JAVA PROGRAMMING BASICS

Module 2: Java Object-oriented Programming

Training program

1. Classes and Instances
2. The Methods
3. The Constructors
4. Static Elements
5. Initialization sections
6. Package
7. Inheritance and Polymorphism
8. Abstract classes and Interfaces
9. String processing
10. Wrapper classes for primitive types
11. Exceptions and Assertions
12. Nested classes
13. Enums
14. Generics
15. Collections
16. Method overload resolution
17. Multithreads
18. **Core Java classes**
19. Object Oriented Design
20. Functional Programming

Module contents

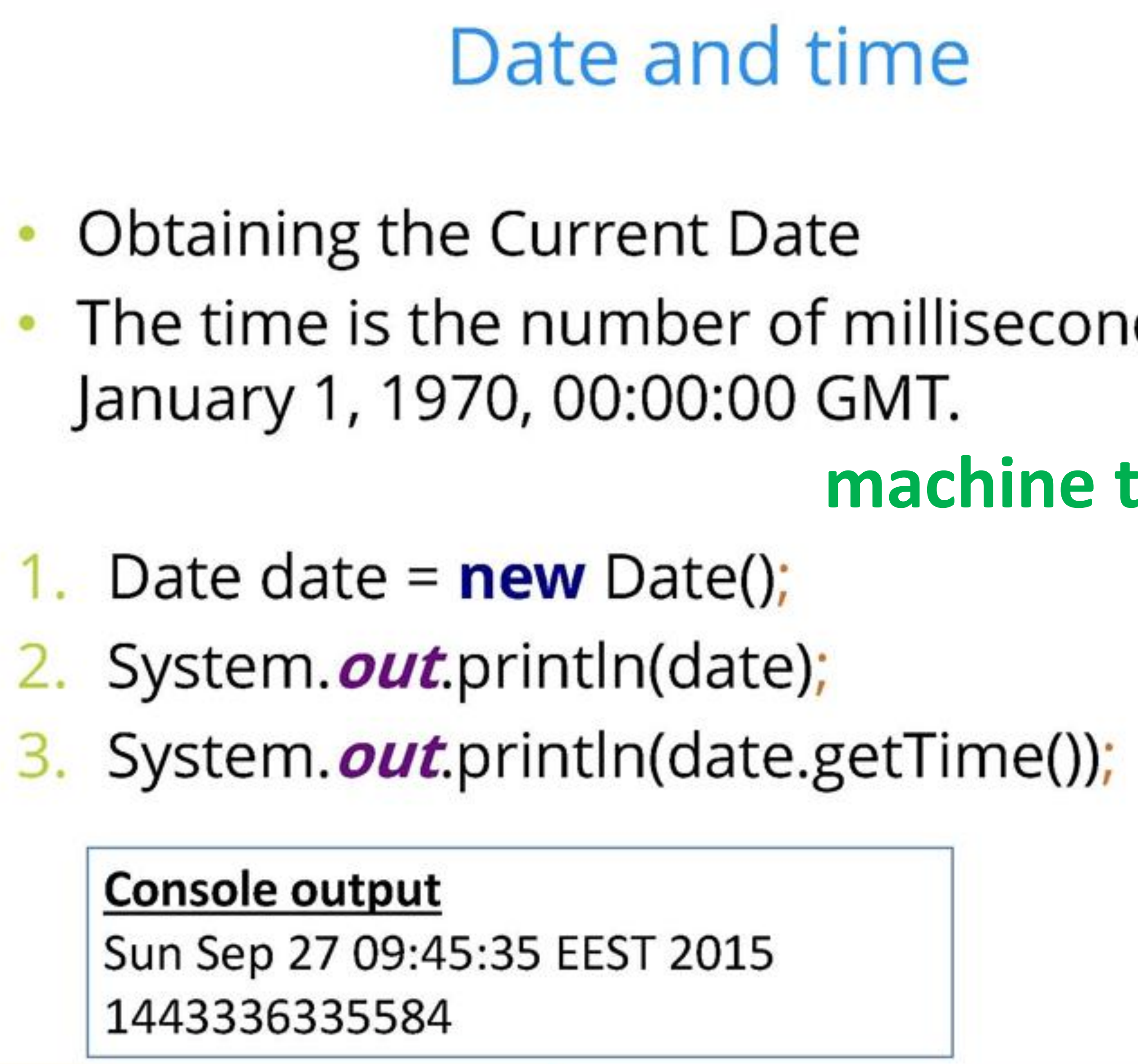
Core Java classes

- The Math class
- Random Numbers
- BigInteger and BigDecimal classes
- The System class
- The Runtime class
- The Properties class
- Creating a Locale
- Numbers and Currencies
- Date and time

Module contents

Core Java classes

- The Math class
- Random Numbers
- BigInteger and BigDecimal classes
- The System class
- The Runtime class
- The Properties class
- Creating a Locale
- Numbers and Currencies
- Date and time



Date and time

- Obtaining the Current Date
- The time is the number of milliseconds since January 1, 1970, 00:00:00 GMT.

machine time approach

```
Date date = new Date();
System.out.println(date);
System.out.println(date.getTime());
```

Console output

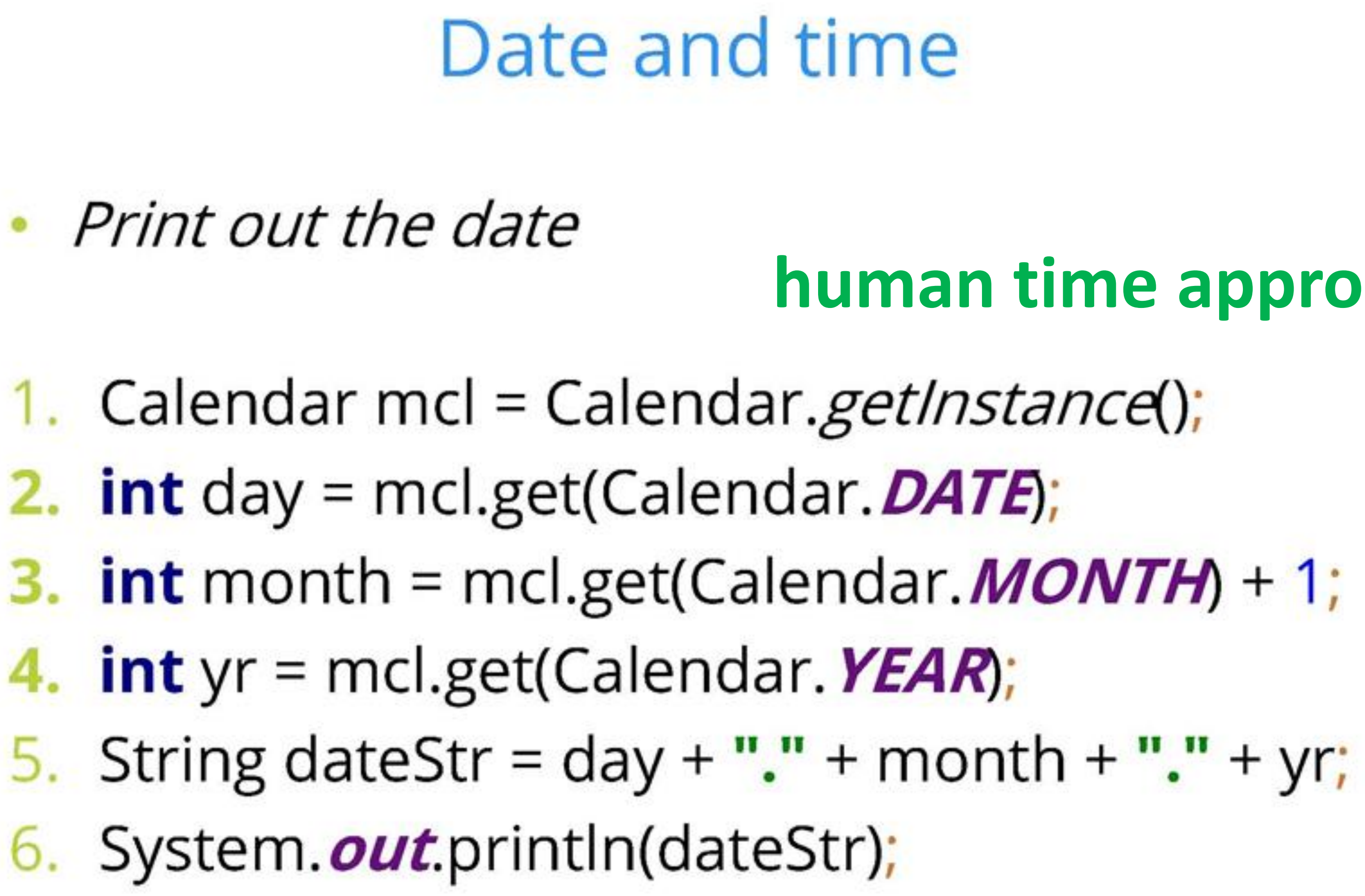
```
Sun Sep 27 09:45:35 EEST 2015
1443336335584
```

java.util.Date

Date and Time

```
public class Date  implements java.io.Serializable, Cloneable, Comparable<Date> {
    private static final BaseCalendar gcal =CalendarSystem.getGregorianCalendar();
    private static BaseCalendar jcal;
    private transient long fastTime;
// Parse 2-digit years within the correct default century
    private static int defaultCenturyStart;
    public Date() {
        this(System.currentTimeMillis());
    }
    /**
     * Initializes a Date object with the specified number of milliseconds since the
     * standard base time known as "the epoch", namely January 1 1970, 00:00:00 GMT.
     */
    public Date(long date) {
        fastTime = date;
    }
...
}
```

See DateDemo



Date and time

- *Print out the date*

```
Calendar mcl = Calendar.getInstance();
int day = mcl.get(Calendar.DATE);
int month = mcl.get(Calendar.MONTH) + 1;
int yr = mcl.get(Calendar.YEAR);
String dateStr = day + "." + month + "." + yr;
System.out.println(dateStr);
```

human time approach

java.util.Calendar

Date and time

- *Print out the time*

```java
Calendar mcl = Calendar.getInstance();
int hour = mcl.get(Calendar.HOUR);
int min = mcl.get(Calendar.MINUTE);
int sec = mcl.get(Calendar.SECOND);
System.out.println(hour + ":" + min + ":" + sec);
```

Date and Time

```
public abstract class Calendar implements Serializable, Cloneable,
                                                    Comparable<Calendar> {

  // The current time is represented in two ways by Calendar: as UTC
  // milliseconds from the epoch (1 January 1970 0:00 UTC), and as local
  // fields such as MONTH, HOUR, AM_PM, etc

  public static final int ERA = 0;
  public static final int YEAR = 1;
  public static final int MONTH = 2;
...
  //A style specifier for getDisplayNames(int, int, Locale) getDisplayNames indicating
  public static final int ALL_STYLES = 0;
  public static final int SHORT = 1;
...
  // 1. Initially, no fields are set, and the time is invalid.
  // 2. If the time is set, all fields are computed and in sync.
  // 3. If a single field is set, the time is invalid.
  // Recomputation of the time and fields happens when the object needs
  // to return a result to the user, or use a result for a computation.
  protected int          fields[];
  protected long          time;
...
```

See CalendarDemo

Formatted date and time output

java.text.DateFormat styles - **int** constants:

SHORT	13.01.23 10:24
MEDIUM (default)	13 січ. 2023 р., 10:24:18
LONG	13 січня 2023 р. 10:24:18 EET
FULL	п’ятниця, 13 січня 2023 р. 10:24:18 за східноєвропейським стандартним часом

```
Date today = new Date();
DateFormat dtf = DateFormat.getDateTimeInstance(int style);
DateFormat df = DateFormat.getDateInstance(int style);
DateFormat tf = DateFormat.getTimeInstance(int style);
System.out.println(dtf.format(today));
```

See DateFormat

Formatted date and time output

java.util.Calendar styles - **int** constants:

NARROW_FORMAT	J
SHORT	Jan.
SHORT_STANDALONE	Jan
LONG	Januar

You can use _SANDALONE styles to avoid dot printing at the end

```
Calendar cal = Calendar.getInstance();
Locale locale = Locale.GERMANY;
System.out.println(cal.getDisplayName(Calendar.MONTH,
                                     Calendar.LONG, locale));
```

See DateFormat

Java Date and Calendar issues

- Date is mutable. It's possible to change the Date instance by client
  without the class-owner of that instance knowing;
- Date constructor parameters are mistakeable (year, month);
- We get timezone using String instead of Enum so we can mistake without exception throw.
- Not intuitive Calendar instance constructing including TimeZone.
- DateFormat cannot be applied to Calendar instance (and SimpleDateFormat is thread-unsafe).

See DateCalendarIssues

Java Date-Time API

- **Java Date-Time API** was introduced in the Java SE 8 (package **java.time**)
- **Java Date-Time API** based on the ISO-8601 *Data elements and interchange formats - Representation of dates and times,* that uses the de facto world *Gregorian calendar*.
- **Java Date-Time API** uses **java.time.chrono** package tools for alternative calendar systems (*Japanese Imperial* or *Thai Budd*hist or you can create your own).
- **Java Date-Time API** uses the *Unicode Common Locale Data Repository (CLDR)* with the world's largest collection of locale data available and the *Time-Zone Database (TZDB)*
- Most of the Date-Time API classes create immutable (and thread-safe) objects.
- The Date-Time API provides a fluent interface, making the code easy to read.

Since JDK 8

Java Date-Time API Temporal classes

The **java.time** package provides the main classes for dealing with dates and times (we call temporal objects for these classes instances):

- **LocalDate** - represents a date (year, month, day).
- **LocalTime** - represents time in a 24-hour day (hour, minute, second, nanosecond).
- **LocalDateTime** - represents the date and time combined, in terms of both date and time fields. A date-time has no time zone.
- **ZonedDateTime** - represents the date-time with a time zone.
- **Instant** - represents a measurement of time starting from an epoch (Jan 1, 1970 00:00:00 GMT). An instant values store as a *long*-type seconds and *int*-type nanoseconds, both can be a negative value.
- **Period** - represents the difference between two dates in years, months, and days. It related to date timeline and timezones. It can be negative.
- **Duration** - represents the difference between two times in seconds and nanoseconds. It does not related to date timeline and timezones. It can be negative.

java.time package

Java Date-Time API Basic TemporalClasses

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

* with nanosecond precision,

** has methods to provide time in these units.

*** daylight saving time or other local time differences are observed.

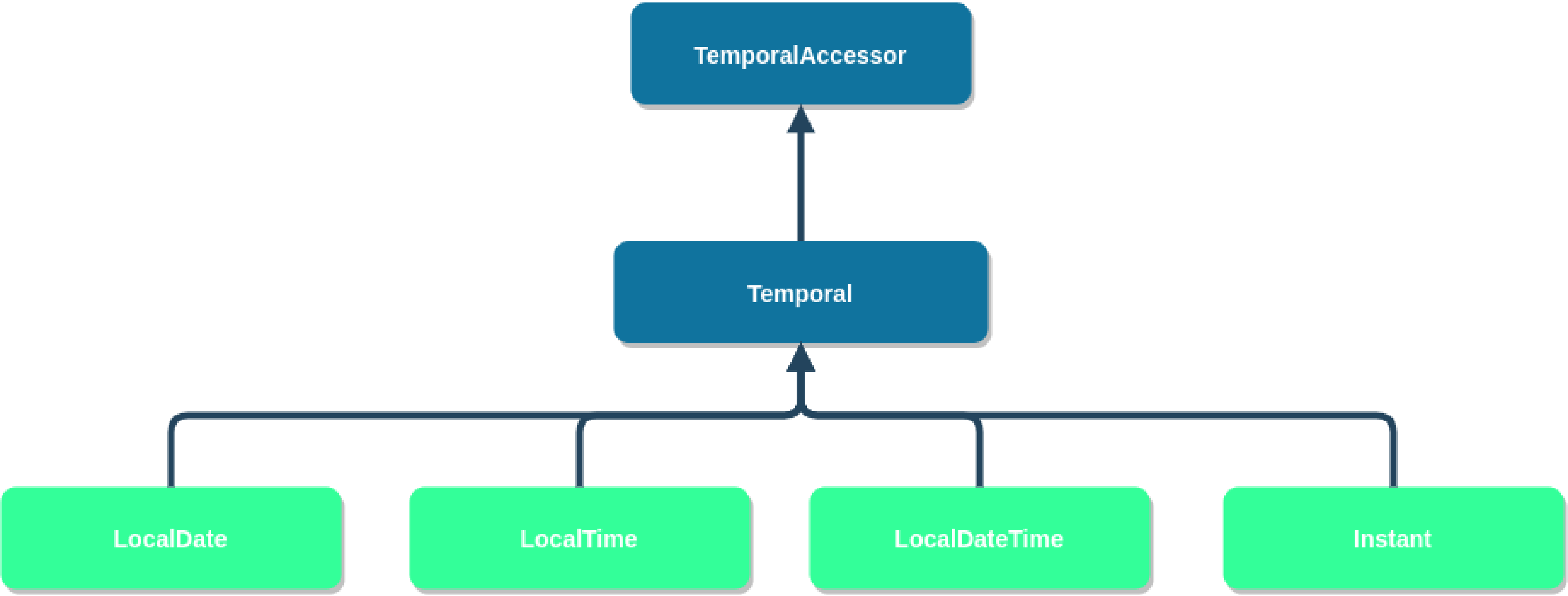
Throwing java.time.DateTimeException indicates problems with creating, querying and manipulating date-time objects.

Java Date-Time API Supplement Classes

Class or Enum	Year	Month	Day	Hours	Minutes	Seconds*	Zone Offset	Zone ID	toString Output
DayOfWeek**			✓ ✓						FRIDAY
MonthDay		✓	✓						--08-20
Month**		✓							AUGUST
YearMonth	✓	✓							2013-08
Year	✓								2013
OffsetTime				✓	✓	✓	✓		08:16:26.957-07:00
OffsetDateTime	✓	✓	✓	✓	✓	✓	✓		2013-08-20T08:16:26.954-07:00
ZoneId								✓	Europe/Paris
ZoneOffset							✓		+02:00

* Seconds are captured to nanosecond precision.
* This is enum

Java Date-Time API main classes hierarchy



All temporal classes common methods

Method	Use
of(int year, int month, ...)	**static**. Construct temporal object from provided temporal fields.
int **get**(TemporalField field)	Return a specific field of this temporal object.
boolean **isSupported**(TemporalField field)	Check specific property of this temporal object.
minus(long val, ChronoUnit unit) **minusXXX**(long val)	Returns a copy of the target object after subtracting an amount of time.
plus(long val, ChronoUnit unit) **plusXXX**(long val)	Returns a copy of the target object after adding an amount of time.
atTime(int hour, int minute, ...)	Create a new temporal object by combining this temporal object and another temporal object. Not provided by ZonedDateTime class.
with(TemporalField field, long newVal) **withXXX**(long newVal)	Create a copy of this temporal object with one field modified.
toXXX()	Convert this temporal object to another type.

XXX can be a specific field, a specific unit or a class name

Additional common methods for LocalDate, LocalTime, LocalDateTime, ZonedDateTime, Instant

Method	Use
now()	**static**. Obtain the current date and/or time from the system clock in the default or specified time zone.
from(TemporalAccessor temporal)	**static.** Obtain a specialized instance of this temporal class from temporal class instance-parameter.
long **until**(Temporal endExclusive, TemporalUnit unit)	Calculates the amount of time until another date-time in terms of the specified unit.
parse(CharSequence text, DateTimeFormatter formatter)	**static.** Obtain a temporal instance from a specified text string.
String **format**(DateTimeFormatter formatter)	Create a text representation of this temporal object using a specified formatter. Instant class does not provide this method.
boolean **isEqual/isBefore/ isAfter**(ChronoLocalDateTime<?> other)	**default.** See LocalDateTimeDemo See DateSupplementClassesDemo See EnumsDemo

Time Zone and Offset Classes

- A *time zone* is a region of the Earth where the same standard time is used. The time observed in a region is usually referred to as the *local time*.
- Each time zone is described by an *identifier* and usually has the format *region/city* (Asia/Tokyo) and an *offset* in time from Greenwich/UTC time (Coordinated Universal Time).
- Java uses the IANA *Time Zone Database* (TZDB) maintained by the Internet Assigned Numbers Authority (IANA) that updates the database regularly, in particular, regarding changes to the rules for DST practiced by a time zone.
- GMT (Greenwich Mean Time) has zero offset from UTC/Greenwich (UTC+0), thus the two are often used as synonyms, for example, GMT-4 is equivalent to UTC-4. However, *GMT is a time zone*, whereas *UTC is a time standard*.

Time Zone and Offset Classes

- The following three classes in the **java.time** package are important when dealing with date and time in different time zones and daylight saving time (DST):
  ZoneId
  ZoneOffset
  ZonedDateTime

See TimeZoneClassesDemo

Period

- For representing an amount of time, the Date and Time API provides the two classes **Period** and **Duration**.
- A **Period** uses date-based amount of time (years, months, days) so it can be used with LocalDate, LocalDateTime and ZonedDateTime classes.
- The total period of time is represented by all three units together: months, days, and years.
- A **Period** of one day, when added to a ZonedDateTime, may vary according to the time zone. For example, if it occurs on the first or last day of daylight saving time.
- The **Period** class provides various get methods, such as getMonths, getDays, and getYears, so that You can extract the amount of time from the period.

See PeriodDemo

Duration

- A **Duration** implements a time-based amount of time in terms of *seconds* and *nanoseconds*, using a *long* and an *int* value for these time units, respectively.
- A **Duration** instance can represent an amount of time in terms of *days*, *hours*, and *minutes*. As these time units have fixed lengths, it makes interoperability between these units possible.
- The **Duration** class can be used with the LocalTime and LocalDateTime classes.
- A **Duration** is most suitable in situations that measure machine-based time, such as code that uses an Instant object.

See DurationDemo

Instant Class

- An **Instant** object represents a point on the timeline, measured with nanosecond precision from a starting point or origin that is, January 1, 1970, at midnight GMT - and is called *epoch*.
- **Instants** before the epoch have negative values, whereas instants after the epoch have positive values.
- An **Instant** object is modeled with two values:
  • A long value to represent the epoch-second
  • An int value to represent the nano-of-second (0 - 999_999_999)
- The **Instant** class can be used for representing computer time, specially timestamps that identify to a higher precision when an event occurred on the timeline. Instants are suitable for persistence purposes - for example, in a database.

See InstantDemo

Temporals Converting from/to Legacy Date & Calendar

From	To	Comments
java.util.Date	java.time.LocalDateTime	The Date **toInstant()** and **LocalDateTime.ofInstant(Instant instant, ZoneId zone)** methods
java.time.LocalDateTime	java.util.Date	The **Date.from(Instant instant)** and LocalDateTime **atZone(ZoneId zone)** and ZonedDateTime **toInstant()** methods
java.util.GregorianCalendar	java.time.ZonedDateTime	The **Calendar.getInstance()** and **ZonedDateTime.ofInstant(Instant instant, ZoneId zone)** methods
java.time.ZonedDateTime	java.util.GregorianCalendar	The **GregorianCalendar.from(ZonedDateTime zdt)** method

- An object of the java.util.Date represents time, date, and time zone. It can be converted to java.time.Instant object.
- Also we can create the java.util.Date object from the java.time.Instant object.

See LegacyCodeConvertDemo