Java API for Time

Times and date and stuff

Example with a time interval

A customer wants you to implement a mechanism for representing a time interval for some product.

It should be possible to create an interval representing a start time and a stop time:

- 20:00:00 - 23:30:00
  - from 8 PM to 11:30 PM
- 22:00:00 - 05:30:00
  - from 10 PM to 5:30 AM

This could be used for instance to decide if camera monitoring should be on.

How would you implement that?

- You need to store the start and stop times
- You need some mechanism to check if a time is within the interval

Think for a minute about what types, classes, constructors and methods you'd need to implement such a time interval.

Think also what it means for a time to be "within" or included in an interval.

How would you check whether 4:30 AM is within the interval
22:00:00 - 04:00:00 ?

What types etc did you choose?

Did you decide to write a class Interval?

What type did you choose for start time and stop time?

What type did you choose for the check for if a time is included in the interval?

Example implementation

An object oriented approach could be to create a class as an abstraction for a time interval.

Let’s call the class Interval:

```
public class Interval {
  // what to put inside?
}
```

Let's use LocalTime from Java's java.time package

```
import java.time.LocalTime;

public class Interval {
  private LocalTime from;
  private LocalTIme to;

  public Interval(LocalTime from, LocalTime to) {
    this.from = from;
    this.to = to;
  }
  // Now what?
}
```

java.time.LocalTime

The java.time.LocalTime class represents a time without a time-zone in the ISO-8601 calendar system, such as 10:15:30. That seems perfect for our needs.

How do we create a LocalTime instance? We can use the parse factory method:

```
LocalTime someTime = LocalTime.parse("22:00:00");
```

Constructing an Interval instance

So, now we know how to construct a new Interval instance:

```
Interval night = new Interval(LocalTime.parse("22:00:00"),
                              LocalTime.parse("04:00:00"));
```

Implementing the check against a LocalTime

We want our Interval objects to be able to check if they encompasses some LocalTime, e.g. we could check if our night interval from the previous slide encompasses 01:30:00.

This sounds like a boolean test, so we could create an instance method of the Interval class like so:

```
public boolean encompasses(LocalTime time) { ... }
```

which enables client code to do:

```
if(night.encompasses(someTime) { ... }
```

Implementing the check against a LocalTime

How would you write the check to see if the time argument is encompassed (is a time inside of the interval) by the Interval?

You could start by writing some tests!

```
Interval night = new Interval(LocalTime.parse("22:00:00"),
                              LocalTime.parse("04:00:00"));
LocalTime hourOfTheWolf = LocalTime.parse("01:00:00");
assert night.encompasses(hourOfTheWolf) :
  "Expected" + night + " to encompass " + hourOfTheWolf;
```

Think for a while about what logic you need to check the above.

Implementing the check against a LocalTime

So how do we test if a time is encompassed by an Interval?

We have to consider two cases:

1. The interval is within the same 24 hours (both before midnight)
2. The interval wraps around midnight, e.g. 23:00:00 - 04:00:00

In the first case, it is enough to check that `from <= time <= to`.
In the second case, we have to check that
`from <= time || time <= to`

Can you see why?

Implementing the check against a LocalTime

Let the Interval have from = `22:00:00` and to = `04:00:00` . And let the time to check be `01:00:00` .

It is not enough to check that `from <= time <= to` because:

22:00:00 is not <= 01:00:00!

We have to check:

```
from <= time || time <= to

(22 <= 01 || 01 <= 04
  ( F     ||    T) == True!
```

Implementing the check against a LocalTime

For same-day-intervals, the check is much easier.

Let the Interval have from = `12:00:00` and to = `13:00:00` . And let the time to check be `14:00:00` .

It is now enough to check that `from <= time <= to`:

```
                          12 <=  14  <= 13
```

We simply check:

```
(from <= time && time <= to)
  (12 <= 14   && 14   <= 13)
    ( T       &&       F   )  == False!
```

Comparing to LocalTimes

LocalTime implements Comparable<LocalTime>, so it is simple to compare two LocalTime instances:

```
LocalTime.parse("09:15:00")
    .compareTo(LocalTime.parse("09:15:00") // 0
LocalTime.parse("09:15:00")
    .compareTo(LocalTime.parse("08:10:00") // positive
LocalTime.parse("09:15:00")
    .compareTo(LocalTime.parse("10:30:00") // negative
```

Writing a helper method isSameDayInterval()

We can write a small helper method isSameDayInterval() to simplify our test:

```
private boolean isSameDayInterval() {
  return from.compareTo(to) <= 0;
}
```

Writing the encompasses(LocalTime) method

```
public boolean encompasses(LocalTime time) {
  if(isSameDayInterval()) {
    return from.compareTo(time) <= 0 &&
           time.compareTo(to)   <= 0;
  } else {
    return (from.compareTo(time)       <= 0 &&
            time.compareTo(LAST_NANO) <= 0) ||
           (time.compareTo(to)         <= 0);
  }
}
```

Writing the encompasses(LocalTime) method

```
public boolean encompasses(LocalTime time) {
  if(isSameDayInterval()) { // Simple case
    return from.compareTo(time) <= 0 &&
           time.compareTo(to)   <= 0;
  } else {
    return (from.compareTo(time)      <= 0 &&
            time.compareTo(LAST_NANO) <= 0) ||
           (time.compareTo(to)        <= 0);
  }
}
```

Writing the encompasses(LocalTime) method

```
public boolean encompasses(LocalTime time) {
  if(isSameDayInterval()) {
    return from.compareTo(time) <= 0 &&
           time.compareTo(to)   <= 0;
  } else { // "Harder" case:
    return from.compareTo(time) <= 0 ||
           time.compareTo(to)   <= 0;
  }
}
```

A full day interval

How would you construct a full day interval?

Hint: Smallest unit of a LocalTime is nanoseconds.

A full day interval

How about:

```
Interval allDayLong
 = new Interval(LocalTime.parse("12:00:00"),
                LocalTime.parse("12:00:00").minusNanos(1L));

// The smallest time unit before 12:00:00 is one nano before

// Example printout:
12:00 - 11:59:59.999999999
12:00 - 11:59:59.999999999 is same day interval? false
```

toString() is always nice to have

```
/**
* Returns this Interval as a String, in this format:
* <code>22:00:00 - 04:00:00</code>
* @return A String representation of this Interval
*/
@Override
public String toString() {
  return from + " - " + to;
}
```

What to test?

Think about some tests to see if your class works.

Include same-day-intervals as well as over-midnight-intervals

Include test for encompassing and not encompassing times

Include tests for very small violations of encompassing and very small encompassing margins.

Example tests

```
LocalTime time = LocalTime.parse("23:00:00");
Interval interval
 = new Interval(LocalTime.parse("22:00:00"),
                LocalTime.parse("04:00:00"));
assert interval.encompasses(time) : interval + " should encompass " + time;

time = LocalTime.parse("06:00:00");
assert !interval.encompasses(time) : interval + " shouldn't encompass " + time;

time = LocalTime.parse("21:59:59.999999999");
assert !interval.encompasses(time) : interval + " shouldn't encompass " + time;

time = LocalTime.parse("04:00:01");
assert !interval.encompasses(time) : interval + " shouldn't encompass " + time;

time = LocalTime.parse("04:00:00").plusNanos(1L);
assert interval.encompasses(time) : interval + " should encompass " + time;
```

Bonus - Using Inteval as a TemporalQuery

The java.time.temporal.TemporalQuery<R> interface declares one method:

```
public R queryFrom(TemporalAccessor date)
```

This allows all TemporalAccessors to use a TemporalQuery for its query method.

Let’s look at how Interval can become a TemporalQuery.

Bonus - Using Interval as a TemporalQuery

All we have to do is to implement `TemporalQuery<Boolean>` and the method

```
public Boolean queryFrom(TemporalAccessor date) {
  return new Boolean(encompasses((LocalTime)time));
}
```

Now, we can use the following syntax in client code:

```
time = LocalTime.parse("12:00:00");
if(time.query(interval)) {
  System.out.println(interval + " encompasses " + time);
} else {
  System.out.println(interval + " does not encompass " + time);
}
```

Bonus - Using Interval as a TemporalQuery

However, we're not sure this syntax:

```
time = LocalTime.parse("12:00:00");
if(time.query(interval)) {
  System.out.println(interval + " encompasses " + time);
} else {
  System.out.println(interval + " does not encompass " + time);
}
```

is more clear than this syntax:

```
if(interval.encompasses(time)) {
  System.out.println(interval + " encompasses " + time);
} else {
  System.out.println(interval + " does not encompass " + time);
}
```

Source code

Github: https://github.com/progund/java-api-time

Read

http://www.oracle.com/technetwork/articles/java/jf14-date-time-2125367.html

https://docs.oracle.com/javase/tutorial/datetime/

https://www.tutorialspoint.com/java8/java8_datetime_api.htm