Alma Mater Studiorum-Università di Bologna
Scuola di Ingegneria

Date e gestione del tempo

Corso di Laurea in Ingegneria Informatica

Anno accademico 2021/2022

Proff. ENRICO DENTI & GABRIELE ZANNONI

Dipartimento di Informatica – Scienza e Ingegneria (DISI)

LA GESTIONE DEL TEMPO

- Il tempo è un elemento fondamentale in molte applicazioni
  - giorni, mesi, anni, ore minuti
  - tempo trascorso fra.. / da…
  - concetti relativi vs. assoluti
  - convenzioni culturali diversificate

- Ergo, tutti i principali linguaggi OO hanno un'apposita API (Application Programming Interface, ~libreria) per gestirli
  - definisce i concetti chiave
  - fornisce classi e funzioni per esprimerli e manipolarli
  - offre funzionalità di *formattazione* in grado di adeguarsi alle specifiche *convenzioni culturali* di una data nazione

LA GESTIONE DEL TEMPO NEI VARI LINGUAGGI

Java

- In Java è presente una flessibile *date time API* definita nel package **`java.time`**
  - prima, vecchia API scomoda da usare

C#

- C# ha anch'esso la sua API
  - dettagli: `https://zetcode.com/csharp/datetime/` (approccio e concetti abbastanza simili a Java)
  - esistono API alternative di terze parti `https://nodatime.org/`

Scala

Kotlin

- Scala e Kotlin si appoggiano alla *date time API* di Java
  - Kotlin sta sviluppando la sua API, `kotlinx.datetime`, attualmente in versione 0.2.0 (experimental) `https://github.com/Kotlin/kotlinx-datetime/blob/master/README.md`

LA GESTIONE DEL TEMPO in Java

- Date Time API: **`java.time`**

Java

  - 1° Assunto di base: modello basato su oggetti *immutabili*
    → nulla si modifica dopo essere stato costruito
  - 2° Assunto di base: costruzione oggetti sempre *indiretta*
    → non si espongono costruttori pubblici
    → la costruzione avviene sempre tramite un'entità «*fabbrica*», che maschera e nasconde i costruttori effettivi, secondo il *PATTERN «FACTORY»*

- Concetti
  - Tempo relativo vs. tempo assoluto: Date & Orari *locali* vs. *assoluti*
  - Periodo di tempo («un mese») indipendente dall'effettiva durata
  - Durate («tre mesi e due giorni», «due ore e dieci minuti»)
  - Fusi orari, calendari, …

IL PATTERN FACTORY IN UNA SLIDE

- Quando la costruzione di un oggetto non è semplice, o deve essere tenuta sotto controllo o più in generale è inopportuno sia fatta dall'utente per qualsiasi ragione, la si può delegare / incapsulare in un ente terzo: la *FABBRICA (FACTORY)*
  - la fabbrica *incapsula la new* in un metodo statico, che restituisce un *oggetto adatto* senza dire esattamente *come/cosa* farà
  - permette di *disciplinare e controllare* il processo di costruzione
- In **`java.time`**, ogni classe o enumerativo espone un metodo statico di nome **`of`**
  - per ottenere un nuovo orario o una nuova data, *non* la si costruisce direttamente con new
  - *ce la si fa costruire* dal metodo-fabbrica **`of`**

DATE & TIME FACTORY: ESEMPI

Java

- Alcuni esempi
  - per ottenere la data del giorno di Natale del 2020, *non* si scrive:

```
LocalDate xmas = new LocalDate(2020, 12, 25);
```

NO!

ma si scrive

```
LocalDate xmas = LocalDate.of(2020, 12, 25);
```

  - per ottenere il mese di ottobre o il giorno «lunedì»:

```
Month     m = Month.of(10);     // OCTOBER
DayOfWeek d = DayOfWeek.of(1);  // MONDAY
```

Il metodo **of** è la fabbrica: incapsula e nasconde i costruttori

  - per ottenere l'orario del mezzogiorno locale, o l'orario attuale:

```
LocalTime noon = LocalTime.of(12, 0);
LocalTime now  = LocalTime.now();
```

La factory controlla e disciplina il processo di costruzione: non vorreste due «lunedì», tre «aprile», due «mezzogiorni»,...!

TEMPO RELATIVO VS. ASSOLUTO

- **`java.time`** introduce sia concetti *relativi* sia *assoluti*
  - *nel quotidiano* la vita umana si svolge in un *luogo fisso*, che ha il suo *orario locale* e si esprime sempre *rispetto ad esso*
  - tuttavia a volte occorre ragionare su riferimenti temporali che *non dipendono da dove ci si trova*
- Esempi
  - "Ci vediamo domani alle 10": ci si riferisce *al proprio sistema di riferimento* Altrove nel mondo "domani alle 10" sarà *un orario diverso* e forse anche *una data diversa* → concetto di orario *relativo (locale)*
  - "La teleconferenza inizierà oggi alle 17, ora di New York" o "L'attacco è previsto per le ore 6.00 Zulu" *sono invece esempi di orari assoluti* che non dipendono da dove ci si trova

TEMPO RELATIVO VS. ASSOLUTO

- A volte non è immediato distinguerli: il punto chiave è se quell'orario sia percepito *solo in quel luogo* o *anche altrove*
  - dove può essere in vigore un'ora locale diversa
- Esempio: "L'aereo parte da Bologna alle 11.30 e arriva a Mosca alle 15.45"
  - quell'aereo decolla in quel preciso momento indipendentemente da dove ti trovi tu nel mondo: sono forme concise per orari assoluti
  - sono infatti orari (indirettamente) riferiti al *fuso orario* del Paese
  - l'espressione "parte da Bologna alle 11.30" in realtà specifica:
    - le 11.30 GMT+1 in inverno
    - le 11.30 GMT+2 in estate (ora legale)

IL PACKAGE `java.time`

Java

- Concetti *relativi (locali)*
  - **`LocalDate`**: una data relativa (giorno/mese/anno)
  - **`LocalTime`**: un orario relativo (ore/minuti/secondi)
  - **`LocalDateTime`**: una data + orario relativi
  - **`Period`**: una durata relativa (misurata in giorni, mesi, anni, etc.)
- Concetti *assoluti*
  - **`Instant`**: un punto sulla linea del tempo *espresso in nanosecondi*
  - **`Duration`**: una durata in (nano)secondi *fra due istanti di tempo*
  - **`OffsetDateTime`**: una data assoluta sulla linea del tempo *espressa come data + orario + delta rispetto a Greenwich (UTC)*
  - **`ZonedDateTime`**: una data assoluta sulla linea del tempo *espressa come data + orario + fuso orario (es. CET, GMT-5, ecc.)*

DATE & ORARI

Java

- Data & orario *locale* *(senza fuso orario né offset da UTC)*
  - **`LocalDateTime`** *Giorno, Mese, Anno, Ore, Minuti, Secondi*
    `2017-12-03T10:15:30`
  - **`LocalDate`** *Giorno, Mese, Anno*
    `2017-12-03`
  - **`LocalTime`** *Ore, Minuti, Secondi*
    `10:15:30`
- Data & orario *assoluto* *(con fuso orario oppure offset da UTC)*
  - **`ZonedDateTime`** *Giorno, Mese, Anno, Ore, Minuti, Secondi,…, Time Zone*
    `2017-12-03T10:15:30+01:00 Europe/Rome`
  - **`OffsetDateTime`** *Giorno, Mese, Anno, Ore, Minuti, Secondi,…, Offset UTC*
    `2017-12-03T10:15:30+01:00`

GIORNI & MESI

Java

- Enumerativi per i *giorni della settimana* e *mesi*
  - **`DayOfWeek`**
    Valori leciti: **`MONDAY`**, **`TUESDAY`**, … **`SUNDAY`**
  - **`Month`**
    Valori leciti: **`JANUARY`**, **`FEBRUARY`**,… **`DECEMBER`**

- Entrambi definiscono:

Non **`ordinal()`**

  - il metodo **`getValue`** che restituisce *il valore intero "corrispondente"*
    - ESEMPIO: **`Month.OCTOBER.getValue()`** → 10
    - ESEMPIO: **`DayOfWeek.MONDAY.getValue()`** → 1
  - il metodo statico **`of(int)`** che restituisce *la costante enumerativa "corrispondente" a quel valore intero*
    - ESEMPIO: **`Month.of(10)`** → **`OCTOBER`**
    - ESEMPIO: **`DayOfWeek.of(1)`** → **`MONDAY`**

DATE & ORARI LOCALI (1)

- Una data / orario locale *non è* un punto univoco sulla linea del tempo: modella un concetto *relativo al luogo*

Java

- Risponde a domande come:
  - Qual è la tua data di nascita? ***LocalDate***
  - Qual è il giorno di Natale nel 2017? ***LocalDate***
  - A che ora inizia la lezione? ***LocalTime***
  - A che ora si pranza nel giorno di Natale? ***LocalDateTime***

- Costruzione tramite *factory* →metodo statico **`of(…)`**

```
LocalDate xmas2020 = LocalDate.of(2020, 12, 25);
LocalDate xmas2016 = LocalDate.of(2016, Month.DECEMBER, 25);
LocalTime noon     = LocalTime.of(12, 0);
LocalDateTime xmas2020noon = LocalDateTime.of(xmas2020,noon);
```

DATE & ORARI LOCALI (2)

- La stampa riflette la relatività del concetto: non ci sono elementi «assoluti» *che collochino quella data / orario in un punto preciso* del globo

Java

```
LocalDate xmas2020 = LocalDate.of(2020, 12, 25);
LocalDate xmas2016 = LocalDate.of(2016, Month.DECEMBER, 25);
LocalTime noon     = LocalTime.of(12, 0);
LocalDateTime xmas2020noon = LocalDateTime.of(xmas2020,noon);
```

```
2020-12-25
2016-12-25
12:00
2020-12-25T12:00
```

DATE & ORARI LOCALI (3)

Java

- Metodi *accessor* per ottenere
  - da ***LocalDate***
    - il giorno del mese (**`getDayOfMonth`**)
    - il mese (**`getMonth`**)
    - l'anno (**`getYear`**)
  - da ***LocalTime***
    - l'ora (**`getHour`**)
    - i minuti (**`getMinute`**)
    - i secondi (**`getSecond`**)
    - i nanosecondi (**`getNano`**)
- Altri metodi per ottenere *informazioni più complesse:*
  - il giorno dell'anno (**`getDayOfYear`**)
  - il giorno della settimana (**`getDayOfWeek`**)
  - verificare se è un anno bisestile (**`isLeapYear`**)
  - …

LAVORARE CON DATE LOCALI

Java

- È definita un'aritmetica delle *date*
  Partendo da una data, si possono ottenere altre date:
  - *sommando* ad essa *giorni, mesi, anni* → metodi **`plus*`**
    - **`plusDays, plusMonths, plusWeeks, plusYears`**
  - *sottraendo* da essa *giorni, mesi, anni* → metodi **`minus*`**
    - **`minusDays, minusMonths, minusWeeks, minusYears`**
  - *cambiando* in essa il *giorno o il mese o l'anno* → metodi **`with*`**
    - **`withDayOfMonth, withDayOfYear, withMonth, withYear`**

- ESEMPIO

```
LocalDate xmas2016 = …;
LocalDate xmas2020 = xmas2016.plusYears(4);
LocalDate xmas2018 = xmas2016.withYear(2018);
```

```
2016-12-25
2020-12-25
2018-12-25
```

LAVORARE CON ORARI LOCALI

Java

- Analogamente è definita un'aritmetica degli *orari*
  Partendo da un orario, si possono ottenere altri orari:
  - *sommando* ad essa *ore, minuti, settimane, anni* → metodi **`plus*`**
    - **`plusHours, plusMinutes, plusWeeks, plusYears`**
  - *sottraendo* da essa *ore, minuti, settimane, anni* → metodi **`minus*`**
    - **`minusHours, minusMinutes, minusWeeks, minusYears`**
  - *cambiando* in essa l'*ora,* i *minuti* o i *secondi* → metodi **`with*`**
    - **`withHours, withMinutes, withSeconds`**
- ESEMPIO

```
LocalTime noon = LocalTime.of(12,0);
LocalTime whatTime = noon.plusMinutes(10)
                         .minusHours(13)
                         .withSeconds(10);
```

Fluent interface!

Che ore sono?

Period

- Un lasso di tempo misurato in *anni, mesi, giorni*
  volutamente privo di una precisa specifica di durata
  - modella una durata *relativa* (un «anno», un «mese»)
  - ma non specifica *esattamente quanto dura* perché anni e mesi *non hanno sempre la medesima durata*
    - quanti giorni dura un «anno»? 365 ? 366 ?
    - quanti giorni ci sono in un «mese»? 30, 31, 28, 29..?

Java

Molte situazioni si basano su tali concetti: lo stipendio *mensile*, l'incasso *annuale*..

- Anche in questo caso:
  - creazione sempre e solo tramite *factory*, con `of*` e `between`
  - elaborazione tramite metodi `plus*`, `minus*`, `with*`
- ESEMPIO

```
Period p1 = Period.ofMonths(2)
                  .plusDays(3);
```

«Due mesi e tre giorni»

P2M3D

Period

Java

- Metodi *accessor* per ottenere
  - il numero di giorni (**`getDays`**), mesi (**`getMonths`**), anni (**`getYears`**)
- Metodo *statico factory* (**`between`**) per ottenere un **`Period`** come *differenza fra due date*

```
Period p1 = Period.ofMonths(2).plusDays(3);
Period p2 = Period.between(xmas2016, xmas2017);
```

```
P2M3D
P1Y
```

- È definita anche una *aritmetica dei periodi*
  - si può calcolare una nuova data *sommando* (***`addTo`***) *o sottraendo* (***`subtractFrom`***) *un periodo* da una data assunta come riferimento

```
LocalDate d1 = (LocalDate)p1.addTo(xmas2017);
LocalDate d2 = (LocalDate)p1.subtractFrom(xmas2017);
```

Occhio al cast…!

```
2018-02-28
2017-10-22
```

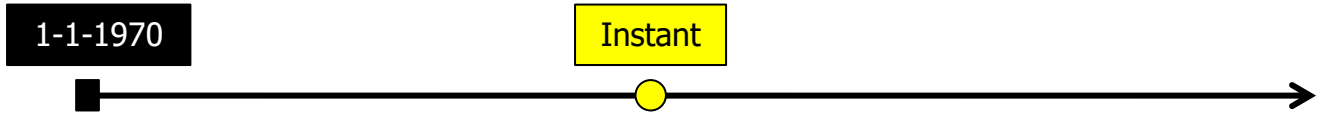
`Period`: perché quel cast?

- In un design pulito, stupisce trovare dei cast!
- Motivo:
  - quei metodi devono poter operare su tutti gli oggetti «temporali», non solo su **`LocalDate`**
  - quindi, il tipo di ritorno «generico» va «aggiustato» al tipo effettivo:
    se entra **`LocalDate`**, esce un **`LocalDate`**;
    se entra **`LocalDateTime`**, esce **`LocalDateTime`**;
    etc.
  - lo capiremo meglio parlando di ereditarietà…

Instant

Java

- Un punto sulla linea assoluta del tempo
  - tecnicamente, il numero di *nanosecondi trascorsi dalla mezzanotte del 1 Gennaio 1970, ora di Greenwich (UTC)*



- È un concetto fisico, non umano
  - non dipende dalla località
  - non ha il concetto di fuso-orario (zona)
  - viene istanziato tramite *metodi factory opportuni*
- L'istante corrente:

```
Instant adesso = Instant.now();
```

```
2019-02-13T17:06:14.654704200Z
```

Metodo factory

Ore 18:06 italiane (17:06 Zulu) del 13/2/2019

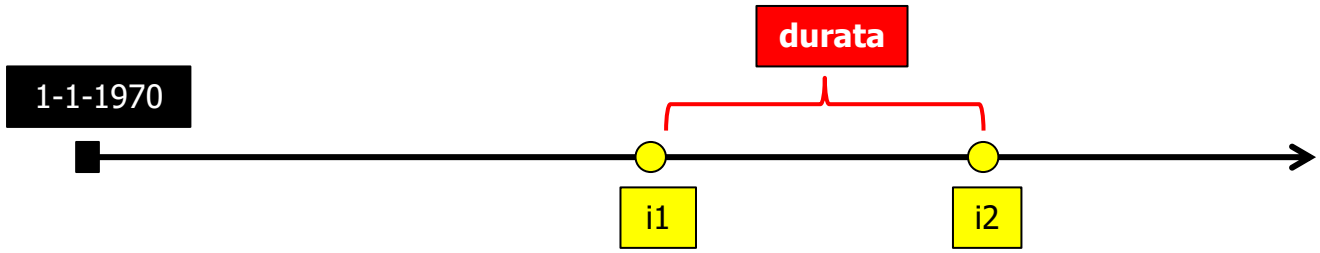
`Instant` – specifica

- In quanto concetto fisico, l'istante di tempo *non ha nulla a che fare* con concetti «umani» come giorni della settimana, mesi, anni
  - tali concetti sono dipendenti dalla *cultura umana locale*
- L'insieme di regole umane in uso in una certa cultura per lo scorrere del tempo definisce un *calendario*
  - esistono nel mondo parecchi *calendari diversi:* *gregoriano*, giuliano, ebraico, islamico…
  - elementi comuni: sono tutti basati su *giorni, ore, minuti, secondi* perché la rotazione della Terra su se stessa (→ giorno) e intorno al sole (→ anno) sono *fatti fisici* uguali per tutti
  - differenze: *tutte quelle basate sulla cultura* anziché sulla fisica, ovvero nomi dei giorni, divisione in mesi, numerazione anni, ecc.

Duration

- Un lasso di tempo misurato in (nano)secondi
  - tipicamente: differenza fra due istanti di tempo

Java



- Esempio:

```
Instant i1 = Instant.now();
doSomething();
Instant i2 = Instant.now();
Duration d = Duration.between(i1,i2);
```

```
2019-02-13T17:15:53.3019…

2019-02-13T17:15:53.3487…
PT0.0468001S      (Eclipse)
```

La durata dipende dall'ambiente di esecuzione:
Eclipse e JDK da terminale sono *diversi!*

```
PT0.0468001S
PT0.0312001S
```

`Duration` – specifica

Java

- Creazione sempre e solo tramite *factory*
  - metodi base della forma **`of*`** che partono da valori interi
- Elaborazione tramite normali metodi di istanza
  - metodi della forma **`plus*`**, **`minus*`**, **`with*`** che *producono una nuova* ***`Duration`*** sommando, sottraendo o cambiando uno o più elementi da una **`Duration`** esistente
- Esempio base (verboso):

```
Duration d1 = Duration.ofDays(1);
Duration d2 = d1.plusHours(3);
Duration d3 = d2.minusMinutes(4);
Duration  d = d3.minusSeconds(10);
```

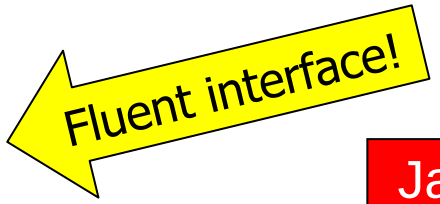
```
PT26H55M50S
```

Durata di *26 ore + 55 minuti + 50 secondi*

`Duration` – specifica

- Però, doversi appoggiare a istanze temporanee (inutili), quando in realtà interessa solo l'ultima, è molto scomodo
- Approccio *fluent interface:* i metodi si possono combinare semplicemente *invocandoli uno dietro l'altro*
  - ognuno restituisce la duration modificata.. furbo!
  - si usa scriverli "in colonna", andando a capo, per leggibilità
- Esempio rivisto:

```
Duration dx = Duration.ofDays(1)
                      .plusHours(3)
                      .minusMinutes(4)
                      .minusSeconds(10);
```



Java

`Duration` – specifica

- La classe offre metodi per estrarre il numero *totale* di
  - giorni `toDays`
  - ore `toHours`
  - minuti `toMinutes`
  - millisecondi `toMillis`

Java

- *troncati* all'unità di misura indicata

Durata di *1 giorno + 2 ore + 55 minuti + 50 secondi*

- *toHours = 26*
- *toMinutes = 26*60+55 = 1615*
- *toMillis = (1615*60+50)*1000 = 96950 * 1000*

```
1
26
1615
96950000
```

Durata di *1 giorno - 10 secondi = 23h59m50s*

```
0
23
1439
86390000
```

DATE & ORARI ASSOLUTI (1)

- Una data o orario assoluto ha lo stesso contenuto informativo di **`Instant`** ma *interpretato con il nostro calendario*
  - *calendario gregoriano* in vigore dal 1582
  - non usato in tutti i paesi (alcuni adottano ancora quello giuliano..)
- Una data & orario assoluto si costruisce *aggiungendo a un* ***`LocalDateTime`*** *una indicazione che la renda assoluta*
  - **`OffsetDateTime`**: offset assoluto rispetto a UTC
  - **`ZonedDateTime`**: identificativo del fuso orario

Java

- Risponde a domande del tipo:
  - quando iniziano le lezioni del 2° ciclo a Bologna nel 2019?
  - a che ora parte il volo EJ842 ?
  - che ore sono adesso a New York?

DATE & ORARI ASSOLUTI (2)

- Esempio
  - quando iniziano le lezioni del 2° ciclo a Bologna nel 2021?

Java

```
LocalDateTime inizio =
    LocalDateTime.of( 2021, 2, 17, 9, 0);
```

```
2021-02-17T09:00
```

```
OffsetDateTime offsInizio =
    OffsetDateTime.of(inizio,
                      ZoneOffset.ofHours(1));
```

```
2021-02-17T09:00
+01:00
```

```
ZonedDateTime  zInizio =
    ZonedDateTime.of( inizio,
                      ZoneId.of("CET"));
```

```
2021-02-17T09:00
+01:00[CET]
```

Si appoggia sullo *standard internazionale* per esprimere i fusi orari

DATE & ORARI ASSOLUTI (3)

Java

- Esempio
  - a che ora è partito il volo EJ842 il 7 ottobre 2017 ?

- con **ZonedDateTime**:
  -
```
ZonedDateTime departure =
        ZonedDateTime.of(2017, Month.OCTOBER.getValue(), 7,
                20, 30, 10, 0, ZoneId.of("CET"));
```

LocalDateTime

Nanosecondi!

- con **OffsetDateTime**:
  -
```
OffsetDateTime departure =
        OffsetDateTime.of(2017, 10, 7, 20, 30, 10, 0,
                ZoneOffset.ofHours(1));
```

Nanosecondi!

LocalDateTime

DATE & ORARI ASSOLUTI (4)

Java

- In dotazione il solito "kit"…
  - metodi *accessor* per recuperare anno (**`getYear`**), mese (**`getMonth`**) e giorno (**`getDayOfMonth`**)
  - aritmetica per sommare (**`plus*`**), sottrarre (**`minus*`**), cambiare (**`with*`**) anni, mesi e giorni
- …più alcuni metodi specifici per *trasformare* e *convertire*
  - una data & orario assoluta in `Instant` → `toInstant`
  - un `OffsetDateTime` in `ZonedDateTime` → `toZonedDateTime`
  - un `ZonedDateTime` in `OffsetDateTime` → `toOffsetDateTime`

ESEMPIO: QUANTO MANCA AL COMPLEANNO

- Esempio: quanto manca al prossimo compleanno?

Java

```
private static Period toNextBirthDay(LocalDate dateOfBirth) {
    LocalDate today = LocalDate.now();
    int currentYear = today.getYear();
    LocalDate nextBirthDay = dateOfBirth.withYear(currentYear);
    if (nextBirthDay.isBefore(today)) {
        nextBirthDay = dateOfBirth.withYear(currentYear + 1);
    }
    return Period.between(today, nextBirthDay);
}
```

```
Esempio di run:
Al prossimo compleanno: P1M30D
```

RIASSUMENDO…

- L'API **`java.time`** è costruita su *precisi pattern* Java
  - *metodi statici factory* **`of(…)`**, **`of*(…)`**
    per creare gli oggetti partendo da opportuni dati di partenza
  - metodi **`to*`**
    per convertire da un mondo a un altro, e precisamente:
    - se l'oggetto di arrivo *non richiede più informazioni di quello di partenza, il metodo non ha alcun parametro*

```
ZonedDateTime zdt = …;
OffsetDateTime odt = zdt.toOffsetDateTime();
LocalTime lt = zdt.toLocalTime();
```

    - se l'oggetto di arrivo *richiede più informazioni di quello di partenza, il metodo ha come parametri le informazioni necessarie (mancanti)*

```
LocalDateTime ldt = …;
Instant myInstant = ldt.toInstant(ZoneOffset.ofHour(1));
```

RIASSUMENDO…

Java

- (segue API **`java.time`**)
  - *metodi accessor di istanza* **`get*`**
    per recuperare i dettagli dell'entità
  - metodi *di confronto* **`isAfter`**, **`isBefore`**, **`isEqual`**
    fra istanze dello stesso tipo

```
OffsetDateTime dt1 = OffsetDateTime.of(…);
OffsetDateTime dt2 = OffsetDateTime.of(…);
if (dt2.isBefore(dt1)) { … }
```

  - metodi per *ottenere nuove istanze* **`plus*`**, **`minus*`**, **`with*`**, :
    - `plus*`: aggiunge *unità di tempo*
    - `minus*`: sottrae *unità di tempo*
    - `with*`: modifica *una determinata unità di tempo*

    sempre restituendo una nuova istanza modificata

ESPERIMENTI AL VOLO

- Usando Jshell, costruiamo alcuni orari locali e accediamo alle relative proprietà. Calcoliamo anche alcune durate:

Java

```
jshell> import java.time.*

jshell> LocalTime t1 = LocalTime.now()
t1 ==> 09:32:32.378172900

jshell> t1.getHour()
$3 ==> 9

jshell> t1.getMinute()
$4 ==> 32

jshell> t1.getSecond()
$5 ==> 32

jshell> t1.getNano()
$6 ==> 378172900
```

```
jshell> LocalTime t2 = LocalTime.now()
t2 ==> 09:33:18.988455500

jshell> Duration.between(t1,t2)
$8 ==> PT46.6102826S

jshell> LocalTime t2 = LocalTime.now()
t2 ==> 09:33:44.126100

jshell> Duration.between(t1,t2)
$10 ==> PT1M11.7479271S
```

ESPERIMENTI AL VOLO

- Analogamente, costruiamo alcuni date locali e verifichiamo l'aritmetica:

```
jshell> LocalDate d1 = LocalDate.now()
d1 ==> 2020-03-23

jshell> LocalDateTime ld1 = LocalDateTime.of(d1,t1)
ld1 ==> 2020-03-23T09:32:32.378172900

jshell> ld1.plusDays(3)
$13 ==> 2020-03-26T09:32:32.378172900

jshell> ld1.plusDays(10)
$14 ==> 2020-04-02T09:32:32.378172900
```

Java

- Idem per alcune date&orari locali:

```
jshell> LocalDateTime ld1 = LocalDateTime.of(d1,t1)
ld1 ==> 2020-03-23T09:32:32.378172900

jshell> ld1.plusDays(3)
$13 ==> 2020-03-26T09:32:32.378172900

jshell> ld1.plusDays(10)
$14 ==> 2020-04-02T09:32:32.378172900
```

ESPERIMENTI AL VOLO

- Ora un esperimento più interessante

Java

  - prima costruiamo alcuni date&orari assoluti
  - poi, applichiamo aritmetica per verificare il fatto che *a fine marzo entri in vigore l'ora legale*

```
jshell> ZonedDateTime z1 = ZonedDateTime.now()
z1 ==> 2020-03-23T09:39:25.776905700+01:00[Europe/Berlin]

jshell> z1.plusDays(1)
$16 ==> 2020-03-24T09:39:25.776905700+01:00[Europe/Berlin]

jshell> z1.plusDays(10)
$17 ==> 2020-04-02T09:39:25.776905700+02:00[Europe/Berlin]

jshell> var z2 = z1.plusDays(10)
z2 ==> 2020-04-02T09:39:25.776905700+02:00[Europe/Berlin]

jshell> Duration.between(z1,z2)
$19 ==> PT239H
```

  - giustamente, aggiungendo 10 giorni il fuso orario cambia *e la durata fra le due date&orari è di 239 ore (non 240)*

CONVENZIONI DI FORMATTAZIONE

- Le convenzioni per date e orari non sono universali: *ogni paese e ogni cultura ha le proprie*
  - nomi dei giorni della settimana
  - nomi dei mesi
  - ordine in cui compaiono (gg/mm/aa, mm/gg/aa…)
  - orari su 12 o 24 ore, mezzogiorno/mezzanotte
  - separatori ammessi
- Per questo esistono appositi *formattatori* che incapsulano e applicano il concetto *cultura locale*
  - insiemi di regole per stampa e parsing di numeri, valute, date, orari

CONVENZIONI DI FORMATTAZIONE

- Esistono *database internazionali* che raccolgono tali convenzioni
  - in continua evoluzione
- Fino a Java 8, *Java aveva il proprio database interno*
  - chiamato semplicemente «JRE»

Java

- Da Java 9, si appoggia invece al *database internazionale Unicode CLDR (Common Locale Data Repository)*
  - occhio: alcune scelte differiscono da quelle «classiche» di Java 8
  - alcuni programmi possono funzionare diversamente da Java 9 in poi

E IN C#..?

- Costruzione diretta con **`new`**, *senza factory*, di: C#
  - **`DateTime`**: data + orario relativa (giorno/mese/anno/ore/minuti/secondi)
    - non esiste il solo orario (cioè l'equivalente di **`LocalTime`**)
    - è invece possibile avere una **`DateTime`** con la parte Time azzerata
  - **`TimeSpan`**: una durata relativa (misurata in giorni e sottomultipli)
    - non esistono concetti intenzionalmente generici come **`Period`**
  - **`DateTimeOffset`**: data + orario assoluta sulla linea del tempo *espressa come data + orario + delta rispetto a Greenwich (UTC)*
    - il delta rispetto a UTC si esprime con un **`TimeSpan`**
  - **`DayOfWeek`**: analogo a Java
    - non esiste invece un enumerativo per i mesi (si gestiscono nelle date)
- Sulle date sono definiti metodi (**`Add`**, **`Subtract`**) e operatori (**`<`**, **`>`**, etc.)

C#: ESEMPI

C#

```
DateTime now   = DateTime.Now;
DateTime today = DateTime.Today;
```

Now contiene data e ora
Today contiene solo la data (l'ora è azzerata)

```
DateTime xmas2020 = new DateTime(2020, 12, 25);
DateTime xmas2020noon = new DateTime(2020, 12, 25, 12, 0, 0);

DateTime xmas2021 = xmas2020.AddYears(1);
DateTime xmas2021noon = xmas2021.AddHours(12);

DateTime whatTime = today                  // 19/02/2021 00:00:00
                    .AddHours(12)          // 19/02/2021 12:00:00
                    .AddMinutes(10)        // 19/02/2021 12:10:00
                    .AddHours(-13)         // 18/02/2021 23:10:00
                    .AddSeconds(10);       // 18/02/2021 23:10:10

DateTime utc = DateTime.UtcNow;

DayOfWeek d = (DayOfWeek)1; // Monday
```

Esistono solo metodi **Add**XX
(non *minus*, né *with*)

C#: ESEMPI

C#

```
DateTime startSemester = new DateTime(2021, 2, 17);
int year   = startSemester.Year;
int month  = startSemester.Month;
int day    = startSemester.Day;
int hour   = startSemester.Hour;
int minute = startSemester.Minute;
int second = startSemester.Second;
int millis = startSemester.Millisecond;

TimeSpan p1 = new TimeSpan(63,0,0,0); // 63 giorni
TimeSpan p2 = xmas2021 - xmas2020;    // 365 gg

DateTime primoAprile = xmas2020.Add(new TimeSpan(97,0,0,0));

DateTimeOffset d = new DateTimeOffset(2021, 2, 19, 21, 4, 0,
                                      new TimeSpan(1, 0, 0));
```

DateTime specifica molte proprietà pubbliche (read-only)

Operatore (-)

Add / Subtract con TimeSpan

97 giorni

Fuso orario UTC+1

1 ora