Date and time

Date and time

As a developer, you'll often need to work with dates and times. This could be one way to do it:

```
int year = 2014;
int month = 9;
int day = 23;
int hour = 13;
...
```

As you can probably imagine, this would quickly become **unmanageable**.

Luckily, we have some classes dedicated to handling this for us: **LocalDate, LocalTime** and **LocalDateTime**.

Working with date and time

Creating a new object is simple, and can be done in a range of ways. Most often, you will use one of the **of methods**.

The most common to use are:

```
// 1918/11/11
LocalDate aDate = LocalDate.of(1918, 11, 11);

// 1918/11/11 11:00:00
LocalDateTime anotherDate =
              LocalDateTime.of(1918, 11, 11, 11, 00, 00);

// 1919/11/11 14:00:00 300ms
LocalDateTime thirdDate =
              LocalDateTime.of(1918, 11, 11, 14, 00, 00, 300);
```


Working with date and time

We can also create common dates using the **now** methods:

```
// Today's date (no time)
LocalDate date = LocalDate.now();

// The current time (no date)
LocalTime time = LocalTime.now();

// The current date and time
LocalDateTime now = LocalDateTime.now();
```

Working with date and time

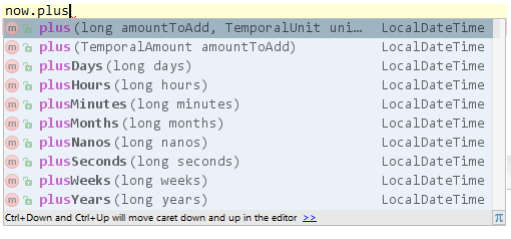
We can modify dates using the **Minus** and **Plus** methods.

```
// Create a date
LocalDateTime now = LocalDateTime.now();

// Two months in the future
LocalDateTime twoMonthsLater = now.plusMonths(2);

// Twelve days behind
LocalDateTime twelveDaysBehind = now.minusDays(12);

// Twelve hours ahead
LocalDateTime middayOnDate = now.plusHours(12);
```



Working with date and time

We can combine date creation with the **add** methods to create dates that are **relative** to other dates, such as tomorrow or next year.

```
// Create a date
LocalDateTime now = LocalDateTime.now();

// Next year
LocalDateTime nextYear = now.plusYears(1);

// One day before the moon landing
LocalDate beforeMoonLanding = LocalDate.of(1969, 7, 20).minusDays(1);
```

```
2016-08-04T11:16:13.096
2017-08-04T11:16:13.096
1969-07-19
```

Working with date and time - duration

LocalDateTime represents both Date and Time.

If we only want to represent time intervals, we can use **Duration**. We can add Duration objects to LocalDateTime.

```
// 2014/09/19 14:45:00
LocalDateTime now = LocalDateTime.of(2014, 9, 19, 14, 45, 00);

// 4 days, 10 hours, 30 minutes
Duration dur = Duration.ofDays(4).plusHours(10).plusMinutes(30);

// 23/09/2014 15:15:00 (added 4 days, 10 hours, 30 minutes)
LocalDateTime future = now.plus(dur);

// 15/09/2014 14:15:00 (subtracted 4 days, 10 hours, 30 minutes)
LocalDateTime past = now.minus(dur);
```

Working with date and time - properties

The **LocalDateTime** class contains a large set of properties to access various facts about a date:

```
LocalDateTime now = LocalDateTime.of(2014, 2, 1, 12, 30, 15);

System.out.println(now.getYear());
System.out.println(now.getMonthValue());
System.out.println(now.getDayOfMonth());
System.out.println(now.getHour());
System.out.println(now.getMinute());
System.out.println(now.getSecond());
System.out.println(now.getNano());
System.out.println(now.getDayOfWeek());
System.out.println(now.getDayOfYear());
System.out.println(now.getMonth());
```

```
2014
2
1
12
30
15
0
SATURDAY
32
FEBRUARY
```

Comparing date and time

You'll often want to **compare** two date instances with each other.

```
LocalDate date1 = LocalDate.of(2016,9,3);
LocalDate date2 = LocalDate.of(2016,9,2);

if(date1.isAfter(date2)){
    System.out.println("Date1 is after Date2");
}

if(date1.isBefore(date2)){
    System.out.println("Date1 is before Date2");
}

if(date1.equals(date2)){
    System.out.println("Date1 is equal Date2");
}
```

Formatting DateTime

If we want to have a string representing a **LocalDateTime** object, we can just call one of many "**To**" methods.

```
LocalDateTime date1 = LocalDateTime.of(2016, 9, 19, 14, 45, 00);

LocalDate date2 = LocalDate.of(2016,9,2);

System.out.println(date1.toLocalDate());
System.out.println(date1.toLocalTime());
System.out.println(date1.toString());
```

```
2016-09-19
14:45
2016-09-19T14:45
```

Formatting DateTime

We can use the **format** method to change how we represent dates:

```
LocalDateTime date1 = LocalDateTime.of(2016, 9, 19, 14, 45, 00);

System.out.println(date1.format(DateTimeFormatter.BASIC_ISO_DATE));
System.out.println(date1.format(DateTimeFormatter.ISO_DATE));
System.out.println(date1.format(DateTimeFormatter.ISO_DATE_TIME));
System.out.println(date1.format(DateTimeFormatter.ISO_TIME));
System.out.println(date1.format(DateTimeFormatter.ISO_LOCAL_TIME));
System.out.println(date1.format(DateTimeFormatter.ISO_LOCAL_DATE));
System.out.println(date1.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));
System.out.println(date1.format(DateTimeFormatter.ISO_ORDINAL_DATE));
System.out.println(date1.format(DateTimeFormatter.ISO_WEEK_DATE));
```

```
20160919
2016-09-19
2016-09-19T14:45:00
14:45:00
14:45:00
2016-09-19
2016-09-19T14:45:00
2016-263
2016-W38-1
```


Formatting DateTime

We can customize this representation by passing a string to DateTimeFormatter

```
System.out.println(date1.format(DateTimeFormatter.ofPattern("yyyy G - dd MMMM hh:mm:ss a")));
```

```
2016 efter Kristus - 02 januari 03:04:05 fm
```

Symbol	Meaning
G	Era
u	Year
y	Year of era
D	Day of year
M/L	Month of year
d	Day of month
Q/q	Quarter of year
Y	Week based year
w	Week of week based year

Symbol	Meaning
W	Week of month
E	Day of week
e/c	Localized day of week
F	Week of month
a	Am/pm of day
h	Clock hour of am/pm (1-12)
K	Hour of am/pm
K	Clock hour of am/pm(1-24)
…	…

More details here:
https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html

Java and Date/Time

In the past we used the classes in the **java.util.Date**, **java.util.Calendar**, and **java.text.SimpleDateFormat**

But they have a lot of problems and issues.

Avoid them and instead use either:

- **java.time.*** package in Java 8
  Recommended for new development, replaces Joda-Time
- **Joda-Time** (for Java <8 applications)
  http://www.joda.org/joda-time/

Exercise 9

Lets do exercise 9