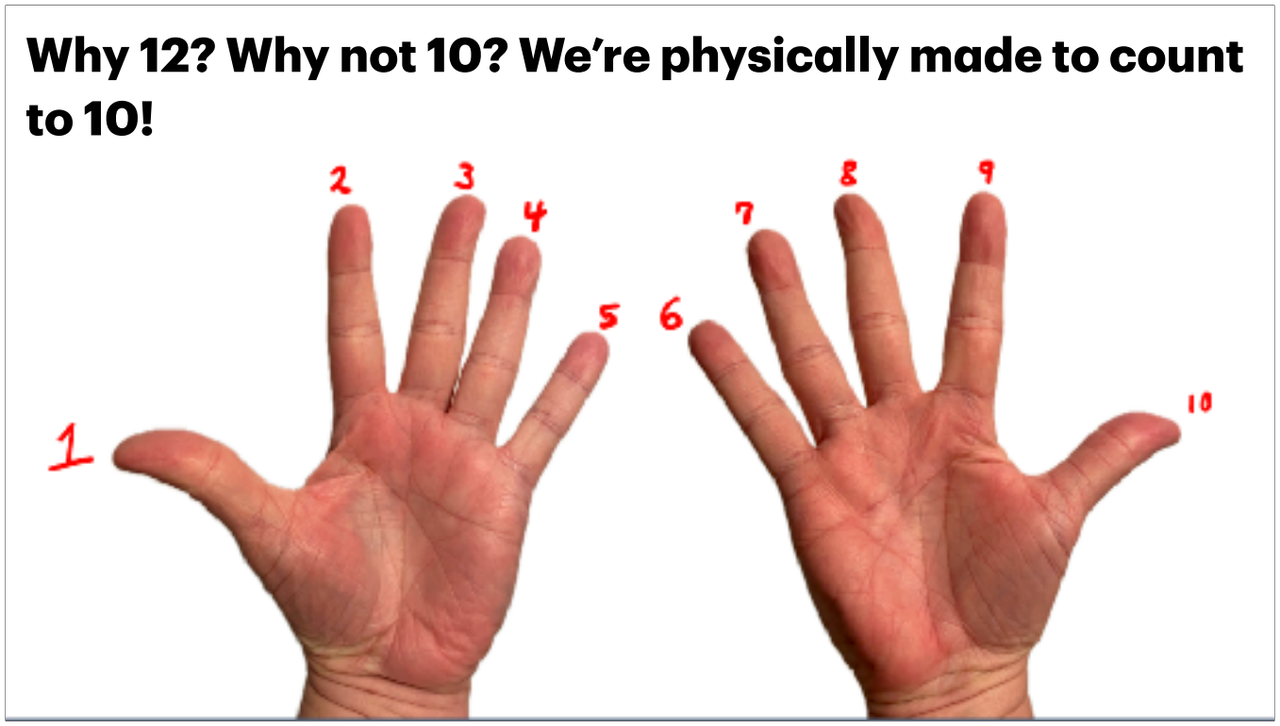


Have you ever wondered about some of the numbers associated with time? Why did the day get sliced into 24 hours? Who decided that there were 60 minutes in an hour, or 60 seconds in a minute? Where did those numbers come from?

We begin in ancient Egypt, around 1500 BC. The Egyptians had had a concept of days and times for hundreds of years, but it was around 1500 BC that they began to be serious about being precise in their measurements of days and times.

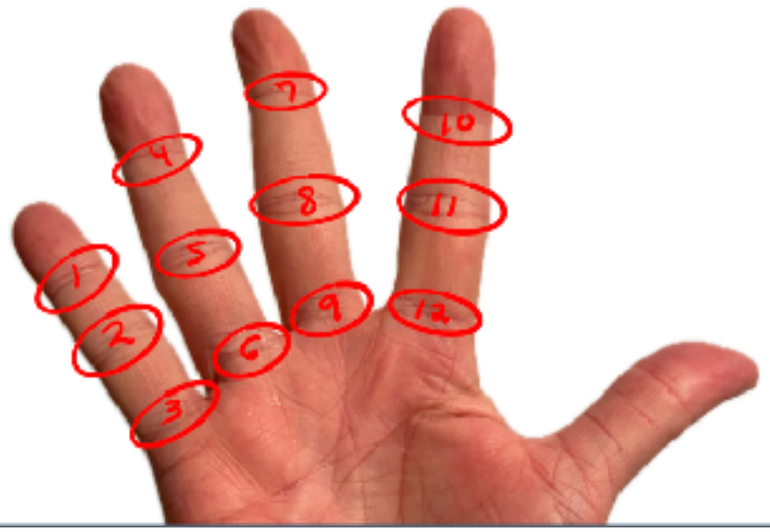


Egypt used two inventions to keep track of the time. The sun dial was used in the daytime: it used a vertical rod or divider that cast a shadow on a flat plate. As the sun moved across the sky, its shadow would move around the dial. The Egyptians made 12 equal divisions on the dial to break the day into sections. For tracking times at night, they used a “water clock”. This was a pot that was filled with water at sunset that would slowly drain throughout the night. The interior of the pot was marked with 12 lines, breaking the night into twelve divisions. But why 12?

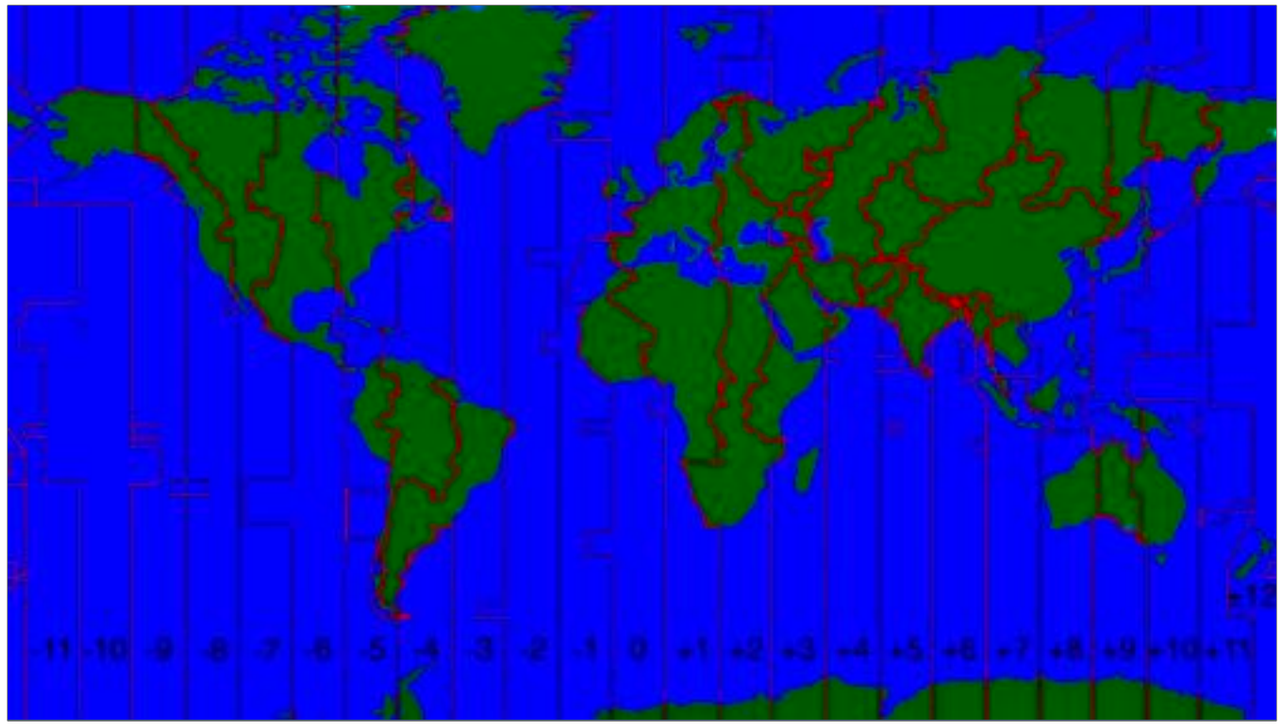


You would think that the Egyptians would have split the day and night into 10 divisions, not 12. After all, we're physically made to count to 10 because we have ten fingers (unless you're a polydactyl or had a really bad experience in high school shop class). So why 12 instead of 10?

There is some evidence that Egyptians counted to 12 on each hand by counting *joints*, not *fingers*.



Well, there is some evidence that the Egyptians counted to 12 on each hand by counting joints instead of fingers. You have four fingers with three joints each, which means you can count to 12 (if you use your thumb to keep track of the current joint).



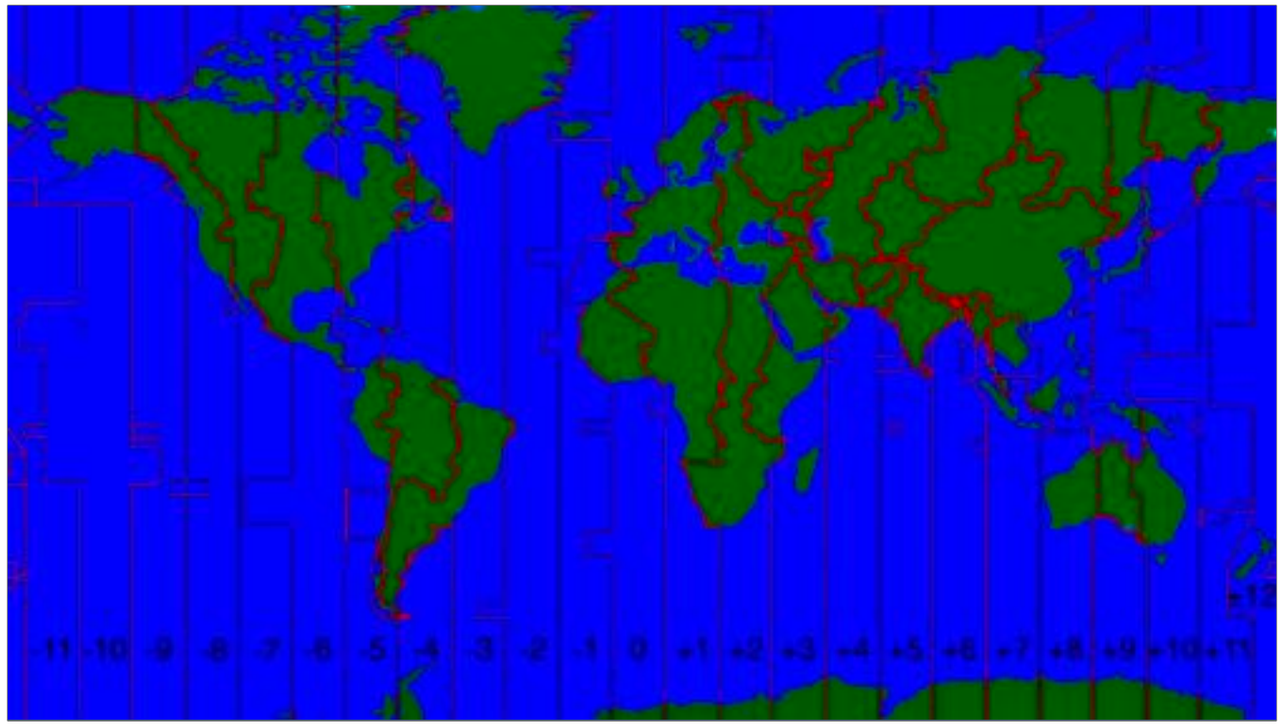
So, the Egyptians broke both the day and the night into 12 divisions each, for a total of 24. And today, we have 24 time zones around the world, each encompassing an hour.

Babylon (Iran) - 1000 AD

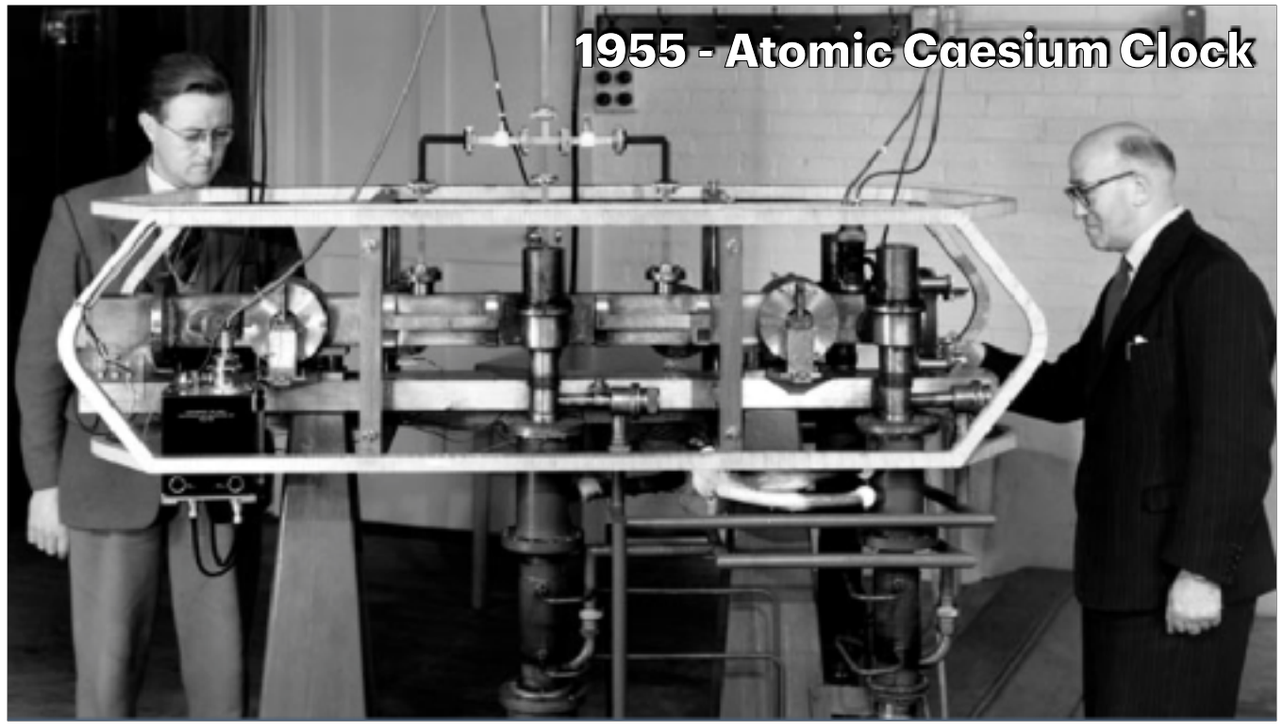
- They used base 60, not base 10.
- al-Biruni split hours into minutes, seconds, thirds, and fourths
- These came from the Latin terms:
  - *pars minuta prima* - “first small part”
  - *pars minuta secunda* - “second small part”
  - each part was 1/60th of the previous

Our next history stop is in Babylon around 1000 AD. The Babylonians were interesting: instead of using base 10 or 12, they used base 60. 60 is an interesting number: it’s divisible by 1, 2, 3, 4, 5, 6, 10, 12, and 15, so it’s a good number to use if you want flexibility in accurately splitting it into smaller parts. And that’s why a Babylonian named al-Biruni used base 60 to split an hour into smaller parts. He split the hour into minutes, seconds, thirds, and fourths, where each division was 1/60th of the previous. An hour was 60 minutes, a minute was 60 seconds, a second was 60 thirds, and a third was 60 fourths. The terms came from Latin: “minute” came from “pars minute prima”, meaning “first small part”; “second” came from “pars minute secunda”, meaning second small part. We don’t use thirds and fourths, although there are still some areas of the world that do.

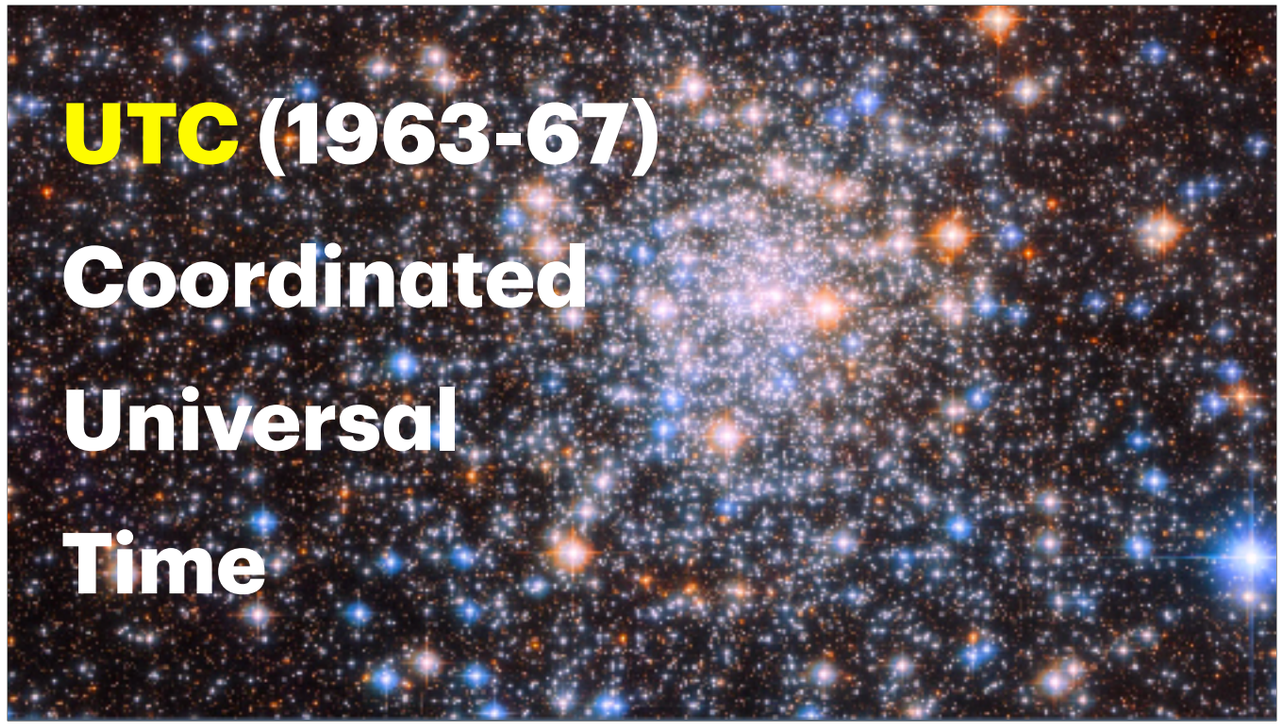
Next, we move to the UK and city of Greenwich, home of the Royal Observatory, in 1675. This is where the Greenwich Mean Time was established. Basically, if you want to figure out what time it is around the world, you need a starting place to count from. And England used the longitude (or meridian) that went through Greenwich. They called this the Prime Meridian. Other countries had their own prime meridians: France had a prime meridian that went through Paris (if you saw or read “The Da Vinci Code”, the “rose line” was their prime meridian). But eventually, England’s prime meridian won out.



If you look at this map that splits the globe into 24 pieces (or “time zones”), you can see that the one labeled with a 0 is the one that includes England’s prime meridian. The “mean” in “Greenwich Mean Time” means “average” - i.e., the prime meridian is in the center of that zone labeled 0. Each of the other zones is labeled by how many time zones they are away from GMT. The east coast of the US, for example, is 5 time zones behind England, while Sydney (Australia) is 10 time zones ahead of England. We also call these numbers “zone offsets”.



Moving on: in 1955, the first atomic clock (powered by cesium) was invented. This gave a precise way to measure the time.



Next, in 1963, UTC (Coordinated Universal Time) was created. This came about because a worldwide standard was needed for telling time instead of relying on a patchwork of different timekeeping systems. You’re probably wondering why it isn’t CUT instead of UTC, but the reason is that the creators wanted the abbreviation to be the same in all languages, and UTC would be used in more countries than CUT.

UTC

- Tries to track the atomic clock closely
- However it occasionally adds or subtracts “leap seconds” to try to make sure it stays synchronized with the Earth’s rotation
- This is because the Earth doesn’t rotate at a constant speed
  - It slows up or speeds down based on many factors (irregular shape, gravity interactions with the Sun and moon, etc.)
- Since we can’t predict leap seconds, it means calculating the difference between a UTC time now and in the future won’t be exact

UTC uses the atomic clock… but with occasional modifications. We like to think that the Earth rotates at a constant velocity, but in reality, it doesn’t. It tends to wobble, slowing up and slowly down. Why? Well, the earth isn’t a perfect sphere: it’s lumpy, and not well distributed (lots of water, mountain ranges, etc.). And it has gravity affecting it: both the Sun and the Moon exert gravitational pulls on it, with the Earth’s orbit around the Sun being elliptical and thus the pull changes during the year. The Earth also “sloshes”: the moon pulls the tides, constantly redistributing the weight; volcanos erupt; magma inside the core moves. In other words, although the Earth rotates at a mostly constant velocity, it can speed up or slow down, and there’s no way to predict it. Which means we can’t accurately predict times in the future. Meanwhile, UTC constantly adjusts the time by adding or subtracting small amounts of time called “leap seconds” to try to keep the constant time from the atomic clocks synchronized with the Earth’s actual observed time.

Epochs

- An epoch is a fixed point in time that computers use as a reference.
- The most famous is the “UNIX epoch”, which starts at January 1, 1970. Computers count the number of seconds that have elapsed since then.
- We may have a “Year 2038” problem when the number of seconds will overflow the 32-bit integer UNIX uses to store it.
  - Most UNIX systems are now 64-bit, which is good for 292 billion years.

With the computer era came yet another way of tracking time: the “epoch”. An epoch is a fixed point in time that computers use as a reference. They pick a “starting point” in time and then measure the number of seconds that have elapsed since then. The most famous one is the “UNIX epoch”, which counts the number of seconds elapsed since January 1, 1970 (a mostly arbitrarily picked date, chosen because it was a nice round time to use). Since UNIX used a 32 bit field to store the number of elapsed seconds, there is a potential “Year 2038” problem looming when the number of seconds elapsed will overflow the 32 bit field. Luckily, most UNIX systems are now 64-bit, but there may be issues with some older, legacy hardware. Also, different computer systems use different epochs, but the UNIX epoch is the most widely known.

ISO 8601 (1988)

- International standard for worldwide exchange of date and time data
- There isn't a single ISO 8601 "format": instead, it provides a description of how to carefully specify date and time formats

One last stop before we delve into code. In 1988, the ISO 8601 standard was developed to establish a means for worldwide exchange of date and time data. You would think it would be something as simple as a single format, such as "YYYY:MM:DD:HH:MM:SS", (for year, month, day, hour, minute, and second), but there's no single format. Instead, it describes a standardized way to specify a format. It's almost like a programming language for creating date and time formats. Let's look at some examples.

Some ISO 8601 examples

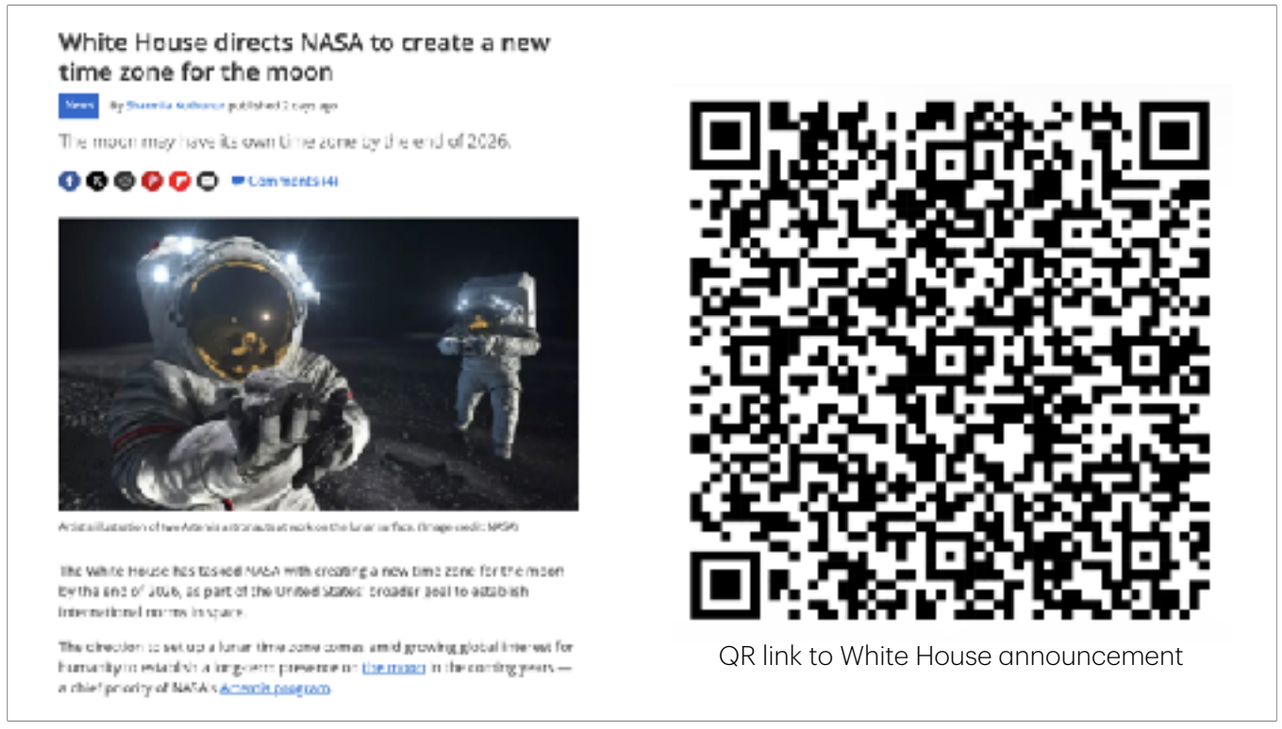
- Date only: `2024-04-10`
- Date and time separated by T: `2024-04-10T12:30:45`
- Date and time with timezone offset: `2024-04-10T12:30:45+02:00`
- Date and time with Zulu time (UTC): `2024-04-10T12:30:45Z`
- Date and time with fractional seconds: `2024-04-10T12:30:45.123`
- Date and time with fractional seconds and timezone: `2024-04-10T12:30:45.123+02:00`
- Date with week number: `2024-W15-3`
- Date with week number and day: `2024-W15-3T12:30:45`
- Ordinal date (day of the year): `2024-100`
- Ordinal date with time: `2024-100T12:30:45`
- Duration: `P3Y6M4DT12H30M5S` (3 years, 6 months, 4 days, 12 hours, 30 minutes, and 5 seconds)
- Duration with negative sign: `-P3Y6M4DT12H30M5S`
- Date range: `2024-04-10/2024-04-20`
- Date range with times: `2024-04-10T12:00:00/2024-04-20T18:00:00`
- Period starting from a date: `2024-04-10/P1Y` (1 year period starting from April 10, 2024)

Here are a number of valid ISO 8601 formats. As you can see, it can describe dates, times, combined dates and times, periods, and even date ranges. So, when someone says they're using "ISO 8601" as a date/time format for an application, you can see that that narrows it down somewhat, but you still need to inquire what the specific format is. Luckily, some of the more popular ones can be parsed automatically by Java.

This is the way I probably looked the first time I saw the ISO 8601 format.

One last stop before we look at codes. Remember when we said that UTC stood for “Coordinated Universal Time”? Well, as it turns out, they really meant “terrestrial” instead of “universal”. What time is it on the moon? It doesn’t fit into the timezones on the earth. It rotates at a different rate than the Earth. So, what time is it on the moon?

Surprise! There’s going to be a time standard for the moon: LTC, for Coordinated Lunar Time!



QR link to White House announcement

CLT was just announced earlier this month. The White House directed NASA to create a time zone for the moon, in preparation for the upcoming Artemis moon landings and the Gateway lunar space station. The QR link here will take you to the official three-page announcement from the White House. The target date will be the end of 2026, so perhaps we'll see LTC in Java several years down the road.

Enough history. Let's look at dates and times in Java!

For now, though, let's get back to looking at dates and times on the good ol' Earth.

Early Java (before version 8)

- Had the following classes for dates:
  - Date
  - Calendar
  - SimpleDateFormat
- Problems:
  - Not thread safe
  - The classes didn’t correspond to the domain very well

In the first versions of Java (before Java 8), there were several built-in classes for dealing with dates and times. The main class was Date, in the java.util package (yes, dates and times were considered as “utilities”). There was also a Calendar object, and a SimpleDateFormat class for formatting dates and times. Other classes existed, but these were the commonly used ones. But they had issues: they weren’t thread safe. They didn’t correspond to the domain very well (e.g., Date was both a date and a time).

The Date class was particularly bad

- No time zones
- Months are 0-indexed (this made it far too easy to introduce off-by-1 errors)
- Uses the local system timezone in too many places
- Doesn't correspond well to the java.sql.Date class
- It's not really a "date" - it's more of an "instant"
- Confusing methods: getDate() returns the day of the month; getDay() returns the day of the week. And it had some rough edges - you could specify February 1 as January 32 (!).

There were other issues with the Date class, as you can see from this list.

JodaTime

- Multiple calendar systems (Gregorian, Julian, etc.) with ISO 8601 as the default.
- Better API. Immutable classes, fluent API.
- Better performance.
- Less confusing to use.
- It is the de facto standard date and time library for Java versions earlier than 8.
- However, the designer (Stephen Colebourne) says it has some “design flaws”. It’s not “broken”, but it could have been designed better.

Because of the problems with the Java date classes, a standalone library called Joda Time was created. It was a major improvement: it was thread safe, with immutable classes. It had a cleaner, easier to use “fluent” API. It had better performance. And it quickly became the de facto standard for dates and times in versions of Java prior to 8. However, the designer said it still had some “design flaws”, meaning that, although it wasn’t “broken”, it could have been designed better.

The Java 8 Date And Time APIs

- Stephen Colebourne (Joda Time) was again a primary author
- He decided to address the design flaws in Joda Time by implementing a new API in Java itself.
- Java 8 is inspired by Joda Time, but with a better design.
- The APIs are in the java.time package.

So, the author of Joda Time (Stephen Colebourne) got a mulligan: he worked with Oracle to create the Java 8 date and time APIs. It wasn't a port of Joda Time: instead, it used Joda Time as an inspiration to rethink and redesign a better date/time API. And the new APIs were finally given their own package: java.time.

LocalDate

Let’s look at some of the classes in the new Java 8 date/time API. We’ll start with LocalDate.

```java
import java.time.*;

public class Java8Example1 {

    public static void print(final String header, final LocalDate localDate) {
        System.out.printf("%s = %s\n", header, localDate);
    }

    public static void main(final String[] args) {
        // --- LocalDate is just the date where your machine is running.
        print("Today", LocalDate.now());

        // --- You can add months, days, weeks, etc. And you can chain them.
        print("Next month", LocalDate.now().plusMonths(1));
        print("90 days from now", LocalDate.now().plusDays(90));
        print("A year and a day from now", LocalDate.now().plusYears(1).plusDays(1));

        // --- You can subtract values, too!
        print("Yesterday", LocalDate.now().minusDays(1));
        print("A year ago today", LocalDate.now().minusYears(1));

        // --- You can define a date, or use the predefined epoch.
        print("The Bicentennial", LocalDate.of(1976, 7, 4));
        print("The UNIX epoch", LocalDate.EPOCH);
        print("Min date", LocalDate.MIN);
        print("Max date", LocalDate.MAX);
    }
}
```

```
Today = 2024-04-11
Next month = 2024-05-11
90 days from now = 2024-07-10
A year and a day from now = 2025-04-12
Yesterday = 2024-04-10
A year ago today = 2023-04-11
The Bicentennial = 1976-07-04
The UNIX epoch = 1970-01-01
Min date = -999999999-01-01
Max date = +999999999-12-31
```

LocalDate tells the date on the machine you're using. For example, my laptop is set to the Eastern time zone (I'm from Georgia), so it tells me the current date in Georgia. It's easy to use: you just call LocalDate.now() to get the date. You can manipulate it by adding or subtracting months, days, weeks, etc. There are a few predefined dates, such as the minimum and maximum dates, and the UNIX epoch (January 1, 1970). Finally, you can define your own dates by using LocalDate.of(year, month, day), where month is 1-based instead of 0-based (as in the original Java Date class).

```java
// -- Leap years are handled.
LocalDate feb282023 = LocalDate.of(2023, 2, 28);
print("One day after Feb 28 in 2023", feb282023.plusDays(1));

LocalDate feb282024 = LocalDate.of(2024, 2, 28);
print("One day after Feb 29 in 2024", feb282024.plusDays(1));
```

```
One day after Feb 28 in 2023 = 2023-03-01
One day after Feb 29 in 2024 = 2024-02-29
```

LocalDate also handles leap years. If you start with February 28 and add 1 day, you can see that it gives you March 1 in 2023, but February 29 in 2024 (a leap year).

LocalTime

That takes care of dates. Now let’s look at the local time.

```java
import java.time.LocalTime;

public class LocalTimeDemo {

    public static void print(final String header, final LocalTime localTime) {
        System.out.printf("%s = %s\n", header, localTime);
    }

    public static void main(final String[] args) {
        // ----- LocalTime is the time where you are at.
        print("Local time", LocalTime.now());

        // ----- There are some predefined times.
        print("Noon", LocalTime.NOON);
        print("Midnight", LocalTime.MIDNIGHT);
        print("Min time", LocalTime.MIN);
        print("Max time", LocalTime.MAX);

        // ----- You can add and subtract.
        print("5 minutes and 10 seconds from now", LocalTime.now().plusMinutes(5).plusSeconds(10));
        print("8 hours ago", LocalTime.now().minusHours(8));

        // ----- You can define one.
        print("Smoke'em time", LocalTime.of(4, 20));

    }
}
```

```
Local time = 02:45:21.540446
Noon = 12:00
Midnight = 00:00
Min time = 00:00
Max time = 23:59:59.999999999
5 minutes and 10 seconds from now = 02:50:31.557875
8 hours ago = 18:45:21.558155
Smoke'em time = 04:20
```

You can get the local time on your machine by calling LocalTime.now(). Again, there are some predefined ones: you can get noon, midnight, and the minimum and maximum times (the minimum is midnight, the maximum is just before midnight. And again, you can add or subtract minutes, seconds, etc., and define your own times by using LocalTime.of(…).

LocalDateTime

There’s also a LocalDateTime, which smooshes the LocalDate and LocalTime classes together.

```
// ----- The local date and time.
print("Now", LocalDateTime.now());

// ----- And you can manipulate it.
print("Yesterday", LocalDateTime.now().minusDays(1));
print("A week from now", LocalDateTime.now().plusWeeks(1));
print("3:27 from now", LocalDateTime.now().plusMinutes(3).plusSeconds(27));


Now = 2024-04-11T02:54:33.565676
Yesterday = 2024-04-10T02:54:33.571665
A week from now = 2024-04-18T02:54:33.571808
3:27 from now = 2024-04-11T02:58:00.571923
```

You can get the LocalDateTime by calling (you guessed it) LocalDateTime.now(). And again, you can manipulate it by adding or subtracting various numbers.

ZonedDateTime

Okay, we've seen how to look at the local time. But what about other times around the world? For that, we need ZonedDateTime. A ZonedDateTime has four parts: a date, a time, a ZoneOffset, and a ZoneId. The ZoneOffset tells how many timezones a location is away from the GMT timezone (remember the chart with the +1, +2, -1, -2 offsets?). The ZoneId identifies a particular time region inside a time zone. For example, there's an "American/New_York" ZoneId within the timezone that's 5 timezone offsets away from GMT. Why do we need both? Well, different areas around the world may have their own unique quirks about telling time. For example, the USA has Daylight Savings Time, which changes the time by an hour twice a year. So, just knowing the time zone offset isn't enough to tell the time: you need to know WHICH AREA inside the time zone so that you can apply any local quirks.

```
// ----- DateTimeFormatter lets us format a ZonedDateTime
DateTimeFormatter customFormatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss z");

// ----- ZonedDateTime is like LocalDateTime with a time zone
ZonedDateTime now = ZonedDateTime.now();
print("now                ", now.format(customFormatter));
print("ISO_ZONED_DATE_TIME", now.format(DateTimeFormatter.ISO_ZONED_DATE_TIME));
print("ISO_DATE_TIME      ", now.format(DateTimeFormatter.ISO_DATE_TIME));
print("ISO_WEEK_DATE      ", now.format(DateTimeFormatter.ISO_WEEK_DATE));
print("ISO_LOCAL_DATE_TIME", now.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));

print("default time zone", now.getZone());
print("zone offset", now.getOffset());
```

```
now                = 2024-04-11 03:31:10 EDT
ISO_ZONED_DATE_TIME = 2024-04-11T03:31:10.953531-04:00[America/New_York]
ISO_DATE_TIME      = 2024-04-11T03:31:10.953531-04:00[America/New_York]
ISO_WEEK_DATE      = 2024-W15-4-04:00
ISO_LOCAL_DATE_TIME = 2024-04-11T03:31:10.953531
default time zone = America/New_York
zone offset = -04:00
```

You can get a ZonedDateTime by calling ZonedDateTime.now(). This will give you the current date and time at your current location (your computer knows your time zone offset and zone id). There is also a DateTimeFormatter class that can be used to format a ZonedDateTime. Here, you can see I defined a custom format, as well as using several predefined formats. You can format a ZonedDateTime by calling the format() method and providing it with a formatter (these classes are all thread safe). Here you can see that my time zone (ZoneId) is “American/New_York”, and my zone offset is -04 (remember, we’re usually in the -05 zone offset, but Daylight Savings Time is currently in affect, changing the time by an hour). And again, this is why you need a zone id (such as “America/New_York”) so that the Java library can look up and apply any local quirks such as Daylight Savings Time.

Instants

Now, let’s look at instants. Remember when we said that computers keep time by counting seconds since a starting point (epoch)?

```
ZonedDateTime nyTime = ZonedDateTime.of(2024, 3, 17, 17, 30, 0, 0, ZoneId.of("America/New_York"));
ZonedDateTime laTime = ZonedDateTime.of(2024, 3, 17, 14, 30, 0, 0, ZoneId.of("America/Los_Angeles"));

print("New York Time      ", nyTime.format(customFormatter));
print("Los Angeles Time   ", laTime.format(customFormatter));

print("New York instant   ", nyTime.toInstant());
print("Los Angeles instant", laTime.toInstant());

print("UTC Time           ", nyTime.withZoneSameInstant(ZoneId.of("UTC")).format(customFormatter));

print("NY time as LA time ", nyTime.withZoneSameInstant(ZoneId.of("American/Los_Angeles")).format(customFormatter));
```

```
New York Time       = 2024-03-17 17:30:00 EDT
Los Angeles Time    = 2024-03-17 14:30:00 PDT

New York instant    = 2024-03-17T21:30:00Z
Los Angeles instant = 2024-03-17T21:30:00Z

UTC Time            = 2024-03-17 21:30:00 UTC
NY time as LA time  = 2024-03-17 14:30:00 PDT
```

An “Instant” in Java is the number of milliseconds (not seconds) since the UNIX epoch (January 1, 1970). And the instant is the same around the world. 3pm on the East Coast in the USA and noon on the West Coast in the USA have different times (3pm and noon), but they have the same instant. To demonstrate this, I’ve created two ZonedDateTimes, one for New York, and one for Los Angeles. You can see that I set the New York time to 5:30pm in the “America/New_York” zone, and the LA time to 2:30pm in the “American/Los_Angeles” zone. Although the times are different (5:30pm in New York is the same moment in time as 2:30pm in LA), the instants are the same. And if you look closely, you can see that both instants correspond to the current UTC (or GMT) time (9:30pm). So, an instant basically corresponds to the current date and time in Greenwich, and Java uses the ZoneId to figure out what the corresponding local time is in that time zone. Finally, you can see what time it is in a different timezone by using the .withZoneSameInstant(…other_time_zone…) method. This basically gives you the same instant in time as it looks in another time zone. In the last print statement here, you can see that I took the New York time and asked “what’s the equivalent time in Los Angeles?” by calling .withZoneSameInstant(…).

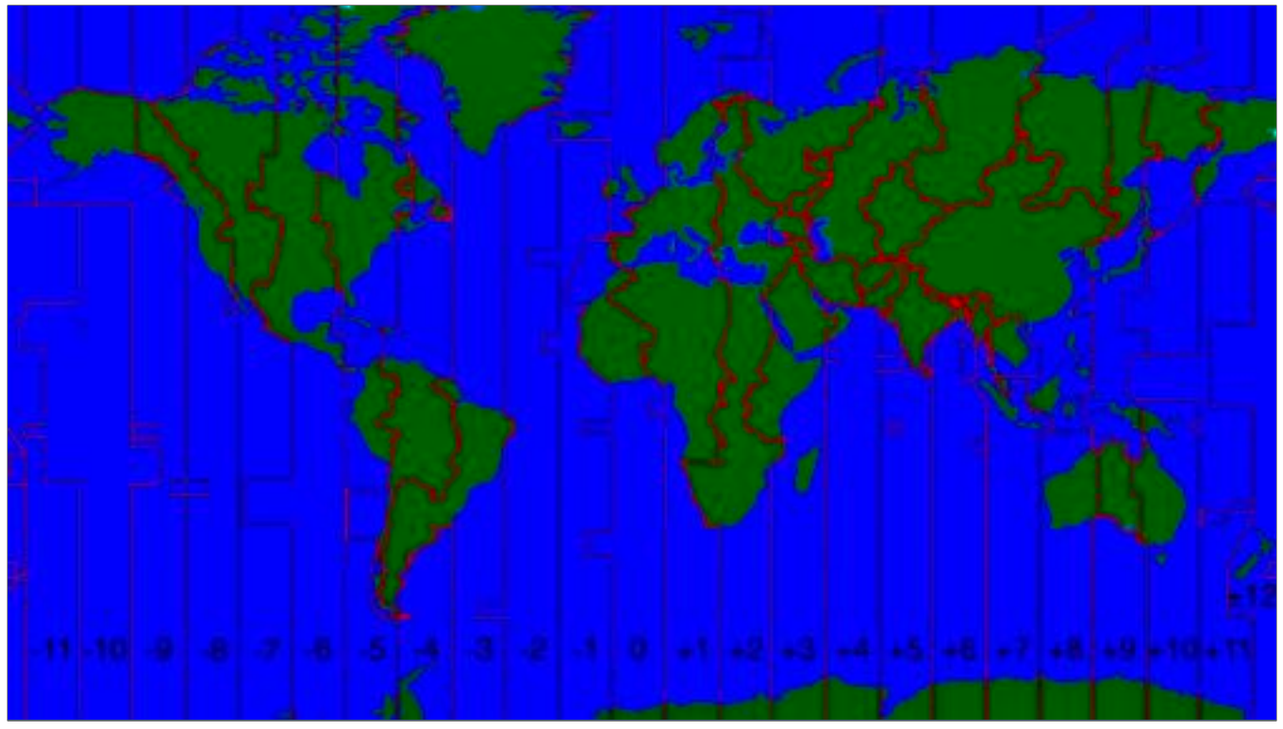
OffsetDateTime

Now, let’s look at OffsetDateTime.

```java
// A ZonedDateTime has a Date, a Time, a ZoneId, and a Zone offset.
ZonedDateTime now = ZonedDateTime.now();

// OffsetDateTime doesn't have a ZoneId.
// So, it can't handle Daylight Savings Time, etc.
OffsetDateTime nowAsOffset = now.toOffsetDateTime();

print("now         ", now);
print("now's offset", now.getOffset());
print("now's ZoneId", now.getZone());
print("nowAsOffset ", nowAsOffset);
```

```
now         = 2024-04-11T04:14:42.160005-04:00[America/New_York]
now's offset = -04:00
now's ZoneId = America/New_York

nowAsOffset  = 2024-04-11T04:14:42.160005-04:00
```

OffsetDateTime is similar to ZonedDateTime, but it doesn’t have a ZoneId. This means that it knows how many time zones you are away from GMT (e.g., +1, -3, etc.), but it doesn’t know what area in the time zone you’re in, so it can’t apply any local quirks such as Daylight Savings Time. You can get the OffsetDateTime from a ZonedDateTime by calling the .getOffset() method. This class can be useful in some instances, but for most cases, it’s better to use ZonedDateTime.



And again, remember that OffsetDateTime only knows the offset of the current location (+4, -1, etc.) It doesn’t know which area in that time zone is intended, so it can’t apply local quirks like Daylight Savings Time.

Periods and Durations

Let’s get back to some more useful classes, such as periods and durations.

```
ZonedDateTime start = ZonedDateTime.now();
ZonedDateTime end = start.plusMonths(3).plusDays(15).plusHours(12);

Period period = Period.between(start.toLocalDate(), end.toLocalDate());

print('start ', start);
print('end   ', end);
print('period', period);

Duration duration = Duration.between(start, end);
print('duration      ', duration);
print('duration days ', duration.toDaysPart());
print('duration hours', duration.toHoursPart());
print('total hours   ', (106*24)+12);
```

```
start  = 2024-04-11T04:25:38.579975-04:00[America/New_York]
end    = 2024-07-26T16:25:38.579975-04:00[America/New_York]
period = P3M15D
duration       = PT2556H
duration days  = 106
duration hours = 12
total hours = 2556
```

Both periods and durations are used to determine the difference between two dates and times. Period are meant for longer differences, such as days, weeks, months, years. Durations are intended for shorter differences, such as timing how fast a wide receiver can run a 50 yard dash. They are easy to call: just use Period.between(…) or Duration.between(…). You can access the individual parts of the answer (e.g., the number of days or hours) or you can print it as a string. For example, “P3M15D” stands for “Period of 3 Months, 15 Days”. “PT2556H” means 2556 hours.

Summary

- Use ZonedDateTime. It has a date, time, ZoneId, and ZoneOffset.
  - The ZoneId lets it deal with Daylight Savings Time, etc.
- You can use DateFormatter to format it to your preferred format.
- Period and Duration are useful for calculating the difference between two dates.
- ZonedDateTime's .withZoneSameInstant(...zoneID...) is useful for seeing what time it is right now in a different ZoneID. For example, if it's 9:03pm in Atlanta, Georgia, what time is it in London, England right now? Or Sydney, Australia?

So, to sum up, use ZonedDateTime. It has a date,, time, zone Id, and zone Offset. Because it has a ZoneId, it can apply local quirks such as Daylight Savings time. You can use DateFormatter to format ZonedDateTimes. Period and Duration are useful for calculating the difference between two ZonedDateTimes. And the .withZoneSameInstant(…) function will let you determine the date and time in a different timezone.

Unit test considerations

- Use ZonedDateTime.of(....) to construct specific date/times instead of using ZonedDateTime.now().
- You can write a function that gets the current ZonedDateTime so that it's easy to override or mock in a unit test.
- Or, you can write a TimeFactory to inject in your code that will let you mock dates and times for testing.

Now, how about unit testing? Remember the case I presented at the beginning where the unit tests broke on January 1? There are several ways to handle dates and times in your unit tests. For example, instead of relying on a dynamic date, use a fixed ZonedDateTime by constructing one using the ZonedDateTime.of(…) method. You can isolate the code that gets the current ZonedDateTime into a single function that you can override in unit tests. Or, you could be more elaborate and write a TimeFactory class that has a method to get the current date and time: you can then mock this class and inject it into your code for testing. We'll take a look at how to do the first case, isolating and overriding the ZonedDateTime retrieval.

```java
public class Test1Demo {

    // Hard to test
    public boolean areTaxesDueToday() {
        // Testing this function depends on what day you're running it.
        ZonedDateTime now = ZonedDateTime.now();
        return now.getMonthValue() == 4 && now.getDayOfMonth() == 15;
    }
```

```java
public class Test1Demo {

    // Easier to test
    2 overrides
    ZonedDateTime getNow() {
        return ZonedDateTime.now();
    }

    public boolean areTaxesDueToday() {
        ZonedDateTime now = getNow();
        return now.getMonthValue() == 4 && now.getDayOfMonth() == 15;
    }
```

Here we see a class called Test1Demo that has a method called areTaxesDueToday(). It gets the current date and time, and checks to see if it is April 15th. If so, it returns true; otherwise, it returns false. Obviously, this code will only return true if you run it on April 15th, so how can you test it? One way to do it is to first isolate the retrieval of the current date and time into a separate function, as seen in the second version, where we have a getNow() function to get the date/time.

```
    public static void main(final String[] args) {
        TestDemo demo1 = new TestDemo() {
            ZonedDateTime getNow() {
                return ZonedDateTime.of(2024, 03, 17, 0, 0, 0, 0, ZoneId.of("America/New_York"));
            }
        };

        TestDemo demo2 = new TestDemo() {
            ZonedDateTime getNow() {
                return ZonedDateTime.of(2024, 06, 15, 0, 0, 0, 0, ZoneId.of("America/New_York"));
            }
        };

        System.out.println("Are taxes due today? " + demo1.areTaxesDueToday());
        System.out.println("Are taxes due today? " + demo2.areTaxesDueToday());

    }
}
```

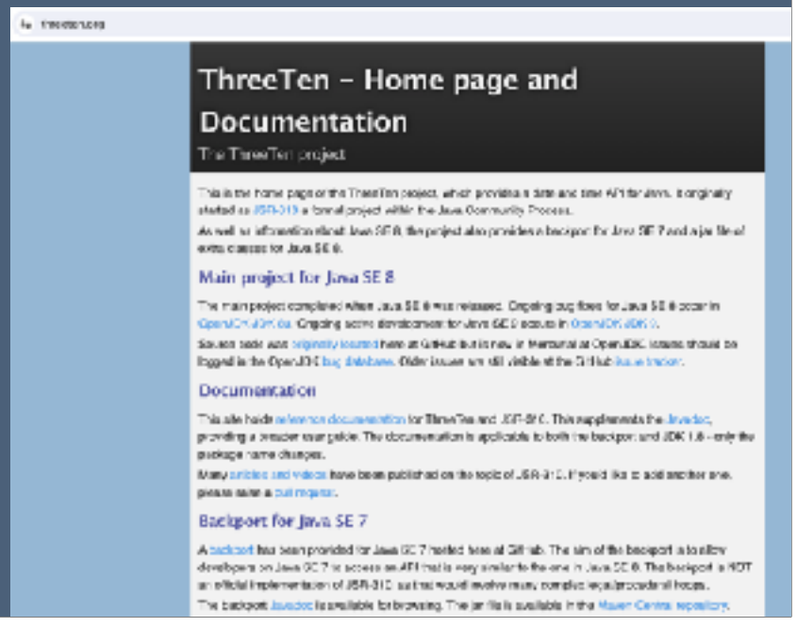
```
Are taxes due today? false
Are taxes due today? true
```

Now, you can override that method in your unit tests. Create an instance of the class, modify it to override the getNow() function, and use your own date/time. Now, you can test whether it’s April 15th for either case on any day.

ThreeTen

One last resource for you: you may want to look at ThreeTen.

ThreeTen - An incubator



ThreeTen was basically an “incubator” for the Java 8 date/time API. You can find early development, blog articles, etc., here. There is also a back port of the Java 8 date/time APIs for Java 7 here.

Thanks for attending my presentation! Again, my name is Kelly Morrison, and I work for Daugherty Business Solutions. If you're interested in working with us, please contact me or come see me. My personal and work emails are listed here, as well as a QR code that takes you to my LinkedIn profile. There is also a QR code that takes you to the slides (and a PDF) of this presentation on GitHub. Finally, as a reward for attending my talk, here's a picture of my adorable French bulldog, "Daisy Mae". Thanks again! I'll stick around for a while to answer any questions you may have.