DAM - M03Block2 - Exercici_de_repas - teoria_001

Contingut:

1. Introducció

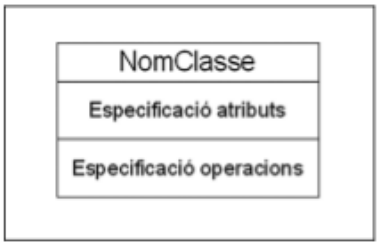
Tot és un objecte, amb una identitat pròpia.
Un programa és un conjunt d'objectes que interactuen entre ells.
Un objecte pot estar format per altres objectes.
Cada objecte pertany a un tipus concret: una classe.
Objectes del mateix tipus tenen un comportament idèntic.

Una classe és l'especificació formal de les propietats (els atributs) i el comportament esperat (la llista d'operacions, els mètodes) d'un conjunt d'objectes del mateix tipus, i que actua com una plantilla per generar cadascun d'ells.

Un cop instanciat un objecte, aquest sempre pertany a la mateixa classe. No es pot canviar el seu tipus dinàmicament.

2. Especificació de les classes

Com sempre, abans de saltar sobre el teclat cal fer una tasca prèvia de reflexió. Per aquest motiu, un cop es té una idea més o menys clara de quins objectes formaran part del vostre programa i com s'estructuren, és el moment de moure el focus a l'especificació formal de les classes de manera completa, amb tots els seus atributs i operacions. Per dur a terme aquesta especificació formal, normalment s'usa el llenguatge **UML** que és un llenguatge estàndard que permet especificar amb notació gràfica el contingut de les classes (atributs i mètodes) i com també com es relacionen entre elles.



(operacions = els mètodes)

2.1. Especificació d'atributs

Els atributs ens permeten especificar les propietats o l'estat dels objectes d'una classe.
En definir una classe, els atributs s'especifiquen segons la sintaxi següent:

visibilitat tipus nomAtribut {valor inicial};

El camp de **valor inicial** es correspon al valor que pren l'atribut en el moment d'instanciar un objecte d'aquesta classe. Concretar-lo en l'especificació dels atributs és opcional.

Visibilitat dels atributs:

Una característica específica de l'orientació a objectes és que per a cada atribut cal definir el que s'anomena la seva visibilitat. La visibilitat d'un atribut indica si aquest és accessible directament des d'altres classes.

Atribut públic: dóna accés a tothom. Es pot accedir des de qualsevol fitxer .java .

Atribut privat : prohibeix l'accés a tothom menys pels mètodes de la pròpia classe. Només des de el fitxer de la pròpia classe (el .java) tindrem accés.

Atribut protected: es comporta com a public per a les classes derivades de la classe i com a private per a la resta de classes.

Sense modificador: es comporta com a public per a les classes del mateix paquet i com a private per a la resta de classes.

Access Levels

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

2.2. Especificació dels mètodes

Dins la definició d'una classe, les operacions disponibles s'especifiquen de la manera següent:

visibilitat tipusRetorn nomMètode (llistaParàmetres) {}

El camp **llistaParàmetres** té el format següent:

tipus nomParàmetre1, ... , tipus nomParàmetreN

Totes les explicacions donades per als tipus o la visibilitat en el cas dels atributs també són aplicables al cas dels mètodes.

Donat un mètode cridat sobre un objecte **a:ClasseA**, es pot considerar que el mètode que s'executarà té accés directe a les dades següents:

- Els valors emmagatzemats en els atributs de l'objecte **a:ClasseA** .
- Els valors dels paràmetres del mètode.

En cas que qualsevol d'aquests valors sigui un objecte **b:ClasseB**, l'operació també pot:

- Accedir als valors dels atributs amb visibilitat pública de **b:ClasseB** .
- Cridar operacions amb visibilitat pública sobre l'objecte **b:ClasseB** .

Per mantenir la **cohesió** en un disseny orientat a objectes, cada classe ha de representar un únic element, perfectament definit, dins la descomposició del problema.
Exemple: un programa de disseny de fabricació de cotxes haurà de tenir una classe definida per a cada tipus de vehicle (cotxe, SUV, tot terreny, motocicleta, moto,...) però en canvi un programa de lloguer de cotxes en tindrà prou amb 1 classe vehicle la qual tingui un atribut que digui quin tipus de vehicle és (cotxe, SUV, tot terreny, motocicleta, moto,...).

Cal ubicar els **mètodes** que operen amb una informació determinada en la mateixa classe en què es troba aquesta informació.

2.3. Com es creen els objectes?

La creació d'un objecte la realitza sempre una operació especial de la classe, anomenada **constructor**, que es distingeix perquè té el mateix nom que la classe (incloent majúscules/minúscules).
Els constructors poden incorporar paràmetres i això permet que hi pugui haver diferents constructors, que es distingeixen pel nombre i/o els tipus dels seus paràmetres.
Tota classe té, com a mínim, 1 constructor.

Creem l'objecte de la següent manera:

tipus nomInstància = new nomDeLaClasse(nomParàmetre1, ... , nomParàmetreN);

on **tipus** hauria de ser el nom de la classe (= a nomDeLaClasse). Els paràmetres només fan falta si el constructor en requereix i serviran per a inicialitzar l'objecte amb un valors determinats.
Més exemples:

```
// Declaració de variable de referència no inicialitzada
X obj1;

// Creació d'objecte al que es podrà accedir via la variable de
// referència obj1
obj1 = new X(...);

// Declaració de variable de referència i creació d'objecte al
// que es podrà accedir via la variable de referència obj2
X obj2 = new X(...);

// Declaració de variable de referència no inicialitzada
X obj3;

// La variable obj3 fa referència al mateix objecte que fa
// referència la variable obj1
obj3 = obj1;

// Declaració de variable de referència que fa referència al
// mateix objecte que fa referència la variable obj2
X obj4 = obj2;
```

```
Date d1 = new Date (109,0,1); //Objecte inicialitzat amb data 1-1-2009 a les
     00:00:00
Date d2 = new Date (0);   //Objecte inicialitzat amb data 1-1-1970 a les
     00:00:00
Date d3 = new Date ();   //Objecte inicialitzat amb la data i l'hora del
     sistema
```

Si no es fa cap constructor, Java crea 1 automàticament i serà sense paràmetres.

2.4. Com es manipulen els objectes?

La manipulació dels objectes d'una classe s'ha de fer per mitjà dels **mètodes**, els **getters** i els **setters** que proporciona la pròpia classe.
Si volem accedir al valor d'un atribut de la classe (per a consultar-lo o modificar-lo), s'hauria de fer servir els **getters** (per agafar el valor) i els **setters** (per afegir o modificar el valor) i no accedir-hi directament.

Els noms dels getters i setters són: get/set + nom del atribut començant amb majúscula.
Exemple: l'atribut **edat** tindrà el getter **getEdat()** i el setter **setEdat()**.

2.5. Com es destrueixen els objectes?

En Java tots els objectes són dinàmics. Per tant, caldria portar un control exhaustiu de tots els objectes creats i anar-los destruint explícitament quan ja no fossin necessaris. Afortunadament, Java ens estalvia aquesta feina, de manera que ens permet crear tants objectes com es vulgui (únicament limitats per la pròpia capacitat de memòria del sistema), els quals mai han de ser destruïts, ja que és l'entorn d'execució de Java el que elimina els objectes quan determina que no s'utilitzaran més.
El **garbage collector** (recuperador de memòria) és un procés automàtic de la màquina virtual Java que periòdicament s'encarrega de recollir els objectes que ja no es necessiten i els destrueix tot alliberant la memòria que ocupaven.

2.6. Declaració d'un classe

Generalment el codi d'una classe està en el següent ordre:

Atributs (línies 3 - 4)
Getters i setters (línies 8 - 31)
Constructors (no hi ha, això implica que Java crearà 1 automàticament sense paràmetres)
Mètodes (línies 33 - 36)

Exemple de la classe **Persona.java**:

```
//Fitxer Persona.java
public class Persona {
   String dni;
   String nom;
   short edat;
   // Retorna: 0 si s'ha pogut canviar el dni
   //          1 si el nou dni no és correcte – No s'efectua el canvi
   int setDni(String nouDni) {
      // Aquí hi podria haver una rutina de verificació del dni
      // i actuar en conseqüència. Com que no la incorporem, retornem sempre 0
      dni = nouDni;
      return 0;
   }

   void setNom(String nouNom) {
      nom = nouNom;
   }

   // Retorna: 0 si s'ha pogut canviar l'edat
   //          1 : Error per passar una edat negativa
   //          2 : Error per passar una edat "enorme"
   int setEdat(int novaEdat) {
      if (novaEdat<0) return 1;
      if (novaEdat>Short.MAX_VALUE) return 2;
      edat = (short)novaEdat;
      return 0;
   }

   String getDni() { return dni; }
   String getNom() { return nom; }
   short getEdat() { return edat; }

   void visualitzar() {
      System.out.println("Dni..........:" + dni);
      System.out.println("Nom..........:" + nom);
      System.out.println("Edat.........:" + edat);
   }
}
```

2.7. Modificadors dins d'una classe

Recordar:

Access Levels

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Ara farem una altra classe (**CridaPersona.java**) que accedeixi a la classe **Persona.java** (veure l'exemple de l'apartat anterior):

```
//Fitxer CridaPersona.java
public class CridaPersona {
   public static void main(String args[]) {
      Persona p = new Persona();
      p.dni = "--$%#@--";
      p.nom = "";
      p.edat = -23;
      System.out.println("Visualització de la persona p:");
      p.visualitzar();
   }
}
```

En aquest cas estem en un programa extern a la classe **Persona** i es veu com accedim directament a les dades **dni**, **nom** i **edat** de la persona creada, i podem fer autèntiques animalades. El compilador no es queixa (cal haver compilat també l'arxiu Persona.java en el mateix directori per a simular que estan en el mateix paquet) i l'execució dóna el resultat:

```
Visualització de la persona p:
Dni..........:--$%#@--
Nom..........:
Edat.........:-23
```

Acabem de veure, doncs, que la versió actual de la classe **Persona** permet el lliure accés als valors dels seus atributs, ja que en la definició d'aquestes dades no s'ha posat al davant el modificador adequat per evitar-ho (i per tant tenen el modificador **no modifier**). Les classes **CridaPersona** i **Persona** , en estar situades en el mateix directori (a llavors s'han considerat del mateix paquet) i, per tant, en no haver-hi cap modificador d'accés en la definició de les dades **dni**, **nom** i **edat**, la classe **CridaPersona** hi ha tingut accés total. A més, en no haver-hi cap modificador d'accés en la definició dels mètodes, aquests no poden ser cridats per classes de paquets diferents del paquet al qual pertany la classe **Persona** .

Sembla lògic, doncs, fer evolucionar la versió actual de la classe **Persona** cap a una classe que tingui les dades declarades com a privades i els getters, setters i mètodes com a públics.
La idea és que per accedir als atributs de la classe A des de un objecte de la classe B, sempre es faci a través dels getters i setters de la classe A. D'aquesta manera es poden manipular les dades quan es consulten o es poden comprovar que siguin correctes abans d'insertar-les o modificar-les.

El mètode **main** sempre serà declarat public.

2.8 Sobrecàrrega de mètodes

De vegades, en els programes, cal dissenyar diverses versions de mètodes que tenen un mateix significat i/o objectiu però que s'apliquen en diferents tipus i/o nombre de dades. Així, si necessitàvem disposar d'una funció que sabés sumar dos enters i d'una funció que sabés sumar dos

reals, podríem fer simplement dos mètodes diferents anomenats sumaEnters i sumaReals . Els dos tenen el mateix objectiu i significat, tot i que la gestió interna pot ser força diferent, i des d'un punt de vista lògic, com que les dues permeten calcular una suma, Java permet declarar mètodes repetits amb el mateix nom.

La **sobrecàrrega** de mètodes és la funcionalitat que permet tenir mètodes amb codis diferents amb un mateix nom.

Exemple:

```
public int suma (int n1, int n2) { ... }
public double suma (double r1, double r2) { ... }
```

Hi ha dues regles per poder aplicar la sobrecàrrega de mètodes:

- La llista d'arguments ha de ser suficientment diferent per permetre una determinació inequívoca del mètode que es crida.
- Els tipus de dades que retornen poden ser diferents o iguals i no n'hi ha prou de tenir els tipus de retorn diferents per distingir el mètode que es crida.

El compilador només pot distingir el mètode que es crida a partir del nombre i tipus dels paràmetres indicats en la crida.

2.9 Declaració de constructors

De vegades es necessita executar un mètode constructor en concret per implementar la inicialització, ja que pot ser necessari fer el següent:

- Recollir valors (pas de paràmetres en el moment de construcció) de manera que es puguin tenir en compte en la construcció de l'objecte.
- Gestionar errors que puguin aparèixer en la fase d'inicialització.
- Aplicar processos, més o menys complicats, en els quals poden intervenir tot tipus de sentències (condicionals i repetitives).

Tot això és possible gràcies a l'existència dels mètodes constructors, un dels quals sempre es crida en crear un objecte amb l'operador **new** . Per tant, a les classes creades per vosaltres pot ser necessari declarar constructors. En el disseny d'una classe es poden dissenyar mètodes constructors, però si no se'n dissenya cap, el llenguatge proveeix automàticament d'un constructor sense paràmetres.

Els mètodes constructors d'una classe han de seguir les normes següents:

- El nom del mètode és idèntic al nom de la classe.
- No se'ls pot definir cap tipus de retorn (ni tant sols void , no es posa absolutament res. Es deixa en blanc).
- Poden estar sobrecarregats, és a dir, podem definir diversos constructors amb el mateix nom i diferents arguments. En cridar l'operador **new**, la llista de paràmetres determina quin constructor s'utilitza.

• Si es defineix algun constructor (amb paràmetres o no), el llenguatge Java deixa de proporcionar el constructor sense paràmetres automàtic i, per tant, per poder crear objectes cridant un constructor sense paràmetres, caldrà definir-lo explícitament.

Exemples:

```
public Persona () {}

public Persona (String sDni, String sNom, int nEdat) {
  dni = sDni;
  nom = sNom;
  if (nEdat>=0 && nEdat<=Short.MAX_VALUE)
   edat = (short)nEdat;
}
```

2.10 La paraula reservada "this"

En Java existeix una paraula reservada especialment útil per tractar la manipulació d'atributs i la seva inicialització. Es tracta de **this**, que té dues finalitats principals:

• Dins els mètodes no constructors, per fer referència a l'objecte actual sobre el qual s'està executant el mètode. Així, quan dins un mètode d'una classe es vol accedir a una dada de l'objecte actual, podem utilitzar la paraula reservada **this**, escrivint **this.nomDada**, i si es vol cridar un altre mètode sobre l'objecte actual, podem escriure **this.nomMètode(...)**. En aquests casos, la utilització de la paraula **this** és redundant, ja que dins un mètode, per referir-nos a una dada de l'objecte actual, podem escriure directament **nomDada**, i per cridar un altre mètode sobre l'objecte actual podem escriure directament **nomMètode(...)**. De vegades, però, la paraula reservada **this** no és redundant, com en el cas en què es vol cridar un mètode en una classe i cal passar l'objecte actual com a argument: **nomMètode(this)**.

• Dins els mètodes constructors, com a nom de mètode per cridar un altre constructor de la pròpia classe. De vegades pot passar que un mètode constructor hagi d'executar el mateix codi que un altre mètode constructor ja dissenyat. En aquesta situació seria interessant poder cridar el constructor existent, amb els paràmetres adequats, sense haver de copiar el codi del constructor ja dissenyat, i això ens ho facilita la paraula reservada **this** utilitzada com a nom de mètode: **this(<llistaParàmetres>)** . La paraula reservada **this** com a mètode per cridar un constructor en el disseny d'un altre constructor només es pot utilitzar en la primera sentència del nou constructor. En finalitzar la crida d'un altre constructor mitjançant **this**, es continua amb l'execució de les instruccions que hi hagi després de la crida **this(<llistaParàmetres>)**.

Exemple dins de mètode constructor:

Volem crear un altre constructor que construeixi una persona a partir d'una persona ja existent, és a

dir, el constructor **Persona (Persona p)**.
Però, d'altra banda, ja tenim un constructor (anomenem-lo xxx) que ens sap construir una persona a partir d'un dni, un nom i una edat passats per paràmetre. Per tant, per construir una persona a partir d'una **persona p** donada, ens interessa cridar el constructor xxx passant-li com a paràmetres el dni, el nom i l'edat de la **persona p**. Això ens ho facilita la paraula reservada **this** com a crida d'un constructor existent:

```
public Persona (Persona p) {
  this (p.dni, p.nom, p.edat);
}
```

Exemple dins de mètode no constructor:
En segon lloc, suposem que volem tenir un mètode, anomenat **clonar()**, que aplicat sobre un objecte **Persona** en creï un clon, és a dir, una altra persona idèntica, i retorni la referència a la nova persona. Per aconseguir-ho hem de dissenyar el mètode que en seu interior cridi un dels constructors de la classe. Si optem per utilitzar el constructor **Persona (Persona p)** necessitem la paraula reservada **this** per fer referència a l'objecte actual (que és el que volem clonar):

```
public Persona clonar () {
  return new Persona (this);
}
```

2.11 Elements estàtics d'una classe (static + final)

Alguns elements d'una classe es poden declarar com **estàtics**. Per fer-ho, el llenguatge Java proporciona la paraula reservada **static**, amb tres finalitats:

- Com a modificador en la declaració de dades membres d'una classe, per aconseguir que la dada afectada sigui comuna a tots els objectes creats d'una mateixa classe.
  Sintaxi:

  static [<altresModificadors>] <tipusDada> <nomDada> [=<valorInicial>];

  Atès que una dada static és comuna per a tots els objectes de la classe, s'hi accedeix de manera diferent de la utilitzada per les dades no static :

  • Per accedir-hi des de fora de la classe (possible segons el modificador d'accés que l'acompanyi), no es necessita cap objecte de la classe i s'utilitza la sintaxi **NomClasse.nomDada**. Recordeu que perquè això funcioni, igualment, la dada s'ha de declarar com pública.
  • Per accedir-hi des de la pròpia classe, no cal indicar cap nom d'objecte (**nomObjecte.nomDada**), sinó directament el seu nom.

Si volem que aquesta variable sigui una constant li haurem de posar **final** (static final int pi).

- Com a modificador en la declaració de mètodes d'una classe, per aconseguir que el mètode afectat es pugui executar sense necessitat de ser cridat sobre cap objecte concret de la classe.
  Sintaxi:

  static <valorRetorn> <nomMètode> (<llistaArguments>)

  Un altre cas potser més habitual i evident és el mètode **main()** que s'usa en les classes principals. Per poder invocar un mètode cal fer-ho sobre un objecte. Però com és possible cridar main , si en iniciar l'execució del programa encara no existeix cap objecte? Els objectes es creen precisament dins el **main()** !. La resposta està a fer-lo static , de manera que és possible fer-ne la crida sense la necessitat que hi hagi cap objecte existent prèviament.
  Dels mètodes static cal saber:
  - Es criden utilitzant la sintaxi **NomClasse.nomMètode()**. El llenguatge Java permet cridar-los pel nom d'un objecte de la classe, però no és lògic.
  - En el seu codi no es pot utilitzar la paraula reservada **this**, ja que l'execució no s'efectua sobre cap objecte en concret de la classe.
  - En el seu codi només es pot accedir als seus propis arguments i a les dades **static** de la classe (perquè no existeix cap objecte en realitat i per tant els atributs **no statics** no existeixen).
  - No es poden sobreescriure (**sobrecarregar-los** en classes derivades) per fer-los **no static** en les classes derivades.
- Com a modificador d'iniciadors (blocs de codi sense nom), per aconseguir un iniciador que s'executi únicament quan es carrega la classe. La càrrega d'una classe es produeix en la primera crida d'un mètode de la classe, que pot ser el constructor involucrat en la creació d'un objecte o un mètode estàtic de la classe. La declaració d'una variable per fer referència a objectes de la classe no provoca la càrrega de la classe.
  La sintaxi a emprar és:

  static {...}

Exemple: la classe següent ens mostra una situació en què la declaració d'una dada **static** és necessària, ja que es vol portar un comptador del nombre d'objectes creats de manera que a cada nou objecte es pugui assignar un número de sèrie a partir del nombre d'objectes creats fins al moment.
Així mateix sembla oportú proporcionar un mètode, anomenat **nombreObjectesCreats()** per donar informació, com el seu nom indica, referent al nombre d'objectes creats de la classe en un moment donat.
Per acabar, s'ha inclòs un parell d'iniciadors per comprovar el funcionament dels iniciadors **static** i **no static**.

```
//Fitxer ExempleUsosStatic.java

public class ExempleUsosStatic {
   private static int comptador = 0;
   private int numeroSerie;
   static { System.out.println ("Iniciador \"static\" que s'executa en carregar
        la classe"); }

   { System.out.println ("Iniciador que s'executa en la creació de cada objecte
       "); }

   public ExempleUsosStatic () {
      comptador++;
      numeroSerie = comptador;
      System.out.println ("S'acaba de crear l'objecte número " + numeroSerie);
   }

   public static int nombreObjectesCreats () {
      return comptador;
   }

   public static void main(String args[]) {
      ExempleUsosStatic d1 = new ExempleUsosStatic();
      ExempleUsosStatic d2;
      d2 = new ExempleUsosStatic();
      System.out.println("Número de sèrie de d1 = " + d1.numeroSerie);
      System.out.println("Número de sèrie de d2 = " + d2.numeroSerie);
      System.out.println("Objectes creats: " + nombreObjectesCreats());
   }
}
```

L'execució del programa dóna el resultat:

```
Iniciador "static" que s'executa en carregar la classe
Iniciador que s'executa en la creació de cada objecte
S'acaba de crear l'objecte número 1
Iniciador que s'executa en la creació de cada objecte
S'acaba de crear l'objecte número 2
Número de sèrie de d1 = 1
Número de sèrie de d2 = 2
Objectes creats: 2
```

3. Llibreries de classes

Normalment, a l'hora de generar diferents classes, serà desitjable organitzar-les de manera que se'n pugui facilitar la gestió i saber quines estan relacionades entre si, per exemple, formant part d'un mateix programa. El llenguatge Java proporciona un mecanisme, anomenat **package**, per poder agrupar classes.

Cada classe dins un programa es representa normalment dins un fitxer amb extensió **.java** i amb un nom idèntic (incloent majúscules i minúscules) al de la pròpia classe.

3.1. Packages

La pertinença d'una classe a un paquet s'indica amb la sentència **package** a l'inici del fitxer font en què resideix la classe i afecta a totes les classes definides en el fitxer. La sentència **package** ha de ser la primera sentència del fitxer font. Abans hi pot haver línies en blanc i/o comentaris, però res més.
Sintaxi:

package <nomPaquet>;

Els noms dels paquets (per conveni, amb minúscules) poden ser paraules separades per punts, fet que provoca que els corresponents .class s'emmagatzemin en una estructura jeràrquica de directoris que coincideix, en noms, amb les paraules que constitueixen el nom del paquet.
Totes les classes d'un paquet anomenat "xxx.yyy.zzz" resideixen dins la subcarpeta "zzz" de l'estructura de directoris "xxx/yyy/zzz".

La inexistència de la sentència package implica que les classes del fitxer font es consideren en el paquet per defecte (sense nom).

Recordem que el codi incorporat en una classe (iniciadors i mètodes) té accés a tots els membres sense modificador d'accés de totes les classes del mateix paquet (a més de l'accés als membres amb modificador d'accés public).

En el disseny d'una classe es té accés a totes les classes del mateix paquet, però per accedir a classes de diferents paquets cal emprar un dels dos mecanismes següents:

- Utilitzar el nom de la classe precedit del nom del paquet cada vegada que s'hagi d'utilitzar el nom de la classe, amb la sintaxi següent:

  nomPaquet.NomClasse

- Explicitar les classes d'altres paquets a les quals es farà referència amb una sentència **import** abans de la declaració de la nova classe, seguint la sintaxi següent:

  import <nomPaquet>.<NomClasse>;

És factible carregar totes les classes d'un paquet amb una única sentència utilitzant un asterisc:

import <nomPaquet>.;*

Les sentències import en un fitxer font han de precedir a totes les declaracions de classes incorporades en el fitxer.