Why is subtracting these two epoch-milli Times (in year 1927) giving a strange result?

Asked 13 years, 5 months ago Modified 10 days ago Viewed 813k times

7588

If I run the following program, which parses two date strings referencing times 1 second apart and compares them:

```
public static void main(String[] args) throws ParseException {
    SimpleDateFormat sf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
    String str3 = "1927-12-31 23:54:07";
    String str4 = "1927-12-31 23:54:08";
    Date sDt3 = sf.parse(str3);
    Date sDt4 = sf.parse(str4);
    long ld3 = sDt3.getTime() /1000;
    long ld4 = sDt4.getTime() /1000;
    System.out.println(ld4-ld3);
}
```

The output is:

```
353
```

Why is `ld4-ld3`, not `1` (as I would expect from the one-second difference in the times), but `353`?

If I change the dates to times 1 second later:

```
String str3 = "1927-12-31 23:54:08";
String str4 = "1927-12-31 23:54:09";
```

Then `ld4-ld3` will be `1`.

Java version:

```
java version "1.6.0_22"
Java(TM) SE Runtime Environment (build 1.6.0_22-b04)
Dynamic Code Evolution Client VM (build 0.2-b02-internal, 19.0-b04-internal,
mixed mode)
```

```
Timezone(`TimeZone.getDefault()`):

sun.util.calendar.ZoneInfo[id="Asia/Shanghai",
offset=28800000,dstSavings=0,
```

```
useDaylight=false,
transitions=19,
lastRule=null]

Locale(Locale.getDefault()): zh_CN
```

java date timezone

Share

Improve this question

Follow

edited May 20, 2023 at 18:11

cellepo

4,459 ● 4 ● 40 ● 65

asked Jul 27, 2011 at 8:15

Freewind

198k ● 163 ● 451 ● 731

133 The real answer is to always, always use seconds since an epoch for logging, like the Unix epoch, with 64 bit integer representation (signed, if you want to allow stamps before the epoch). Any real-world time system has some non-linear, non-monotonic behaviour like leap hours or daylight savings. – Phil H Jul 12, 2012 at 8:34

22 A great video about these kind of things: youtube.com/watch?v=-5wpm-gesOY – Thorbjørn Ravn Andersen Oct 14, 2014 at 10:39

6 And another from the same guy, @ThorbjørnRavnAndersen: youtube.com/watch?v=Uqjg8Kk1HXo (Leap seconds). (This one is from Tom Scott's own YouTube channel, not from Computerphile.) – TRiG Jul 3, 2015 at 16:23

6 @Phil H "seconds since the epoch" (i.e. Unix time) is non-linear as well, in the sense that POSIX seconds are not SI seconds and vary in length – Remember Monica Sep 13, 2020 at 23:57

5 I am amazed, nobody has wondered how this question even came to your mind? You already knew the answer and wanted to share with others? Or you read one of those "magical programming" articles which highlighted this surprising result. Otherwise who would fall upon two exact timestamps from 84 years ago, which could cause this. – Aqeel Ashiq Aug 23, 2022 at 3:43

11 Answers

Sorted by: Highest score (default)

11912

It's a time zone change on December 31st in Shanghai.

See this page for details of 1927 in Shanghai. Basically at midnight at the end of 1927, the clocks went back 5 minutes and 52 seconds. So "1927-12-31 23:54:08" actually happened twice, and it looks like Java is parsing it as the *later* possible instant for that local date/time - hence the difference.

Just another episode in the often weird and wonderful world of time zones.

+1450

If rebuilt with version 2013a of TZDB, The original question would no longer demonstrate quite the same behaviour. In 2013a, the result would be 358 seconds, with a transition time of 23:54:03 instead of 23:54:08.

I only noticed this because I'm collecting questions like this in Noda Time, in the form of unit tests... The test has now been changed, but it just goes to show - not even historical data is safe.

In TZDB 2014f, the time of the change has moved to 1900-12-31, and it's now a mere 343 second change (so the time between `t` and `t+1` is 344 seconds, if you see what I mean).

To answer a question around a transition at 1900... it looks like the Java time zone implementation treats *all* time zones as simply being in their standard time for any instant before the start of 1900 UTC:

```
import java.util.TimeZone;

public class Test {
    public static void main(String[] args) throws Exception {
        long startOf1900Utc = -2208988800000L;
        for (String id : TimeZone.getAvailableIDs()) {
            TimeZone zone = TimeZone.getTimeZone(id);
            if (zone.getRawOffset() != zone.getOffset(startOf1900Utc - 1)) {
                System.out.println(id);
            }
        }
    }
}
```

The code above produces no output on my Windows machine. So any time zone which has any offset other than its standard one at the start of 1900 will count that as a transition. TZDB itself has some data going back earlier than that, and doesn't rely on any idea of a "fixed" standard time (which is what `getRawOffset` assumes to be a valid concept) so other libraries needn't introduce this artificial transition.

Share

Improve this answer

Follow

edited Mar 27, 2023 at 19:45

VLAZ

28.8k ● 9 ● 62 ● 82

answered Jul 27, 2011 at 8:31

Jon Skeet

1.5m ● 889 ● 9.3k ● 9.3k

It doesn't say that thing about 5 minutes and 52 seconds anymore. – ADJenks Oct 24 at 23:28

You've encountered a local time discontinuity:

1747

> When local standard time was about to reach Sunday, 1. January 1928, 00:00:00 clocks were turned backward 0:05:52 hours to Saturday, 31. December 1927, 23:54:08 local standard time instead

This is not particularly strange and has happened pretty much everywhere at one time or another as timezones were switched or changed due to political or administrative actions.

Share Improve this answer Follow

answered Jul 27, 2011 at 8:38

Michael Borgwardt
346k ● 80 ● 486 ● 723

10 It happens twice a year anywhere that observes DST. – uckelman Nov 27, 2020 at 22:57

2 This one is not DST, I think. It's only 10 minutes back, and only once. At the same time, DST related changes can happen twice a year... or 4 times a year (due to Ramadan). or even once a year in some setups. No rule there :) – iwat0qs Mar 27, 2022 at 10:29

1 Generally it doesn't happen like this with DST these days, because the clock changes are done at 1am or 2am, to make sure that no change in date happens, and dates are not repeated. – Henry Brice Apr 10, 2023 at 17:18

750

The moral of this strangeness is:

- Use dates and times in UTC wherever possible.
- If you can not display a date or time in UTC, always indicate the time-zone.
- If you can not require an input date/time in UTC, require an explicitly indicated time-zone.

Share Improve this answer Follow

answered Jul 28, 2011 at 11:50

Raedwald
48.5k ● 47 ● 159 ● 245

17 None of these points would affect this result - it falls squarely under the third bullet point - and moreover, this is a time several decades before UTC was even defined, and thus can not really meaningfully be expressed in UTC. – Dag Ågren Nov 2, 2020 at 16:11

UTC does seem like the right solution for events *in the past*, but careful consideration should be given for events *in the future*, such as scheduling and appointments, where probably local time is more appropriate. ("Dentist at 11 a.m." will probably stay at local time 11 a.m. irrespective of shifts with respect to UTC.) – Pablo H Sep 17 at 12:33

420

When incrementing time you should convert back to UTC and then add or subtract. Use the local time only for display.

This way you will be able to walk through any periods where hours or minutes happen twice.

If you converted to UTC, add each second, and convert to local time for display. You would go through 11:54:08 p.m. LMT - 11:59:59 p.m. LMT and then 11:54:08 p.m. CST - 11:59:59 p.m. CST.

Share
Improve this answer
Follow

edited Nov 10, 2021 at 19:45
Aleksandr Dubinsky
23.2k ● 15 ● 85 ● 103

answered Jul 30, 2011 at 16:55
PatrickO
4,217 ● 1 ● 14 ● 3

349

Instead of converting each date, you can use the following code:

```
long difference = (sDt4.getTime() - sDt3.getTime()) / 1000;
System.out.println(difference);
```

And then see that the result is:

```
1
```

Share
Improve this answer
Follow

edited Dec 24, 2019 at 15:52
new Q Open Wid
2,284 ● 2 ● 20 ● 38

answered May 16, 2012 at 5:31
Rajshri
4,143 ● 2 ● 16 ● 17

268

I'm sorry to say, but the time discontinuity has moved a bit in

JDK 6 two years ago, and in JDK 7 just recently in update 25.

Lesson to learn: avoid non-UTC times at all costs, except maybe for display.

Share
Improve this answer
Follow

edited Dec 17, 2018 at 0:31

davnicwil
30.8k ● 21 ● 116 ● 134

answered Feb 17, 2014 at 22:44
user1050755
11.7k ● 4 ● 48 ● 56

2 MAYBE for display? I'm not sure I'd want to use your software otherwise, and I live in GMT which is close enough to UTC for half the year. – mjaggard Jun 23, 2022 at 15:51

@mjaggard GMT is at most 1s appart of UTC for the *entire* year ;) – Étienne Miret Jan 11, 2023 at 13:28

Sorry @ÉtienneMiret to be clear I meant "I live in GMT ... for half the year" – mjaggard Jan 17, 2023 at 11:34

237

As explained by others, there's a time discontinuity there. There are two possible timezone offsets for `1927-12-31 23:54:08` at `Asia/Shanghai`, but only one offset for `1927-12-31 23:54:07`. So, depending on which offset is used, there's either a one second difference or a 5 minutes and 53 seconds difference.

This slight shift of offsets, instead of the usual one-hour daylight savings (summer time) we are used to, obscures the problem a bit.

Note that the 2013a update of the timezone database moved this discontinuity a few seconds earlier, but the effect would still be observable.

The new `java.time` package on Java 8 let use see this more clearly, and provide tools to handle it. Given:

```
DateTimeFormatterBuilder dtfb = new DateTimeFormatterBuilder();
dtfb.append(DateTimeFormatter.ISO_LOCAL_DATE);
dtfb.appendLiteral(' ');
dtfb.append(DateTimeFormatter.ISO_LOCAL_TIME);
DateTimeFormatter dtf = dtfb.toFormatter();
ZoneId shanghai = ZoneId.of("Asia/Shanghai");

String str3 = "1927-12-31 23:54:07";
String str4 = "1927-12-31 23:54:08";

ZonedDateTime zdt3 = LocalDateTime.parse(str3, dtf).atZone(shanghai);
ZonedDateTime zdt4 = LocalDateTime.parse(str4, dtf).atZone(shanghai);

Duration durationAtEarlierOffset =
Duration.between(zdt3.withEarlierOffsetAtOverlap(),
zdt4.withEarlierOffsetAtOverlap());

Duration durationAtLaterOffset =
Duration.between(zdt3.withLaterOffsetAtOverlap(),
zdt4.withLaterOffsetAtOverlap());
```

Then `durationAtEarlierOffset` will be one second, while `durationAtLaterOffset` will be five minutes and 53 seconds.

Also, these two offsets are the same:

```
// Both have offsets +08:05:52
ZoneOffset zo3Earlier = zdt3.withEarlierOffsetAtOverlap().getOffset();
ZoneOffset zo3Later = zdt3.withLaterOffsetAtOverlap().getOffset();
```

But these two are different:

```
// +08:05:52
ZoneOffset zo4Earlier = zdt4.withEarlierOffsetAtOverlap().getOffset();

// +08:00
ZoneOffset zo4Later = zdt4.withLaterOffsetAtOverlap().getOffset();
```

You can see the same problem comparing `1927-12-31 23:59:59` with `1928-01-01 00:00:00`, though, in this case, it is the earlier offset that produces the longer divergence, and it is the earlier date that has two possible offsets.

Another way to approach this is to check whether there's a transition going on. We can do this like this:

```
// Null
ZoneOffsetTransition zot3 =
shanghai.getRules().getTransition(ld3.toLocalDateTime);

// An overlap transition
ZoneOffsetTransition zot4 =
shanghai.getRules().getTransition(ld3.toLocalDateTime);
```

You can check whether the transition is an overlap where there's more than one valid offset for that date/time or a gap where that date/time is not valid for that zone id - by using the `isOverlap()` and `isGap()` methods on `zot4`.

I hope this helps people handle this sort of issue once Java 8 becomes widely available, or to those using Java 7 who adopt the JSR 310 backport.

Share

Improve this answer

Follow

edited Dec 24, 2019 at 0:17

new Q Open Wid

2,284 ● 2 ● 20 ● 38

answered Jan 3, 2014 at 14:43

Daniel C. Sobral

297k ● 87 ● 505 ● 686

194

`IMHO` the pervasive, *implicit* localization in Java is its single largest design flaw. It may be intended for user interfaces, but frankly, who really uses Java for user interfaces today except for some IDEs where you can basically ignore localization because programmers aren't exactly the target audience for it. You can fix it (especially on Linux servers) by:

- export `LC_ALL=C TZ=UTC`
- set your system clock to UTC
- never use localized implementations unless absolutely necessary (ie for display only)

To the Java Community Process members I recommend:

- make localized methods, not the default, but require the user to explicitly request localization.
- use `UTF-8/UTC` as the *FIXED* default instead because that's simply the default today. There is no reason to do something else, except if you want to produce threads like this.

I mean, come on, aren't global static variables an anti-OO pattern? Nothing else is those pervasive defaults given by some rudimentary environment variables.....

Share
Improve this answer
Follow

edited Oct 20, 2020 at 3:40

Derek Wang
10.2k ● 4 ● 19 ● 40

answered Nov 26, 2014 at 15:58

user1050755
11.7k ● 4 ● 48 ● 56

42

As others said, it's a time change in 1927 in Shanghai.

It was `23:54:07` in Shanghai, in the local standard time, but then after 5 minutes and 52 seconds, it turned to the next day at `00:00:00`, and then local standard time changed back to `23:54:08`. So, that's why the difference between the two times is 343 seconds, not 1 second, as you would have expected.

The time can also mess up in other places like the US. The US has Daylight Saving Time. When the Daylight Saving Time starts the time goes forward 1 hour. But after a while, the Daylight Saving Time ends, and it goes backward 1 hour back to the standard time zone. So sometimes when comparing times in the US the difference is about `3600` seconds not 1 second.

But there is something different about these two-time changes. The latter changes continuously and the former was just a change. It didn't change back or change again by the same amount.

It's better to use UTC unless if needed to use non-UTC time like in display.

Share
Improve this answer
Follow

edited Nov 15, 2020 at 1:01

answered Feb 10, 2019 at 21:47

new Q Open Wid
2,284 ● 2 ● 20 ● 38

> UTC does seem like the right solution for events *in the past*, but careful consideration should be given for events *in the future*, such as scheduling and appointments, where probably local time is more appropriate. ("Dentist at 11 a.m." will probably stay at local time 11 a.m. irrespective of shifts with respect to UTC.) – Pablo H Sep 17 at 12:33

💡Extra information

0

It's because of daylight saving as others mentioned in their answer. I want to point out other facts about the date and calendar to show you:

🧠 Why you must NEVER assume the date

In addition to many existing calendars and rules like leap years and daylight saving, Throughout history, many changes have happened to the calendar that do not follow any order.

For example in 1752, the calendar used in England and its colonies was **11 days out-of-sync** with the Gregorian Calendar in use in most other parts of Europe. 👨‍🦲

So they changed it in a series of steps:

- December 31, 1750 was followed by January 1, 1750 (under the "Old Style" calendar, December was the 10th month and January the 11th)
- March 24, 1750 was followed by March 25, 1751 (March 25 was the first day of the "Old Style" year)
- December 31, 1751 was followed by January 1, 1752 (the switch from March 25 to January 1 as the first day of the year)
- September 2, 1752 was followed by September 14, 1752 (drop of 11 days to conform to the Gregorian calendar)
- More information here

Another example is Moon-based calendars like the Hijri date which can be changed every year depending on the moon phase in every month.

So there are even gaps in some calendars for more than 10 days and some inconsistency in some calendars from year to year! We should always pay attention to a lot of parameters when working with calendars, (like the timezone, locale, daylight saving, the calendar standard itself, etc.) and always try to use a tested and accurate date system.

⚠️ Preferred an online one, because some calendars like the Iranian Hijri Calendar are actively changing without the ability to forecast.

Share Improve this answer Follow

answered Nov 11, 2023 at 13:01

Mojtaba Hosseini

118k ● 43 ● 320 ● 310

It cannot ever be "1" as the result because **getTime() returns long milliseconds not seconds** (of which 353 milliseconds is a fair point but the epoch for Date is started at

-2

1970 not the 1920's). cmmnt: The API section you are using is largely considered deprecated. http://windsolarhybridaustralia.x10.mx/httpoutputtools-tomcat-java.html

Share Improve this answer Follow

answered Oct 23, 2021 at 6:39

Samuel Marchant
320 ● 2 ● 9

2 The original code is dividing the milliseconds by 1000 in order to convert to seconds. So the output certainly can be 1 (and is 1 in most time zones). – Anonymous Oct 23, 2021 at 8:57

Strange you should mention time zones as there are no "good" official time comparators before the atomic clock, only general agreements using astronomy. To relate any date (at absolute best) before the atomic clock to a java TZ (has current TZ database runtime updates for zones and calendar systems) is a farce. Using java TZ for time calc before runtime version database is a farce caused by there being no true zoning for accurate usage before the installed TZ database JRE file version. See historians and forensics for a TZ and custom zone implementations. – Samuel Marchant Oct 23, 2021 at 9:52

Highly active question. Earn 10 reputation (not counting the association bonus) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.