POLITECNICO DI TORINO

e-Lite

Dates and Times in Java

Tecniche di Programmazione – A.A. 2015/2016

Summary

- Libraries and History
- The "old" Date/Calendar classes
- The new (≥Java8) java.time package
  - Basic concepts
  - Main classes
  - Date operations
- Dealing with SQL dates

Libraries and History

Dates and Times in Java

A common problem

- Most software programs need to deal with dates and/or times
- The human calendar system is extremely complex
  - Uneven months, Leap years, Leap seconds
  - Time zones, Daylight savings time
  - Localized representations
  - Time instants vs. time intervals vs. recurring instants
  - Different calendar systems
- Available libraries, in all languages, are often over-simplified or over-engineered

Falsehoods programmers believe about time

All of these assumptions are wrong

1. There are always 24 hours in a day.
2. Months have either 30 or 31 days.
3. Years have 365 days.
4. February is always 28 days long.
5. Any 24-hour period will always begin and end in the same day (or week, or month).
6. A week always begins and ends in the same month.
7. A week (or a month) always begins and ends in the same year.
8. The machine that a program runs on will always be in the GMT time zone.
9. Ok, that's not true. But at least the time zone in which a program has to run will never change.
10. Well, surely there will never be a change to the time zone in which a program hast to run *in production*.
11. The system clock will always be set to the correct local time.
12. The system clock will always be set to a time that is not wildly different from the correct local time.
13. If the system clock is incorrect, it will at least always be off by a consistent number of seconds.
14. The server clock and the client clock will always be set to the same time.
15. The server clock and the client clock will always be set to *around* the same time.
16. Ok, but the time on the server clock and time on the client clock would never be different by a matter of *decades*.
17. If the server clock and the client clock are not in synch, they will at least always be out of synch by a consistent number of seconds.
18. The server clock and the client clock will use the same time zone.
19. The system clock will never be set to a time that is in the distant past or the far future.
20. Time has no beginning and no end.
21. One minute on the system clock has exactly the same duration as one minute on any other clock
22. Ok, but the duration of one minute on the system clock will be *pretty close* to the duration of one minute on most other clocks.
23. Fine, but the duration of one minute on the system clock would never be more than an hour.
24. You can't be serious.
25. The smallest unit of time is one second.
26. Ok, one millisecond.
27. It will never be necessary to set the system time to any value other than the correct local time.
28. Ok, *testing* might require setting the system time to a value other than the correct local time but it will never be necessary to do so *in production*.
29. Time stamps will always be specified in a commonly-understood format like 1339972628 or 133997262837.
30. Time stamps will always be specified in the same format.
31. Time stamps will always have the same level of precision.
32. A time stamp of sufficient precision can safely be considered unique.
33. A timestamp represents the time that an event actually occurred.
34. Human-readable dates can be specified in universally understood formats such as 05/07/11.

UPDATED: There's more! Read the rest of the falsehoods…

http://infiniteundo.com/post/25326999628/falsehoods-programmers-believe-about-time

Two ways at representing time

- Machine time
  - A given number of seconds (ms, ns) measured starting from a known reference point
    - Fixed reference (Epoch): absolute time
    - Variable reference: time intervals
- Human time
  - The passing of time, as we humans measure it
  - Dates: day, month, year, week, weekday, century, …
  - Times: hours, minutes, seconds, ms, …
  - Takes into account local culture
    - Gregorian Calendar, localization, time zones, DST

What we want to represent

- Exact **time instants**: Now. The moment of moon landing.
- **Days** (without times): The date I was born. The discovery of Americas.
- **Times** (without dates): Office hours are 9-17.
- **Recurring dates**: Wedding anniversary. Christmas day. (date without year)
- **Date intervals**: One week. Seven Days. 30 working days.
- **Relative dates**: next Thursday. By the end of next month.
- …

Basic operations

- Parsing: convert a string into a date/time object
- Formatting: covert a date/time object into a string
- Building: create a date/time object starting from its components
- Analyzing: extracting date/time components from an object
- Arithmetic: sum or subtract a quantity from a date/time; compute the difference between two dates/times; equality or majority comparing

In Java (<=7)

- java.util.Date (and related)
  - Since the first version of Java (JDK 1.0)
  - Oversimplified, incomplete
  - Most of it was deprecated in JDK 1.1
  - But still alive today
- java.util.Calendar (and related)
  - Code donated by IBM to Sun
  - Supports nearly all time and date details
  - Overengineered, complex
  - Unexpected behaviors
  - Cannot completely replace Date (need to convert back&forth)

In Java (>=8)

- New java.time package
  - Inspired by the «JodaTime» library
- Cleaner structure, easier usage
- Optimized on common use cases
  - While supporting the more complex ones
- Explicit distinction between machine time and human time

The “old” Date/Calendar classes

Dates and Times in Java

java.util.Date

- The Date object is really just a wrapper around a **long** integer
  - The number of milliseconds since January 1, 1970.
  - It represents a date **and a time** (the name is wrong!)
  - Works in UTC time, but not prefectly (leap seconds)
  - Most methods are deprecated, now, in favor of Calendar or DataFormatter objects

Date constructors

- Date() Allocates a Date object and initializes it so that it represents the time at which it was allocated, measured to the nearest millisecond.
- Date(long date) Allocates a Date object and initializes it to represent the specified number of milliseconds since the standard base time known as "the epoch", namely January 1, 1970, 00:00:00 GMT.

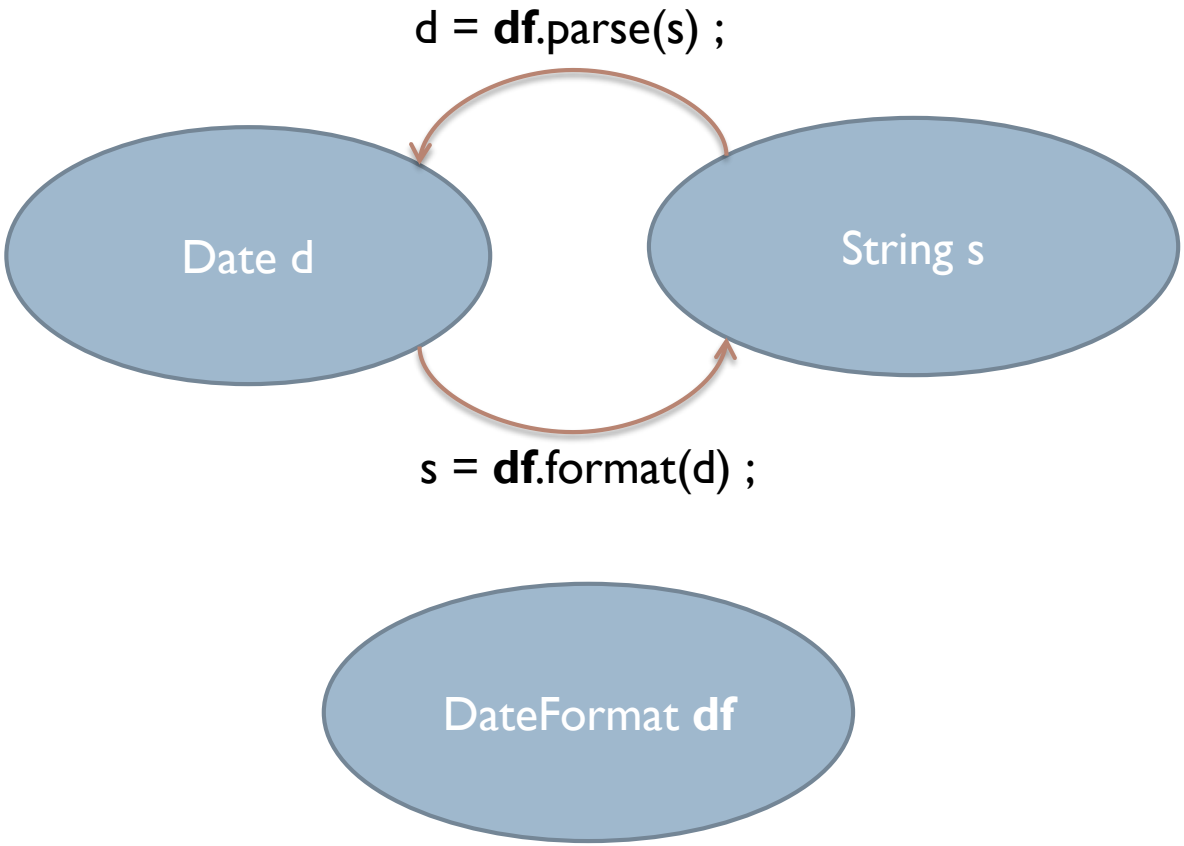
Date methods (non-Deprecated)

boolean	after(Date when) Tests if this date is after the specified date.
boolean	before(Date when) Tests if this date is before the specified date.
long	getTime() Returns the number of milliseconds since January 1, 1970, 00:00:00 GMT represented by this Date object.
void	setTime(long time) Sets this Date object to represent a point in time that is time milliseconds after January 1, 1970 00:00:00 GMT.
String	toString() Converts this Date object to a String of the form: dow mon dd hh:mm:ss zzz yyyy (example: Thu May 21 10:07:28 CEST 2015)
boolean	equals(Object obj) Compares two dates for equality.
int	hashCode() Returns a hash code value for this object.
Object	clone() Return a copy of this object.
int	compareTo(Date anotherDate) Compares two Dates for ordering.

java.text.DateFormat

- Abstract class for date/time formatting subclasses which formats and parses dates or time in a language-independent manner
  - Subclasses: SimpleDateFormat
  - allows for formatting (i.e., date → text), parsing (text → date), and normalization
  - The formatting styles include FULL, LONG, MEDIUM, and SHORT
- A formatter is generated by a .get*Xxx*Instance static factory method
  - DateFormat.getDateInstance()
  - DateFormat.getTimeInstance()
  - DateFormat.getDateTimeInstance()

Operations in DateFormat



Examples

Date today = new Date() ; System.out.println(today.toString()) ;	Thu May 21 10:14:33 CEST 2015
DateFormat format = DateFormat.getDateInstance() ; System.out.println(format.format(today)) ;	21-mag-2015
System.out.println(DateFormat.getDateInstance(DateFormat.FULL).format(today)) ;	giovedì 21 maggio 2015
System.out.println(DateFormat.getDateInstance(DateFormat.LONG).format(today)) ;	21 maggio 2015
System.out.println(DateFormat.getDateInstance(DateFormat.MEDIUM).format(today)) ;	21-mag-2015
System.out.println(DateFormat.getDateInstance(DateFormat.SHORT).format(today)) ;	21/05/15

Format localization

System.out.println(DateFormat.getDateInstance(DateFormat.FULL, Locale.FRANCE) .format(today)) ;	jeudi 21 mai 2015
System.out.println(DateFormat.getDateInstance(DateFormat.FULL, Locale.GERMANY) .format(today)) ;	Donnerstag, 21. Mai 2015
System.out.println(DateFormat.getDateInstance(DateFormat.FULL, Locale.US).format(today)) ;	Thursday, May 21, 2015
System.out.println(DateFormat.getDateInstance(DateFormat.FULL, Locale.CHINA).format(today)) ;	2015年5月21日 星期四
System.out.println(DateFormat.getDateInstance(DateFormat.FULL, Locale.JAPAN).format(today)) ;	2015年5月21日
System.out.println(DateFormat.getDateInstance(DateFormat.FULL, new Locale("AR")) .format(today)) ;	21 مايو, 2015

Custom formats

- Use SimpleDateFormat
  - new SimpleDateFormat(String pattern)
- Defines a «pattern» for representing dates/times
- May format or parse according to the pattern

SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd") ; System.out.println(sdf.format(today)) ;	2015-05-21
SimpleDateFormat sdf2 = new SimpleDateFormat("hh:mm:ss") ; System.out.println(sdf2.format(today)) ;	10:38:52

Formatting Patterns

Letter	Date or Time Component	Presentation	Examples
G	Era designator	Text	AD
y	Year	Year	1996; 96
Y	Week year	Year	2009; 09
M	Month in year (context sensitive)	Month	July; Jul; 07
L	Month in year (standalone form)	Month	July; Jul; 07
w	Week in year	Number	27
W	Week in month	Number	2
D	Day in year	Number	189
d	Day in month	Number	10
F	Day of week in month	Number	2
E	Day name in week	Text	Tuesday; Tue
u	Day number of week (1 = Monday, ..., 7 = Sunday)	Number	1
a	Am/pm marker	Text	PM
H	Hour in day (0-23)	Number	0
k	Hour in day (1-24)	Number	24
K	Hour in am/pm (0-11)	Number	0
h	Hour in am/pm (1-12)	Number	12
m	Minute in hour	Number	30
s	Second in minute	Number	55
S	Millisecond	Number	978
z	Time zone	General time zone	Pacific Standard Time; PST; GMT-08:00
Z	Time zone	RFC 822 time zone	-0800
X	Time zone	ISO 8601 time zone	-08; -0800; -08:00

Examples

Date and Time Pattern	Result
"yyyy.MM.dd G 'at' HH:mm:ss z"	2001.07.04 AD at 12:08:56 PDT
"EEE, MMM d, ''yy"	Wed, Jul 4, '01
"h:mm a"	12:08 PM
"hh 'o''clock' a, zzzz"	12 o'clock PM, Pacific Daylight Time
"K:mm a, z"	0:08 PM, PDT
"yyyyy.MMMMM.dd GGG hh:mm aaa"	02001.July.04 AD 12:08 PM
"EEE, d MMM yyyy HH:mm:ss Z"	Wed, 4 Jul 2001 12:08:56 -0700
"yyMMddHHmmssZ"	010704120856-0700
"yyyy-MM-dd'T'HH:mm:ss.SSSZ"	2001-07-04T12:08:56.235-0700
"yyyy-MM-dd'T'HH:mm:ss.SSSXXX"	2001-07-04T12:08:56.235-07:00
"YYYY-'W'ww-u"	2001-W27-3

Parsing

- public Date parse(String text) Parses text from a string to produce a Date.

```
try {
    String nataleString = "25/12/2015" ;

    SimpleDateFormat sdf_it = new SimpleDateFormat("dd/MM/yyyy") ;
    Date nataleDate = sdf_it.parse(nataleString) ;

    System.out.println(nataleDate.toString()) ;
} catch (ParseException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
```

Date operations?

- The class Date is **not** able to do **any** computation on dates
- Only methods are:
  - `date1.before(date2)`
  - `date1.after(date2)`
  - `date1.compareTo(date2)`
- For all the rest, you must use **Calendar**.

java.util.Calendar

- Abstract class that provides methods for
  - converting between a specific instant in time and a set of calendar fields (YEAR, MONTH, DAY_OF_MONTH, HOUR, …)
  - manipulating the calendar fields, such as getting the date of the next week.
- An instant in time can be represented by a millisecond value that is an offset from the *Epoch*, January 1, 1970 00:00:00.000 GMT (Gregorian).
- May obtain a localized instance:
  - Calendar rightNow = Calendar.getInstance();

Setting a date / time

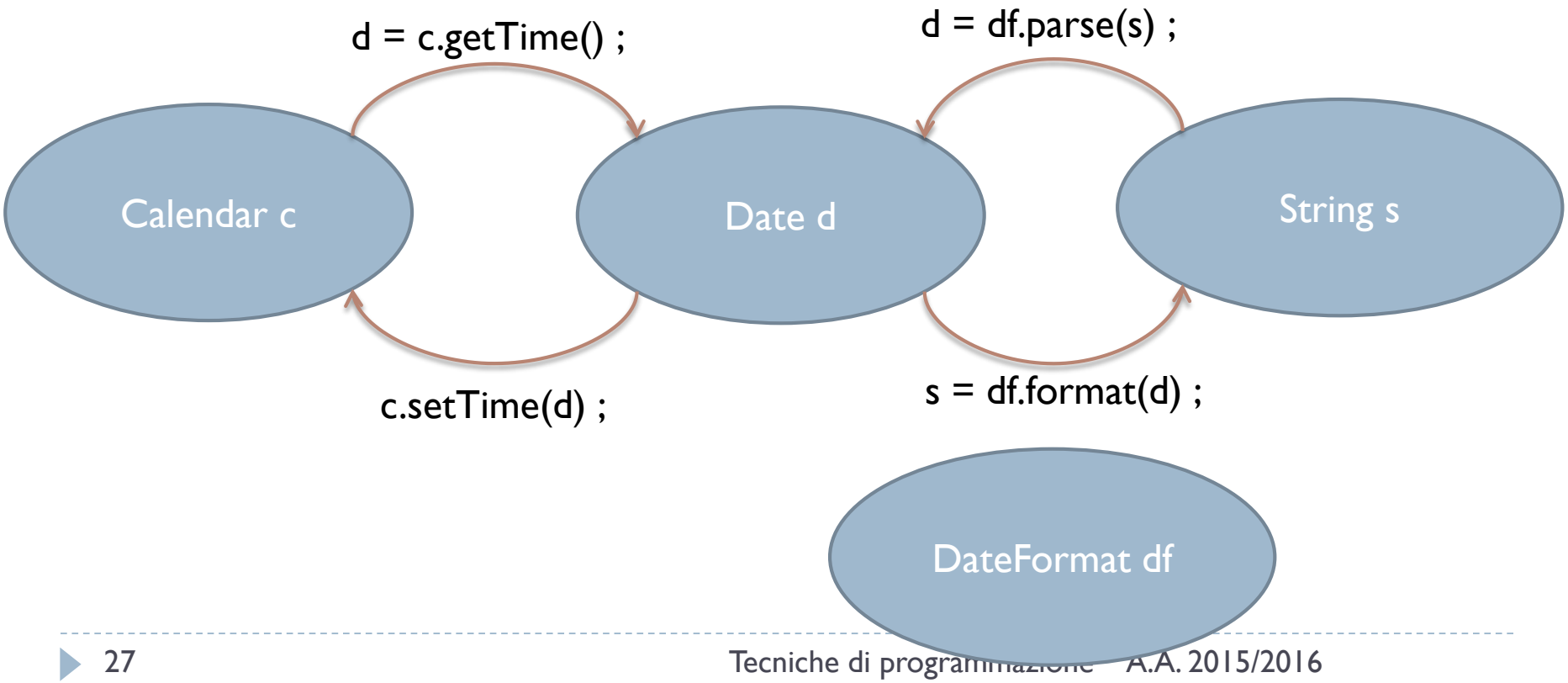
- calendar.setTime(Date date)
  - Will store in the calendar the same value of the Date object
- calendar.set(int field, int value)
  - Sets or modifies one specific field
  - Fields may be calendar-specific, we use GregorianCalendar
- Set full dates in one call
  - set(int year, int month, int date) Sets the values for the calendar fields YEAR, MONTH, and DAY_OF_MONTH.
  - set(int year, int month, int date, int hourOfDay, int minute) Sets the values for the calendar fields YEAR, MONTH, DAY_OF_MONTH, HOUR_OF_DAY, and MINUTE.
  - set(int year, int month, int date, int hourOfDay, int minute, int second) Sets the values for the fields YEAR, MONTH, DAY_OF_MONTH, HOUR_OF_DAY, MINUTE, and SECOND.

GregorianCalendar fields

Field	Default Value
ERA	AD
YEAR	1970
MONTH	JANUARY
DAY_OF_MONTH	1
DAY_OF_WEEK	the first day of week
WEEK_OF_MONTH	0
DAY_OF_WEEK_IN_MONTH	1
AM_PM	AM
HOUR, HOUR_OF_DAY, MINUTE, SECOND, MILLISECOND	0

Formatting/Parsing calendars

- **No methods** available in Calendar
- Must use DateFormat objects
- This implies converting to/from Date objects



Date arithmetics with Calendar

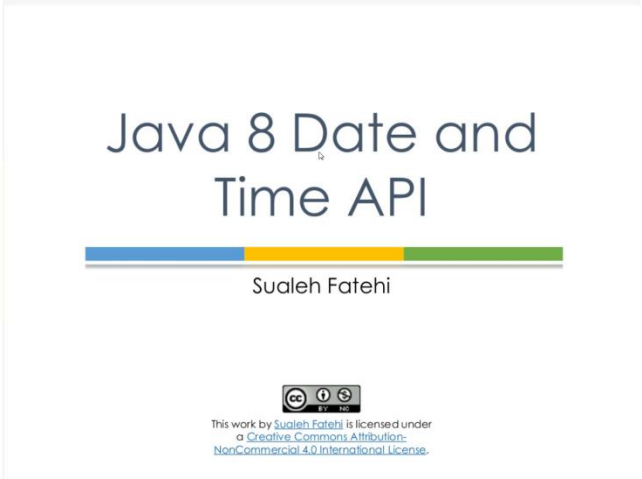
void	add(int field, int amount) Adds or subtracts the specified amount of time to the given calendar field, based on the calendar's rules.
boolean	after(Object when) Returns whether this Calendar represents a time after the time represented by the specified Object.
boolean	before(Object when) Returns whether this Calendar represents a time before the time represented by the specified Object.

The new java.time package

Dates and Times in Java

Good introductions

http://www.slideshare.net/sualeh/java-8-date-and-time-api



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

Main principles (1/2)

- **Clear**
  - The methods in the API are well defined and their behavior is clear and expected.
  - For example, invoking a Date-Time method with a null parameter value typically triggers a NullPointerException.
- **Fluent**
  - "Fluent" interface, code easy to read.
  - Most methods do not allow null parameters and do not return null ➔ method calls can be safely chained
  - For example:
    - LocalDate today = LocalDate.now(); LocalDate payday = today.with(TemporalAdjusters.lastDayOfMonth()).minusDays(2);

Main principles (2/2)

- **Immutable**
  - Most of the classes in the Date-Time API create objects that are **immutable**.
  - To alter the value of an immutable object, a **new** object must be constructed as a modified copy of the original.
  - Methods to create date/time objects are prefixed with **of**, **from**, or **with**, rather than constructors, and there are **no set methods**. For example:
    - LocalDate dateOfBirth = LocalDate.of(2012, Month.MAY, 14);
    - LocalDate firstBirthday = dateOfBirth.plusYears(1);
- **Extensible**
  - The Date-Time API is extensible wherever possible (you can define your own time adjusters and queries, or build your own calendar system).

A variety of Temporal Classes

java.time.

Class or Enum	Year	Month	Day	Hours	Minutes	Seconds*	Zone Offset	Zone ID	toString Output
Instant						✓			2013-08-20T15:16:26.355Z
LocalDate	✓	✓	✓						2013-08-20
LocalDateTime	✓	✓	✓	✓	✓	✓			2013-08-20T08:16:26.937
ZonedDateTime	✓	✓	✓	✓	✓	✓	✓	✓	2013-08-21T00:16:26.941+09:00[Asia/Tokyo]
LocalTime				✓	✓	✓			08:16:26.943
MonthDay		✓	✓						--08-20
Year	✓								2013
YearMonth	✓	✓							2013-08
Month		✓							AUGUST
OffsetDateTime	✓	✓	✓	✓	✓	✓	✓		2013-08-20T08:16:26.954-07:00
OffsetTime				✓	✓	✓	✓		08:16:26.957-07:00
Duration			**	**	**	✓			PT20H (20 hours)
Period	✓	✓	✓				***	***	P10D (10 days)

Consistent Method Naming Conventions

Prefix	Method Type	Use
`of`	static factory	Creates an instance where the factory is primarily validating the input parameters, not converting them.
`from`	static factory	Converts the input parameters to an instance of the target class, which may involve losing information from the input.
`parse`	static factory	Parses the input string to produce an instance of the target class.
`format`	instance	Uses the specified formatter to format the values in the temporal object to produce a string.
`get`	instance	Returns a part of the state of the target object.
`is`	instance	Queries the state of the target object.
`with`	instance	Returns a **copy** of the target object with **one element changed**; this is the immutable equivalent to a set method on a JavaBean.
`plus`	instance	Returns a copy of the target object with an amount of time added.
`minus`	instance	Returns a copy of the target object with an amount of time subtracted.
`to`	instance	**Converts** this object to another type.
`at`	instance	Combines this object with another.

Examples

LocalDateTime now = LocalDateTime.now() ;	now.toString() = 2015-05-21T11:36:48.008	Defaults to ISO 8601 format
LocalDate natale = LocalDate.of(2015, 12, 25) ; LocalDate natale = LocalDate.of(2015, Month.DECEMBER, 25) ;	natale.toString() = 2015-12-25	A Date has no Time component

Accessing fields

- In general: get(TemporalField field)
- In detail:
  - getDayOfMonth()
  - getDayOfWeek()
  - getDayOfYear()
  - getHour()
  - getMinute()
  - getMonth()
  - getMonthValue()
  - getNano()
  - getSecond()
  - getYear()

Machine time

- The `Instant` class, represents the start of a nanosecond on the timeline. It counts time beginning from the first second of January 1, 1970 (1970-01-01T00:00:00Z) - the *Epoch*.
- The `Instant` class does not work with human units of time, such as years, months, or days
- If you want to perform calculations in those units, you can **convert** an `Instant` to another class, such as `LocalDateTime`
  - ```
    LocalDateTime ldt =
    LocalDateTime.ofInstant(instant,
    ZoneId.systemDefault());
    ```
    ```

Parsing and Formatting

- Methods `.parse()` and `.format()` in **all** date and time classes
- By default, work with ISO formats
- May use a `DateTimeFormatter` to customize the format
  - Many commonly used `DateTimeFormatter` isntances are pre-defined

Predefined formatters

Formatter	Description	Example
`ofLocalizedDate(dateStyle)`	Formatter with date style from the locale	'2011-12-03'
`ofLocalizedTime(timeStyle)`	Formatter with time style from the locale	'10:15:30'
`ofLocalizedDateTime(dateTimeStyle)`	Formatter with a style for date and time from the locale	'3 Jun 2008 11:05:30'
`ofLocalizedDateTime(dateStyle,timeStyle)`	Formatter with date and time styles from the locale	'3 Jun 2008 11:05'
`BASIC_ISO_DATE`	Basic ISO date	'20111203'
`ISO_LOCAL_DATE`	ISO Local Date	'2011-12-03'
`ISO_OFFSET_DATE`	ISO Date with offset	'2011-12-03+01:00'
`ISO_DATE`	ISO Date with or without offset	'2011-12-03+01:00'; '2011-12-03'
`ISO_LOCAL_TIME`	Time without offset	'10:15:30'
`ISO_OFFSET_TIME`	Time with offset	'10:15:30+01:00'
`ISO_TIME`	Time with or without offset	'10:15:30+01:00'; '10:15:30'
`ISO_LOCAL_DATE_TIME`	ISO Local Date and Time	'2011-12-03T10:15:30'
`ISO_OFFSET_DATE_TIME`	Date Time with Offset	2011-12-03T10:15:30+01:00'
`ISO_ZONED_DATE_TIME`	Zoned Date Time	'2011-12-03T10:15:30+01:00[Europe/Paris]'
`ISO_DATE_TIME`	Date and time with ZoneId	'2011-12-03T10:15:30+01:00[Europe/Paris]'
`ISO_ORDINAL_DATE`	Year and day of year	'2012-337'
`ISO_WEEK_DATE`	Year and Week	2012-W48-6'
`ISO_INSTANT`	Date and Time of an Instant	'2011-12-03T10:15:30Z'
`RFC_1123_DATE_TIME`	RFC 1123 / RFC 822	'Tue, 3 Jun 2008 11:05:30 GMT'

Predefined formatters

Formatter	Description	Example
ofLocalizedDate(dateStyle)	Formatter with date style from the locale	'2011-12-03'
ofLocalizedTime(timeStyle)	Formatter with time style from the locale	'10:15:30'
ofLocalizedDateTime(dateTimeStyle)	Formatter with a style for date and time from the locale	'3 Jun 2008 11:05:30'
ofLocalizedDateTime(dateStyle,timeStyle)	Formatter with date and time styles from the locale	'3 Jun 2008 11:05'
BASIC_ISO_DATE	Basic ISO date	'20111203'
ISO_LOCAL_DATE	ISO Local Date	'2011-12-03'
ISO_OFFSET_DATE	ISO Date with offset	'2011-12-03+01:00'
ISO_DATE	ISO Date with or without offset	'2011-12-03+01:00'; '2011-12-03'
ISO_LOCAL_TIME	Time without offset	'10:15:30'
ISO_OFFSET_TIME	Time with offset	'10:15:30+01:00'

```
DateTimeFormatter.ISO_DATE.format(natale)

DateTimeFormatter.ofLocalizedDate(FormatStyle.LONG).format(natale)
```

Custom formatters

Symbol	Meaning	Presentation	Examples
G	era	text	AD; Anno Domini; A
u	year	year	2004; 04
y	year-of-era	year	2004; 04
D	day-of-year	number	189
M/L	month-of-year	number/text	7; 07; Jul; July; J
d	day-of-month	number	10
Q/q	quarter-of-year	number/text	3; 03; Q3; 3rd quarter
Y	week-based-year	year	1996; 96
w	week-of-week-based-year	number	27
W	week-of-month	number	4
E	day-of-week	text	Tue; Tuesday; T
e/c	localized day-of-week	number/text	2; 02; Tue; Tuesday; T
F	week-of-month	number	3
a	am-pm-of-day	text	PM
h	clock-hour-of-am-pm (1-12)	number	12
K	hour-of-am-pm (0-11)	number	0
k	clock-hour-of-am-pm (1-24)	number	0
H	hour-of-day (0-23)	number	0
m	minute-of-hour	number	30
s	second-of-minute	number	55
S	fraction-of-second	fraction	978
A	milli-of-day	number	1234
n	nano-of-second	number	987654321
N	nano-of-day	number	1234000000
V	time-zone ID	zone-id	America/Los_Angeles; Z; -08:30
z	time-zone name	zone-name	Pacific Standard Time; PST
O	localized zone-offset	offset-O	GMT+8; GMT+08:00; UTC-08:00;
X	zone-offset 'Z' for zero	offset-X	Z; -08; -0830; -08:30; -083015; -08:30:15;
x	zone-offset	offset-x	+0000; -08; -0830; -08:30; -083015; -08:30:15;
Z	zone-offset	offset-Z	+0000; -0800; -08:00;
p	pad next	pad modifier	1
'	escape for text	delimiter	
''	single quote	literal	'

Custom formatters

Symbol	Meaning	Presentation	Examples
G	era	text	AD; Anno Domini; A
u	year	year	2004; 04
y	year-of-era	year	2004; 04
D	day-of-year	number	189
M/L	month-of-year	number/text	7; 07; Jul; July; J
d	day-of-month	number	10
Q/q	quarter-of-year	number/text	3; 03; Q3; 3rd quarter
Y	week-based-year	year	1996; 96
w	week-of-week-based-year	number	27
W	week-of-month	number	4
E	day-of-week	text	Tue; Tuesday; T
e/c	localized day-of-week	number/text	2; 02; Tue; Tuesday; T
F	week-of-month		
a	am-pm-of-day		
h	clock-hour-of-		
K	hour-of-am-pm		
k	clock-hour-of-		
H	hour-of-day (0-		
m	minute-of-hour		
s	second-of-minut		
S	fraction-of-se		
A	milli-of-day		
n	nano-of-second		
N	nano-of-day		
V	time-zone ID		
z	time-zone name		
O	localized zone-		
X	zone-offset 'Z'		
x	zone-offset		
Z	zone-offset		
p	pad next		
'	escape for text	delimiter	
''	single quote	literal	'

```
DateTimeFormatter formatter =
DateTimeFormatter.ofPattern("yyyy MM dd");

String text = date.toString(formatter);

LocalDate date = LocalDate.parse(text,
                                 formatter);
```

Date/Time arithmetics

- The date and time classes already contain basic operations for adding/subtracting/comparing

- For more complex operations, you may use the `TemporalAdjuster` classes, as a parameter of the `.with()` method of date/time classes
  - Many predefinted `TemporalAdjuster` classes already defined as static instances of `TemporalAdjusters`

Arithmetic in LocalDateTime

- `isAfter(other)` `isBefore(other)` `isEqual(other)`
- `minus(long amountToSubtract, TemporalUnit unit)` `minus(TemporalAmount amountToSubtract)`
- `minusDays(long days)` `minusHours(long hours)` `minusMinutes(long minutes)` `minusMonths(long months)` `minusNanos(long nanos)` `minusSeconds(long seconds)` `minusWeeks(long weeks)` `minusYears(long years)`
- Same with `plus`*`XXX`*`()`

Computing differences

- With `LocalDate` objects
  - `until(Temporal endExclusive, TemporalUnit unit)`
  - Calculates the amount of time until another date-time in terms of the specified unit.
- With `Instants` and `Duration.beteween`
  - `Instant t1, t2;`
  - `long ns = Duration.between(t1, t2).toNanos();`
- With `Dates` and `Period.between`
  - `LocalDate today = LocalDate.now();`
  - `LocalDate birthday = LocalDate.of(1960, Month.JANUARY, 1);`
  - `Period p = Period.between(birthday, today);`

Temporal Adjusters

- dayOfWeekInMonth(int ordinal, DayOfWeek dayOfWeek) a new date in the same month with the ordinal day-of-week
- firstDayOfMonth() a new date set to the first day of the current month
- firstDayOfNextMonth() a new date set to the first day of the next month
- firstDayOfNextYear() a new date set to the first day of the next year.
- firstDayOfYear() a new date set to the first day of the current year.
- firstInMonth(DayOfWeek dayOfWeek) a new date in the same month with the first matching day-of-week
- lastDayOfMonth() a new date set to the last day of the current month
- lastDayOfYear() a new date set to the last day of the current year
- lastInMonth(DayOfWeek dayOfWeek) a new date in the same month with the last matching day-of-week.
- next(DayOfWeek dayOfWeek) adjusts the date to the first occurrence of the specified day-of-week after the date being adjusted
- nextOrSame(DayOfWeek dayOfWeek) adjusts the date to the first occurrence of the specified day-of-week after the date being adjusted unless it is already on that day in which case the same object is returned.
- previous(DayOfWeek dayOfWeek) adjusts the date to the first occurrence of the specified day-of-week before the date being adjusted
- previousOrSame(DayOfWeek dayOfWeek) adjusts the date to the first occurrence of the specified day-of-week before the date being adjusted unless it is already on that day in which case the same object is returned

Example

```
LocalDate date = LocalDate.of(2000, Month.OCTOBER, 15);
DayOfWeek dotw = date.getDayOfWeek();
System.out.printf("%s is on a %s%n", date, dotw);

System.out.printf("first day of Month: %s%n",
                  date.with(TemporalAdjusters.firstDayOfMonth()));
System.out.printf("first Monday of Month: %s%n",
                  date.with(TemporalAdjusters.firstInMonth(DayOfWeek.MONDAY)));
System.out.printf("last day of Month: %s%n",
                  date.with(TemporalAdjusters.lastDayOfMonth()));
System.out.printf("first day of next Month: %s%n",
                  date.with(TemporalAdjusters.firstDayOfNextMonth()));
System.out.printf("first day of next Year: %s%n",
                  date.with(TemporalAdjusters.firstDayOfNextYear()));
System.out.printf("first day of Year: %s%n",
                  date.with(TemporalAdjusters.firstDayOfYear()));
```

Compatibility JDK7-JDK8

- Calendar.toInstant() converts the Calendar object to an Instant.
- GregorianCalendar.toZonedDateTime() converts a GregorianCalendar instance to a ZonedDateTime.
- GregorianCalendar.from(ZonedDateTime) creates a GregorianCalendar object using the default locale from a ZonedDateTime instance.
- Date.from(Instant) creates a Date object from an Instant.
- Date.toInstant() converts a Date object to an Instant.
- TimeZone.toZoneId() converts a TimeZone object to a ZoneId.

Summary

- The **Instant** class provides a machine view of the timeline.
- The **LocalDate**, **LocalTime**, and **LocalDateTime** classes provide a human view of date and time without any reference to time zone.
- The ZoneId, ZoneRules, and ZoneOffset classes describe time zones, time zone offsets, and time zone rules.
- The ZonedDateTime class represents date and time with a time zone. The OffsetDateTime and OffsetTime classes represent date and time, or time, respectively. These classes take a time zone offset into account.
- The **Duration** class measures an amount of time in seconds and nanoseconds.
- The **Period** class measures an amount of time using years, months, and days.

Dealing with SQL dates

Dates and Times in Java

Adding SQL into the picture

- How are dates and times represented in standard SQL?
- How are dates and times implemented in MySQL?
  - Differences, incompatibilities
- How are dates and times transferred over JDBC?

“Standard” SQL

- `DATE`: for date values (e.g. `2011-05-03`)
- `TIME`: for time values (e.g. `15:51:36`). The granularity of the time value is usually a *tick* (100 nanoseconds).
- `TIMESTAMP`: This is a `DATE` and a `TIME` put together in one variable (e.g. `2011-05-03 15:51:36`).
- `TIME WITH TIME ZONE` or `TIMETZ`: the same as `TIME`, but including details about the time zone in question.
- `TIMESTAMP WITH TIME ZONE` or `TIMESTAMPTZ`: the same as `TIMESTAMP`, but including details about the time zone in question.

http://en.wikipedia.org/wiki/SQL#Date_and_time

MySQL (1/2)

- DATE: values with a date part but no time part, in 'YYYY-MM-DD' format. Supported range '1000-01-01' to '9999-12-31'.
- DATETIME: values that contain both date and time parts, in 'YYYY-MM-DD HH:MM:SS' format. Supported range is '1000-01-01 00:00:00' to '9999-12-31 23:59:59'.
- TIMESTAMP: values that contain both date and time parts. Range of '1970-01-01 00:00:01' UTC to '2038-01-19 03:14:07' UTC. Internally represented as Unix milliseconds

- TIMESTAMP and DATETIME offer automatic initialization and updating to the current date and time:
  - Non-standard DEFAULT CURRENT_TIMESTAMP column attribute

MySQL (2/2)

- `TIME`: values in 'HH:MM:SS' format (or 'HHH:MM:SS'). Values may range from '-838:59:59' to '838:59:59'. May represent the hour or the day, or an elapsed time interval (even >24hr)
- YEAR: a 1-byte type used to represent year values. It can be declared as YEAR or YEAR(4) and has a display width of four characters.
  - Year values in the range 00-69 are converted to 2000-2069.
  - Year values in the range 70-99 are converted to 1970-1999.

JDBC (MySQL Connector/J)

- Supported SQL types are enumerated in java.sql.Types
  - http://docs.oracle.com/javase/8/docs/api/java/sql/Types.html
- Represented in java as classes in `java.sql`
  - `java.sql.Date` (subclass of `java.util.Date`)
    - the millisecond values wrapped by a `java.sql.Date` instance must be '*normalized*' by setting the hours, minutes, seconds, and milliseconds to **zero**
  - `java.sql.Time` (subclass of `java.util.Date`)
    - The date components should be set to the "**zero** epoch" value of January 1, 1970 and should not be accessed.
  - `java.sql.Timestamp` (subclass of `java.util.Date`)
    - Supports fractional seconds. A composite of a `java.util.Date` and a separate nanoseconds value.
- **Must** be used in st.setXxx() and rs.getXXX() methods

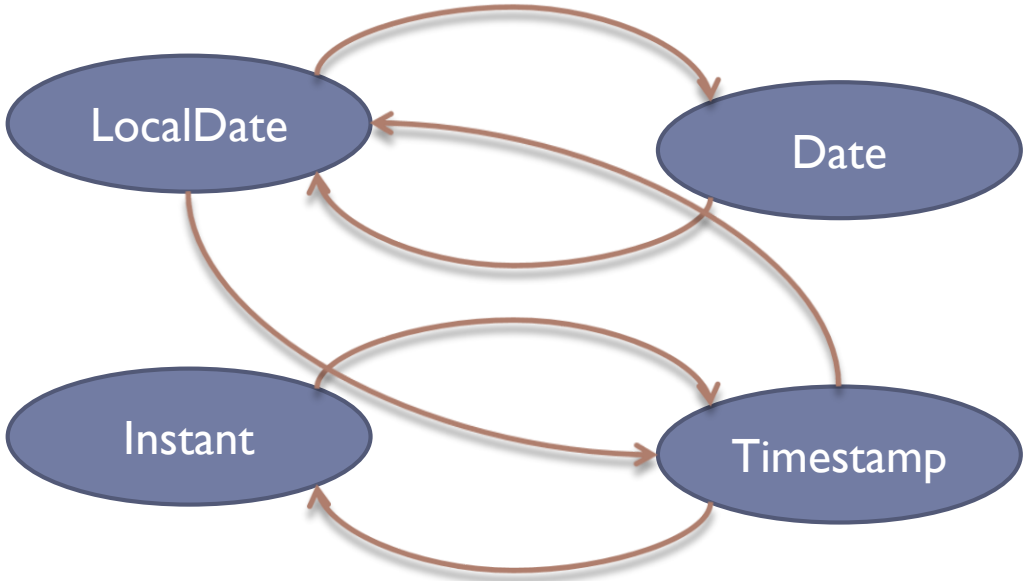
MySQL to Java mappings

MySQL Type Name	Return value of GetColumn ClassName	Returned as Java Class
DATE	DATE	java.sql.Date
DATETIME	DATETIME	java.sql.Timestamp
TIMESTAMP[(M)]	TIMESTAMP	java.sql.Timestamp
TIME	TIME	java.sql.Time
YEAR[(2|4)]	YEAR	If yearIsDateType configuration property is set to false, then the returned object type is java.sql.Short. If set to true (the default), then the returned object is of type java.sql.Date with the date set to January 1st, at midnight.

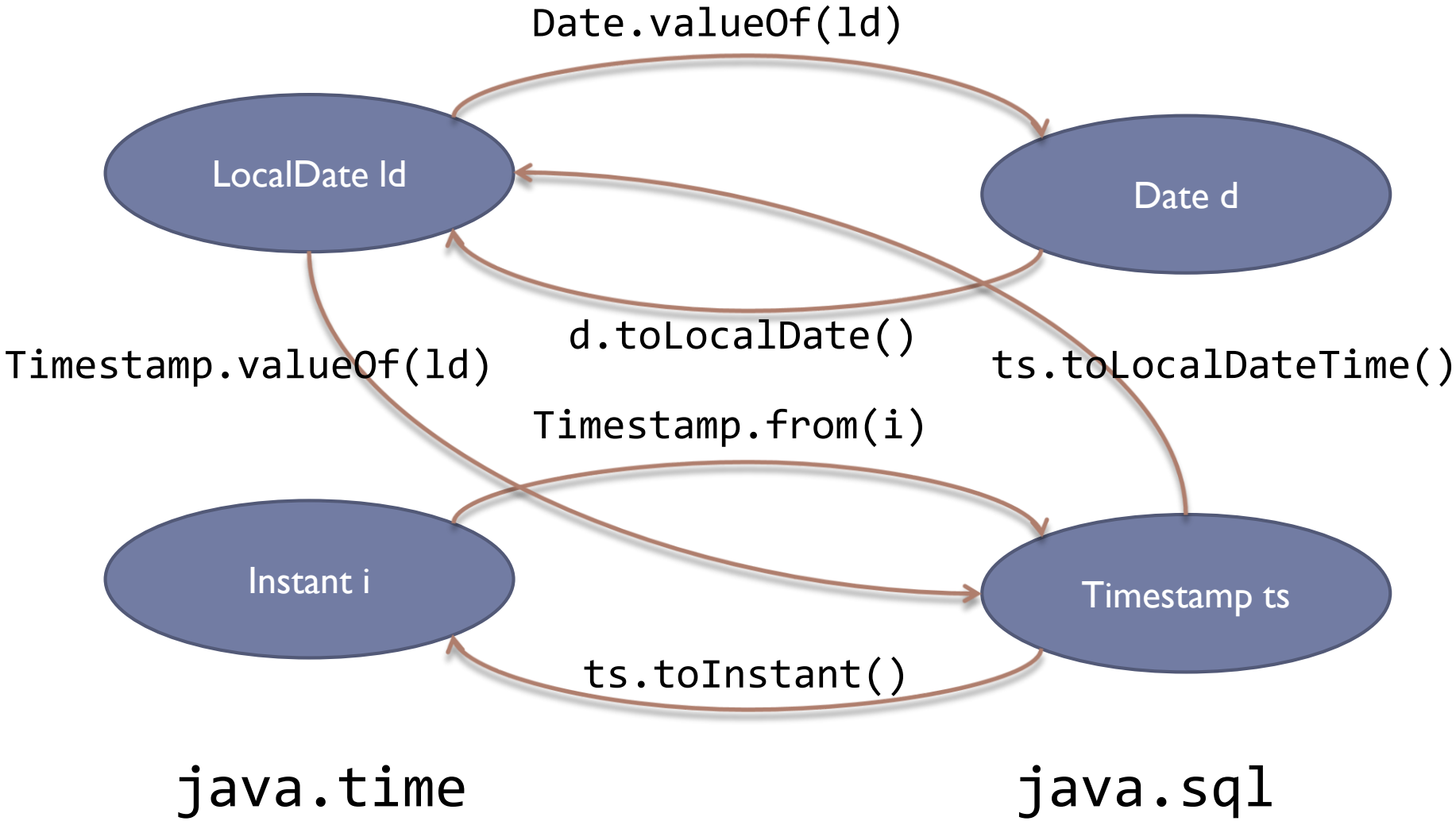
http://dev.mysql.com/doc/connector-j/en/connector-j-reference-type-conversions.html

From SQL to Java 8

- `java.sql.Timestamp` supports
  - `static Timestamp from(Instant instant)`
  - `Instant toInstant()`
  - `LocalDateTime toLocalDateTime()`
  - `static Timestamp valueOf(LocalDateTime dateTime)`
- `java.sql.Date` supports
  - `LocalDate toLocalDate()`
  - `static Date valueOf(LocalDate date)`



From SQL to Java 8



Resources

- JDK8 java.time
  - Official tutorial http://docs.oracle.com/javase/tutorial/datetime/TOC.html
  - JavaDoc https://docs.oracle.com/javase/8/docs/api/java/time/package-summary.html
- MySql Date and times
  - http://dev.mysql.com/doc/refman/5.7/en/date-and-time-types.html
- MySQL Connector/J
  - http://dev.mysql.com/doc/connector-j/en/index.html
- Comparison of different SQL implementations
  - http://troels.arvin.dk/db/rdbms/

Licenza d'uso

- Queste diapositive sono distribuite con licenza Creative Commons "Attribuzione - Non commerciale - Condividi allo stesso modo (CC BY-NC-SA)"
- Sei libero:
  - di riprodurre, distribuire, comunicare al pubblico, esporre in pubblico, rappresentare, eseguire e recitare quest'opera
  - di modificare quest'opera

- Alle seguenti condizioni:
  - **Attribuzione** — Devi attribuire la paternità dell'opera agli autori originali e in modo tale da non suggerire che essi avallino te o il modo i cui tu usi l'opera.

  - **Non commerciale** — Non puoi usare quest'opera per fini commerciali.

  - **Condividi allo stesso modo** — Se alteri o trasformi quest'opera, o se la usi per crearne un'altra, puoi distribuire l'opera risultante solo con un licenza identica o equivalente a questa.

- http://creativecommons.org/licenses/by-nc-sa/3.0/