PAU CASESNOVES
Centre Integrat de Formació Professional

Desenvolupament d'aplicacions web

Programació

UT 4.2: Programació Orientada a Objectes

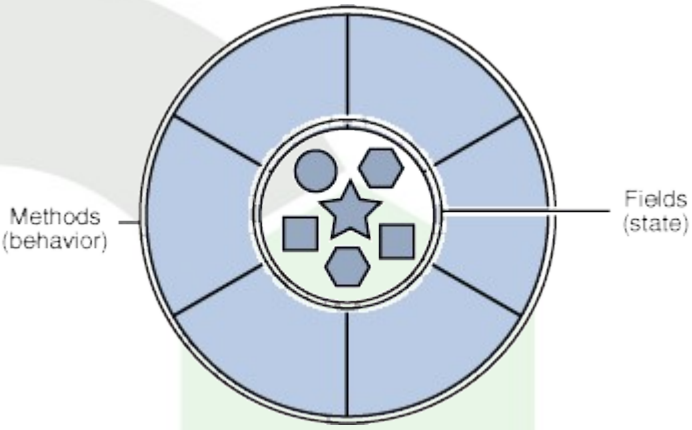
Joan Pons Tugores

Índex de continguts

Objectes

En el mon real tenim clar que és un objecte: un cotxe, un televisor, ... Podem dir que un objecte es pot definir amb un **estat** (o propietats) i un **comportament** o les accions que podem fer amb ell.

En programació, un objecte és similar a un objecte del mon real. Mostra un estat, definit per els atributs (variables) i un comportament, les accions que podem fer amb aquest objecte definides pels mètodes (funcions) que implementa.



Els mètodes actuen sobre l'estat de l'objecte i són el principal mecanisme de comunicació entre objectes.

Un objecte ha de definir tots els atributs necessaris per a definir el seu estat i ha de proporcionar mètodes per a modificar aquest estat, no hauria de permetre que d'altres objectes ho facin directament => Encapsulació.

Programar utilitzant objectes facilita la modularitat, l'ocultació d'informació, la reutilització de codi, ...

Classes

En el mon real hi ha molts objectes amb les mateixes característiques: Tenen el mateix comportament i el seu estat es pot definir amb els mateixos atributs. Per exemple, tots els cotxes d'un model només es diferencien amb els valors de certs atributs: matrícula, bastidor, color...

De fet, és possible que per a la nostra aplicació tots els cotxes es puguin definir amb els mateixos atributs i puguin executar els mateixos mètodes.

Aquesta idea general és el que anomenam classe. Una classe defineix la idea abstracta d'un cert tipus d'objecte. **Cada objecte és un cas particular, una instància, de la seva classe**.

Els atributs i els mètodes de la classe s'anomenen **membres**.

El més habitual és guardar cada classe en un fitxer amb el mateix nom (exactament igual) i l'extensió *java*.

Paquets

A una aplicació podem utilitzar un gran nombre de classes. Per organitzar-les agrupant les que estan relacionades utilitzam paquets.

Un paquet és una estructura lògica, no física. Una classe pertany a un paquet perquè s'especifica a la seva declaració, no perquè estigui dins una carpeta o una altra.

Així i tot la pràctica habitual, i el que fan per defecte la majoria d'entorns de desenvolupament, és crear una estructura de carpetes on es guarden els fitxers de les classes que segueix l'estructura dels paquets.

A més d'organitzar les classes, els paquets permeten definir l'accés que permetrà un objecte als seus membres. Podem definir que un membre sigui accessible només per objectes de la mateixa classe, de classes del mateix paquet o per qualsevol altre objecte.

Definició de classes

Ja hem dit abans que cada classe es sol definir dins un fitxer separat i que hauria de pertànyer a un paquet per afavorir la seva reutilització.

Declaració

A l'inici del fitxer hem d'indicar a quin paquet pertany la classe.

modificadorAccés determina qui podrà utilitzar la classe: public la podrà utilitzar qualsevol altra classe i si no ho especificam només classes del mateix paquet.

```
package nompaquet;

modificadorAccés class NomClasse{

        Declaració atributs;

        Declaració constructors;

        Declaració mètodes;

}
```

El **nom de la classe** segueix la mateixa convenció que les variables, excepte que l**a primera lletra ha de ser majúscula**.

Els membres de la classe es defineixen al cos, dins les claus.

En definir una classe estam creant un nou tipus de dades que es pot utilitzar per definir variables, paràmetres, tipus de retorn, ... Podríem dir que a qualsevol lloc on es podria utilitzar per exemple *String* podem utilitzar el nou tipus.

Declaració d'atributs

Un atribut és una dada que defineix l'estat de l'objecte. Es defineixen dins el cos de la classe, fora de qualsevol mètode o constructor. Es solen posar a l'inici. Una variable utilitzada per una tasca auxiliar no s'hauria de declarar com un atribut.

```
public class Cotxe{

  private String matricula;

  public int numPortes;
```

La forma de definir-los és *modificadorAccés tipus nom*;

El tipus pot ser qualsevol dels primitius o de les classes que hem creades. **S'inicialitzen als constructors, no a la declaració**.

Els modificadors d'accés i l'accés que permeten de més permissiu a menys:

Modificador d'accés	Classe	Paquet	Subclasse	Tothom
public	Si	Si	Si	Si
protected	Si	Si	Si	No
En blanc	Si	Si	No	No
private	Si	No	No	No

Per norma, **tots els atributs haurien de ser privats i crear mètodes públics que permetin accedir-hi**, així controlam millor el comportament de l'objecte.

Declaració de mètodes

Els mètodes defineixen el comportament dels objectes, quines accions es poden fer amb ells.

```
modificadorAccés void nom(paràmetres){

        Cos;

}
```

- El modificador d'accés pot ser qualsevol dels vists abans.
- El tipus que retorna el mètode pot ser qualsevol primitiu o qualsevol classe. Si el mètode no ha de tornar cap resultat posam **void**.
- El nom del mètode. Segueix la convenció de les variables, excepte que la primera paraula hauria de ser un verb.
- La llista de paràmetres, cada un amb el seu nom i tipus, entre parèntesis. Si no té paràmetres, posam els parèntesis buids.
- El cos del mètode amb el codi i les declaracions de variables locals necessaris per fer la seva tasca.

Sobrecàrrega de mètodes

Podem tenir més d'un mètode amb el mateix nom, sempre que es diferenciïn en el número, tipus i ordre dels seus paràmetres.

Constructors

Quan cream una instància d'una classe ho feim cridant uns mètode especials de la classe anomenats constructors. La JVM crea la instància assignant-li memòria, ... i després s'executa el codi que hem programat al constructor.

Aquest codi hauria de servir per inicialitzar l'estat de l'objecte, assignant valors als seus atributs.

Per defecte, si no cream cap constructor se'n crea un sense paràmetres que no fa res. Si en cream un ja no es genera aquest constructor per defecte. Podem crear els constructors que vulguem a una classe sempre que es diferenciïn a la signatura.

Es diferencien de la resta de mètodes perquè **tenen el mateix nom que la classe i no tenen tipus de retorn**, **ni tan sols void**. Normalment tenen accés public.

```
public class Casella {

    public Casella(int i, int j, int valor, boolean fixa) {

        ...

    }

    ...

}
```

Podem definir tants constructors com vulguem, sempre que es diferenciïn en la seva signatura.

Pas de paràmetres a mètodes i constructors

En definir un mètode o constructor decidim quins paràmetres tendrà. Per a cada paràmetre deim el nom i el tipus de dades, tant primitius com classes.

Els paràmetres sempre es passen per valor, això vol dir que les modificacions fetes dins el mètode a aquest valor no es veuen fora. **Les referències a objectes, com les dels arrays, no poden canviar, però si els seus atributs**.

Els paràmetres segueixen la mateixa nomenclatura que les variables.

Distinció entre paràmetres i arguments:

- Paràmetres: Les variables definides a la declaració del mètode, a i b.
- Arguments: Els valors passats al mètode, x i y.

```
public double multiplica(double a, double b) {
    return a*b;
}
```

this

this fa referència al propi objecte. Es pot utilitzar amb:

- Atributs: La raó més habitual per utilitzar-lo amb atributs és que un paràmetre del mètode o constructor oculta l'atribut:

```
public void setI(int i) {
    this.i = i;
}
```

- Constructors: És pot utilitzar per a cridar un constructor des de dins un altre. En aquest cas, la crida a l'altre constructor ha de ser la primera instrucció.

```
public Rectangle(int width, int height) {
    this(0, 0, width, height);
}
public Rectangle(int x, int y, int width, int height) {
    this.x = x;
    this.y = y;
    this.width = width;
    this.height = height;
}
```

Utilització d'objectes

Per a poder utilitzar un objecte s'han de seguir dues passes: declaració i instanciació

Declaració

Abans d'utilitzar un objecte d'una determinada classe hem de dir al compilador que ho volem fer:

```
Cotxe fordFesta;
```

Utilitzarem una variable que es diu *fordFesta* que serà un objecte de la classe Cotxe.

Instanciació

Hem de crear l'objecte per a poder-lo utilitzar:

```
fordFesta=new Cotxe(); //Si la classe Cotxe té un constructor sense paràmetres
```

O

```
fordFesta=new Cotxe("2345-ZHX",1400,3); //Si en té un que necessita la matrícula,

// el cubicatge del motor i el nombre de portes.
```

Utilització d'objectes

Una vegada tenim una referència a un objecte el podem utilitzar. Tenim accés a tots els atributs i mètodes que determinin els seus selectors d'accés.

- Atributs:

```
objecte.atribut=valor;          fordFesta.portes=4;
variable=objecte.atribut;              int cc=forFesta.cubicatge;
```

- Mètodes:

```
objecte.metode();               forFesta.omplirDeposit();
variable=objecte.metode();  matricula=fordFesta.getMatricula();
```

- objecte2=objecte

  Assigna una altra referència al mateix objecte, no en fa una còpia. Si canviam el valor d'alguns dels atributs de l'objecte, canviarà per a les dues referències.

El valor null

Significa nul, és a dir que la variable que conté el valor null no està associada a cap objecte:

```
Cotxe c=null;
```

Com que no està associada a cap objecte, l'accés a qualsevol dels seus membres donarà error en temps d'execució, en concret botarà una excepció del tipus NullPointerException. És un error de programació. No s'hauria de donar.

Seria convenient que sempre que dubteu si una variable té un objecte associat o no, ho comprovésseu:

```
if (c!=null) ...
```

O si heu de fer qualque comprovació podem jugar amb l'ordre dels operands i amb el fet que els operadors lògics estan curtcircuitats. Si c==null:

```
if(c.getMarca().equals("")) Donar error.        If ("".equals(c.getMarca())) no en dona.

If(!c.getMarca().equals("") && c!=null) dona error. En canvi

If(c!=null && !c.getMarca().equals("")) no.
```

Pas d'objectes com a paràmetres

En definir una classe no feim altre cosa que definir un tipus de dades que es pot utilitzar com a qualsevol tipus primitiu definit en el llenguatge.

Per tant, podem passar objectes de qualsevol classe com a paràmetres d'un mètode, i també tornar-los com a resultat del mètode.

Com qualsevol altre tipus, els objectes es passen al mètode per valor, no per referència.

Hem vist que en crear un objecte el que emmagatzemam a la variable realment és una referència a l'objecte. Seria el mateix cas que amb els arrays.

Per tant, dins un mètode podrem canviar atributs d'un objecte passat com a paràmetre i els canvis es mantindran en sortir del mètode.

Si el que feim és assignar al paràmetre un objecte nou, en acabar el mètode la variable que hem passat com a argument seguirà apuntant a l'objecte original.

Per exemple

```
public void assignarDades(Modul p){        //Rep un Modul com a paràmetre
                                           //i li canvia el nom. Correcte
    p.nom="SWPRO";
}
```

```
public void incorrecte(Modul p){           //Rep un mòdul com a paràmetre,
i n'hi assigna un altre dins el mètode
```

```
        p=new Modul();                  //En tornar a inici m mantendrà el seu
                                        //valor original. Per tant no servirà de

        p.nom="SWDWS";                  //res el que facem aquí
}
```

```
public Modul correcte(String n){        //Rep una cadena com a paràmetre.
                                        //Crea un Modul i el torna com a resultat
        Modul p=new Modul();            //del mètode. Correcte.
        p.nom=n;
        return p;
}
```

```
public void inici(){
  Modul m=new Modul();
  assignarDades(m);
  System.out.println(m.nom); //SWPRO;
  incorrecte(m);
  System.out.println(m.nom); //SWPRO;
  m=correcte("AIDWC");
```

```
  System.out.println(m.nom); //SWPRO;

}
```

Comparació d'objectes

L'operador de comparació == s'utilitza per comprovar si dues variables tenen el mateix valor.

```
Cotxe a = new Cotxe("1234ASD");
Cotxe b = new Cotxe("1234ASD");
System.out.println(a==b);
```

Encara que els dos objectes semblin iguals, són de la mateixa classe i tenen les mateixes dades, per pantalla apareixerà **false**.

Hem dit que l'operador == mira si el contingut de les variables és el mateix. En el cas d'un objecte, el contingut de la variable és la referència de l'objecte. Com que són dos objectes diferents, tenen referències diferents, per tant la comparació torna false.

```
Cotxe a = new Cotxe("1234ASD");
Cotxe b = a;
System.out.println(a==b);
```

En aquest cas mostrarà per pantalla **true**.

Si a la nostra aplicació ens serveix aquest concepte d'igualtat, si ens va bé que la comparació torni true només si són exactament el mateix objecte, perfecte.

Si a la nostra aplicació consideram que dos objectes són iguals depenent dels valors dels seus atributs, per exemple, dos cotxes són iguals si tenen la mateixa matrícula, llavors l'operador comparació no ens serveix.

En el seu lloc hem d'utilitzar el mètode *equals*. Aquest mètode l'hereten totes les classes, totes les classes el tenen. Aquest mètode té un paràmetre i compara l'objecte que executa el mètode amb l'objecte que rep al paràmetre.

En la seva versió per defecte el mètode equals torna el resultat de l'operador ==.

La gràcia és que a la nostra classe podem modificar aquest mètode de manera que compari els atributs que consideram necessaris per decidir si l'objecte que executa el mètode és igual al que rep per paràmetre.

```
public class Cotxe{
	private String matricula;
	...
	public boolean equals(Cotxe altre){
		return this.matricula.equalsIgnoreCase(altre.matricula);
	}
}
```

En aquest cas,

```
Cotxe a = new Cotxe("1234ASD");
Cotxe b = new Cotxe("1234ASD");
System.out.println(a.equals(b));
```

mostrarà per pantalla **true**.

Destructors. Garbage collector.

Per utilitzar un objecte el cream amb el constructor. Però que feim quan volguem deixar d'utilitzar-lo? Altres llenguatges permeten definir un *destructor* que s'executa quan *destruïm* l'objecte, però Java no té res semblant.

El mecanisme que té Java és una mica diferent. El que fa és mantenir una llista amb totes les variables o atributs d'altres objectes que fan referència a cada objecte.

Cada vegada que assignam un objecte a una variable modifica la llista.

Cada vegada que una variable deixa de fer referència a l'objecte (per que li assignam un altre valor, perquè la posam a null, perquè la variable deixa d'existir en acabar el mètode on estava definida, ...) s'actualitza la llista de referències a l'objecte.

La màquina virtual té un procés que es diu ***garbage collector*** que s'executa independentment. El que fa aquest procés és comprovar la llista de referències de cada objecte. Si està buida, si res apunta a l'objecte, vol dir que no es podrà utilitzar més. Per tant allibera la memòria que ocupa.

En alguns casos està justificat escriure un mètode que cridaríem abans d'anular les seves referències. Es tracta dels objectes que utilitzen recursos del sistema com pot ser un fitxer. Per evitar problemes és millor que abans de deixar d'utilitzar l'objecte alliberem aquests recursos.

Membres de classe

Fins ara tots els atributs i mètodes que hem creat formen part d'un objecte, però pot ser que en necessitem que pertanyin a la classe, no a cap dels objectes.

És el que s'anomenen membres de classe. És defineixen amb la paraula clau *static*. Es poden inicialitza a la declaració.

```
private static int numModuls = 0;
```

La manera correcte d'utilitzar-los és directament amb el nom de la classe. No s'haurien d'utilitzar des de cap objecte encara que sigui possible.

```
Cicle.numModuls=10;
```

Podem definir

- Atributs de classe: Son dades que fan referència a la classe i no a cap objecte en concret.

```
private static tipus nom;

NomClasse.nomAtribut;
```

- Constants:

```
private static final tipus NOM;
```

- Mètodes de classe: No poden utilitzar membres d'instància (no pertanyen a cap instància), només de classe. Es solen crear per a accedir a variables de classe o en classes d'utilitat, pe Math

```
accés static tipus nom(paràmetres){...}

NomClasse.nom(arguments);  //Es crida posant el nom de la classe, no un objecte.
```

Per exemple:

```
Integer.parseInt(“23”);
```

Si feim un import estàtic *import static nomClasse.*;* podem utilitzar els mètodes estàtics de la classe directament, sense posar abans el nom de la classe.

```
import static Integer.*;

int x = parseInt("23");
```

Classes d'utilitat

S'anomenen així a les classes que no tenen atributs (o només constants) i tots els seus membres són de classe.

S'utilitzen per a centralitzar mètodes que s'utilitzen a diversos llocs de l'aplicació però no pertanyen a cap objecte concret.

Per exemple les classes *UtilitatsConsola* o *OrdenacióICerca* que hem fet durant el curs són classes d'utilitat. La primera agrupa mètodes útils per a treballar amb la consola del sistema (llegirSencer, ..mostrarArray, ..) i la segona agrupa mètodes per ordenar i cercar dins arrays (quicksort, ...).

public static void main(String[] args)

En definir una classe declaram els atributs i els mètodes que tendrà. Hem vist que podem decidir quins es podran utilitzar des d'objectes d'altres classes, però on es comença a executar el codi de l'aplicació?

Per executar una aplicació java hem d'introduir l'ordre

```
java nomClasse.
```

La JVM cerca a aquesta classe el mètode ***main*** i executa el seu codi. Aquest mètode té una sèrie de característiques:

- És de classe, *static*. Té sentit per que no podem crear un objecte abans d'executar l'aplicació per executar un mètode d'instància.
- Com que és *static* no podem utilitzar directament ni atributs ni mètodes de la pròpia classe que no siguin també static.
- Podem declarar-hi variables locals i accedir als membres d'aquestes variables locals que ens permetin els seus modificadors d'accés public, ...
- Ha de tenir un únic paràmetre, un array de cadenes de text. El paràmetre recull els arguments que li passam a la classe des de la línia de comandes.

La forma habitual d'utilitzar el main és per crear un objecte de la classe on està definit i executar un o diversos mètodes que posin en marxa la lògica del programa.

Algunes classes de l'API

L'API de Java és una part fonamental del llenguatge. És una col·lecció de classes i altres tipus de dades que implementen aspectes comuns a la majoria d'aplicacions, de manera que no faci falta programar-ho tot des de zero.

Wrappers

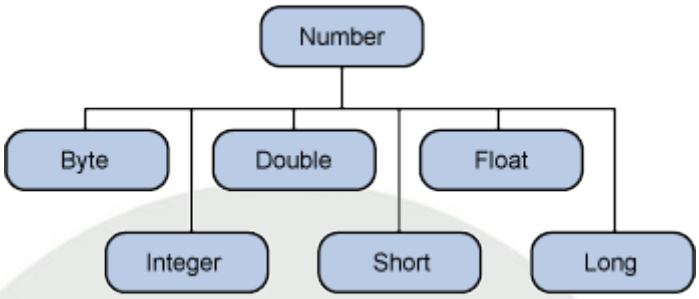
Per a cada tipus primitiu hi ha una classe que el representa. Pot ser útil quan:

- Hem d'utilitzar un mètode que espera un objecte com a paràmetre.
- Necessitam utilitzar les constants definides per les classes descendents de Number.
- Necessitam utilitzar qualque mètode de conversió de tipus que implementen aquestes classes, per exemple

```
int  x= Integer.parseInt("23);
```

Aquestes classes s'anomenen *wrappers* del respectiu tipus primitiu. La màquina virtual fa la conversió entre classe i tipus primitiu i a la inversa (*boxing* i *unboxing*) .

Per a tipus numèrics tenim:



Per exemple,

```
Integer x, y;

x = 12;

y = 15;

System.out.println(x+y);
```

A part de les classes de la imatge n'hi ha quatre més: *BigInteger* i *BigDecimal* per a càlculs de precissió, i *AtomicInteger* i *AtomicLong* per a aplicacions multithread.

Character

Wrapper per al tipus char.

Proporciona mètodes per:

- Comprovar si el caràcter és un dígit, una lletra, ...
- Comprovar si és una majúscula o minúscula.
- Transformar-lo en majúscula o minúscula.

Boolean

Wrapper per el tipus primitiu *boolean*.

Proporciona mètodes per:

- Convertir cadenes a booleà
- Comparar booleans
- Aplicar operacions lògiques, ...

java.util.Arrays

És una classe d'utilitat, amb tots els mètodes i constants de classe. Implementa mètodes com:

- .binarySearch(array, valor): Cerca el valor dins l'array. Torna la primera posició de l'element dins l'array.
- .compare(array1, array2): Compara els dos arrays. Torna 0 si són iguals, un negatiu si el primer és menor que els segon i un positiu si el primer és major que el segon.
- .copyOf(array, longitud): Fa una còpia de l'array. Si la longitud és menor que la de l'array desprecia els elements que sobren. Si és major omple les posicions que falten amb nulls.
- .fill(array, valor): Omple l'array amb el valor donat.
- .sort(array): Torna l'array ordenat segons l'ordre natural del tipus dels elements. Per exemple de menor a major per valors numèrics, l'ordre alfabètic per arrays de cadenes, ...
- .toString(array): Torna una cadena amb les dades de l'array en un format adequat per imprimir, per exemple [1, 2, 3]

java.lang.Math

És una classe d'utilitat, amb tots els mètodes i constants de classe. Incorpora les constants *E* i *PI* i implementa mètodes com:

- Operacions de valor absolut, arrodoniment, màxims i mínims.
- Operacions exponencials i logarítmiques.
- Operacions trigonomètriques.
- Números aleatoris.

java.lang.String

Encara que l'hem utilitzada com un tipus primitiu, es tracta d'una classe. Cada vegada que li assignam un valor cream un objecte nou.

Alguns mètodes:

- .equals(unAltreString): És la forma correcta de comprovar que el contingut de la cadena que executa el mètode té el mateix contingut que la que rep com a paràmetre.
- .equalsIgnoreCase(unAltreString): comprova si el contingut és el mateix ignorant majúscules i minúscules.
- .toUpperCase() / .toLowerCase(): per passar a majúscules/minúscules la cadena que l'executa.
- charAt(int): torna el caràcter que ocupa la posició que rep al paràmetre.
- statrtsWith / endsWith: Si una cadena começa o acaba per la cadena o expressió regular que rep com a paràmetres
- trim: Torna una cadena amb el contingut de la que executa el mètode eliminant els espais en blanc inicials i finals.
- valueOf(): Mètode de classe sobrecarregat. Rep un sencer, double, ... i torna una cadena representant el valor del paràmetre que ha rebut
- ...

java.util.ArrayList

Contenidor que es pot utilitzar tant com un array com com una llista. No hem de definir una mida inicial, en afegir elements, si fa falta, creix per que hi càpiguen.

No pot contenir valors de tipus primitius, només objectes.

Ofereix operacions com add (afegir un element), get (recuperar -lo), toArray(torna un array amb els elements que conté), ...

- En crear-la hem d'indicar de quina classe són els elements que conté:

```
ArrayList<Doi> llistaDois=new ArrayList<>();
```

- Podem afegir un element al final:

```
Doi primer=new Doi(1,"primer");
llistaDois.add(primer);
```

- O a una posició determinada. Desplaça l'element que ocupa aquesta posició i els següents cap al final de la llista:

```
Doi y=new Doi(4,"quart");
llistaDois.add(1, y);
```

- Podem substituir l'element que ocupa una posició determinada:

```
Doi z=new Doi(5,"cinquè");
LlistaDois.set(1, z);
```

- Podem recuperar l'element que ocupa una posició determinada:

```
Doi d1=llistaDois.get(1);
```

- Recórrer la llista:

```
for(int i = 0; i < llistaDois.size(); i++){
  System.out.println(llistaDois.get(i));
}
```

o, si no necessitam per rés la posició que ocupa cada element, millor:

```
for(Doi d: llistaDois){
        System.out.println(d);
}
```

- Eliminar l'element que ocupa una posició determinada:

```
llistaDois.remove(1);
```

- Eliminar la primera aparició d'un determinat element:

```
llistaDois.remove(primer);
```

java.time

Paquet que inclou el necessari per a treballar amb dates i hores. Inclòs al Java 8.

java.time.LocalDate

Crea un objecte que representa la data especificada als arguments. Té una sèrie de getters que ens permeten recuperar tota la informació de la data.

Per crear una data concreta:

```
LocalDate date=LocalDate.of(2016, Month.NOVEMBER, 24);
```

Per crear un objecte amb la data actual del sistema:

```
LocalDate date=LocalDate.now();
```

java.time.LocalTime

Torna un objecte que conté l'hora actual, des de l'hora fins a nanosegons.

```
LocalTime time=LocalTime.now();
```

Podem construir-ne un passant-li les dades:

```
LocalTime time=LocalTime.of(hores, minuts, segons);
```

Té una sèrie de mètodes que ens permeten establir l'hora amb distinta precisió.

java.time.LocalDateTime

Inclou informació sobre l'hora i la data

java.time.Instant

Crea un objecte que representa l'instant determinat fins el detall de nanosegons.

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

Té mètodes que permeten incrementar-lo, ...

```
Instant oneHourLater = Instant.now().plusHours(1);
```

java.time.Duration

Pensat per calcular la duració d'un període de temps breu, que no inclogui dates. Té dos atributs, els segons del període i els nanosegons que "sobren" del darrer segon. Es recuperen amb els getters. També té mètodes toXXX que permeten transformar l'interval complet a segons, millisegons, ... Si es volen hores i minuts s'han de calcular a partir dels segons.

Un mètode molt útil és *between* que ens permet calcular la duració entre dos instants:

```
Duration d = Duration.between(t1, t2);
```

java.time.Period

Pensat per calcular la duració d'un període de temps breu, que no necessiti detallar més que dies. Té getters per els anys, mesos i dies del període.

```
LocalDate today = LocalDate.now();

LocalDate birthday = LocalDate.of(1960, Month.JANUARY, 1);

Period p = Period.between(birthday, today);

System.out.println("You are " + p.getYears() + " years, " + p.getMonths() +  "
months, and " + p.getDays() +      " days old.");
```

java.time.temporal.ChronoUnit

Defineix unitats de temps, per exemple dies, mesos, hores, segons, ... i permet una sèrie d'operacions amb elles, per exemple between:

```
long dies = ChronoUnit.DAYS.between(naixament, ara);
```

Calcula les unitats, en aquest cas dies, entre els dos arguments.

Si els paràmetres són del tipus *LocalDate* només podreu arribar fins a dies.

Si els paràmetres són de tipus *LocalDateTime* podreu arribar fins a nanosegons

```
long dies = ChronoUnit.NANOS.between(naixament, ara);
```