Abstracció i encapsulació

PID_00269652

David García Solórzano

Temps mínim de dedicació recomanat: 4 hores

Universitat Oberta de Catalunya

David García Solórzano

Graduat Superior en Enginyeria en Multimèdia i Enginyer en Informàtica per la Universitat Ramon Llull des de 2007 i 2008, respectivament. És també doctor per la Universitat Oberta de Catalunya des de 2013, en la qual va presentar una tesi doctoral relacionada amb l'àmbit de l'*e-learning*. Des de 2008 és professor de la Universitat Oberta de Catalunya en els Estudis d'Informàtica, Multimèdia i Telecomunicació.

L'encàrrec i la creació d'aquest recurs d'aprenentatge UOC han estat coordinats pel professor: David García Solórzano (2020)

Primera edició: febrer 2020

Av. Tibidabo, 39-43, 08035 Barcelona
Realització editorial: FUOC

Índex

Introducció

En aquest mòdul veurem dos dels quatre pilars del paradigma de la programació orientada a objectes: l'abstracció i l'encapsulació.

Gràcies a aquests aprendrem com pensar seguint un enfocament *bottom-up*. Com a conseqüència, aprofundirem en què és una classe i quins en són els seus membres, què és un objecte, què caracteritza a un objecte (estat i comportament), com s'instancia, què són els nivells o modificadors d'accés, quins hi ha i quines implicacions tenen.

Finalment, introduirem el llenguatge de modelatge de sistemes de *software* UML (*Unified Modeling Language*). En aquest mòdul ens centrarem exclusivament en com representar una classe dins d'un diagrama de classes UML.

Tret que s'indiqui un llenguatge de programació concret, els exemples de codificació estan escrits amb un llenguatge de programació inventat, és a dir, un pseudocodi. Si haguéssim de dir a quin llenguatge de programació real s'assembla el pseudocodi emprat, diríem que és semblant al Java (però sense elements que dificultin l'enteniment dels exemples).

Així doncs, haurem de consultar la documentació del llenguatge de programació que volem utilitzar per a veure com es codifiquen els conceptes explicats i els exemples proporcionats.

Objectius

L'objectiu principal d'aquest mòdul és assentar les bases del paradigma de la programació orientada a objectes, concretament:

1. Veure el procés d'abstracció que cal seguir a l'hora de dissenyar una classe i, per tant, per a solucionar un problema mitjançant el paradigma de la programació orientada a objectes.

2. Saber diferenciar entre un objecte i una classe i, al mateix temps, comprendre la relació que hi ha entre ells.

3. Conèixer i entendre el procés d'encapsulació per a agrupar els membres d'una classe, és a dir, els atributs i els mètodes. Així mateix, comprendre els beneficis que l'encapsulació aporta.

4. Entendre els conceptes de *constructor* i *destructor* d'una classe.

5. Conèixer l'ocultació com un mecanisme que defineix diferents nivells d'accés als membres d'una classe.

6. Conèixer els conceptes relacionats amb un objecte: instància, estat, comportament i missatge.

7. Saber què significa i quines implicacions té que un atribut, un mètode, un constructor o una classe siguin estàtics.

8. Tenir un primer contacte amb el llenguatge de modelatge UML.

1. Del problema al disseny d'una solució

Quan abordem un problema, aquest està format per objectes (alguns més tangibles i altres menys). Seran aquests objectes i les relacions entre ells el que ens permeti dissenyar un programa que doni resposta o solució al problema.

1.1. Què és un objecte?

Com ja t'hauràs imaginat, l'**objecte** és l'element principal de la programació orientada objectes i entorn del qual gira aquest paradigma, per això el nom.

Encara que acabes de començar a llegir aquests materials, atura't un moment. És igual on siguis, aparta la mirada d'aquesta pàgina i mira al teu voltant, però torna, que has de continuar llegint! Què has vist? Potser respons: «coses». Bé, no està malament, però pots ser més concret?

Exemple 1 (objecte) – Buscant objectes

Posem que ets al despatx on estudies o, millor, al sofà del menjador (l'estudi no és incompatible amb la comoditat). Potser has vist una taula, una cadira, dos llums, cinc llapis, un ordinador, un gos, etc. Podem dir que el que has vist al teu voltant són objectes? Sí? Genial!

Un moment, has vist aquest cotxe blanc que acaba de passar per davant de la finestra? Efectivament, és un objecte compost de molts altres objectes: un volant, quatre rodes, etc. Correcte, un cotxe és un objecte tan complex que està format o compost per altres objectes.

I què me'n dius de l'escriptori que tens al despatx? És clar que sí, també és un objecte! A més, està format per tres calaixos, cadascun dels quals és un objecte.

Espera! Acaba de passar un altre cotxe? Vaja, sí que hi ha trànsit! Ja has vist dos cotxes, ja has vist dos objectes similars.

Què és aquest soroll? És el teu *smartphone*! Si us plau, silencia les notificacions, estàs estudiant! Sí, sí, tens raó, el teu *smartphone* és un objecte que a dins té una llista d'aplicacions, i sí, cada aplicació és un objecte.

En aquest punt, segurament t'estàs dient: «D'acord, sé el que és un objecte en la vida real! Però, què s'entén per objecte en el paradigma de la POO?». Per a respondre't, ho farem de manera formal i informal. La formal ens porta a la definició següent:

Definició formal d'objecte

Un objecte en POO representa alguna entitat de la vida real, és a dir, algun dels objectes únics que pertanyen al problema amb el qual ens estem enfrontant i amb el qual podem interactuar.

Cada objecte, d'igual manera que l'entitat de la vida real que representa, té un estat (definit per uns atributs que tenen uns valors concrets) i un comportament (és a dir, té funcionalitats o sap fer unes accions concretes).

Entitat

Els objectes del món real se solen anomenar *entitats* per a diferenciar-los dels seus homòlegs en el món de la programació orientada a objectes, anomenats *objectes*.

De manera informal podem dir:

Definició informal d'objecte

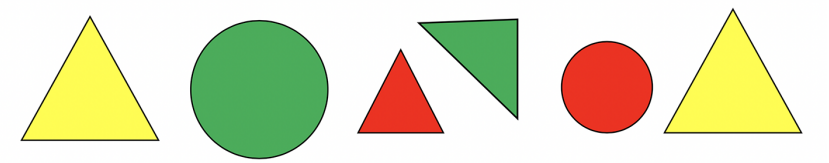
Un objecte en POO és qualsevol element del món real amb el qual es pot interactuar.

Així doncs, el primer cotxe que ha passat per davant de la teva finestra és un objecte, mentre que el segon cotxe (encara que sigui del mateix color, marca i model que l'anterior) és un altre objecte diferent.

Per acabar d'entendre què és un objecte, vegem l'exemple següent.

Exemple 2 (objecte) – Jugant amb els objectes

A la Marina, la seva mare Elena li ha posat els objectes següents al davant i li diu que digui quants objectes hi ha:



La Marina els mira, comença a comptar «un, dos, tres...» i acaba dient «sis». «Correcte!» –crida contenta l'Elena. La Marina veu que, encara que a simple vista hi ha dos objectes idèntics (els dels extrems), aquests són dos objectes diferents.

1.2. Què caracteritza a un objecte?

Com hem llegit en la definició formal d'*objecte*, tot objecte en la POO té un **estat** i un **comportament**. Això és així perquè els objectes (o entitats) de la vida real comparteixen aquestes dues característiques.

Definició d'estat

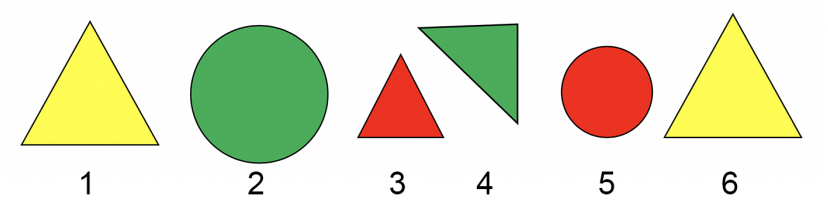
L'estat d'un objecte està determinat pels valors que prenen en un instant determinat els atributs que defineixen aquest objecte.

Així doncs, si un televisor concret (l'objecte) té l'atribut `canal actual` igual a 5 , aquest televisor està en un estat diferent de si tingués el `canal actual` igual al número `6`.

Vegem el concepte d'*estat* utilitzant el joc de les formes amb el qual jugaven la Marina i l'Elena.

Exemple 3 (estat d'un objecte) – Estat en el joc de les formes

Fixa't en els sis objectes següents:



Totes les figures tenen un atribut o propietat en comú: el color. Això sí, cada objecte té un valor assignat per a aquesta propietat. Per exemple, els objectes 1 i 6 són grocs, mentre que els objectes 2 i 4 són verds, i els objectes 3 i 5 són vermells. Així mateix, els objectes 1, 3, 4 i 6 tenen tres atributs, cadascun dels quals representa un costat del triangle i el valor del qual és igual a la longitud d'aquest costat. Aquestes longituds poden variar, com és el cas dels objectes 1 i 6 respecte als objectes 3 i 4. A més, els objectes 1, 3, 4 i 6 tenen un altre atribut que representa l'angle de rotació, essent el seu valor 0º per als objectes 1, 3 i 6, i un valor diferent per a l'objecte 4.

Així doncs, les diferències en els valors dels atributs d'un objecte fan que aquest objecte estigui en un estat diferent. Els objectes 1 i 6 estan en el mateix estat (tots els seus atributs tenen els mateixos valors), però els objectes 3 i 4 estan en un estat diferent.

Ara vegem què s'entén per *comportament*:

Definició de comportament

És el conjunt de funcionalitats que un objecte és capaç de dur a terme. Aquestes funcionalitats són determinades pels mètodes.

Per exemple, les accions que pot fer un televisor són les següents: encendre's, apagar-se, canviar de canal, pujar i baixar el volum, etc.

Seguim amb l'exemple en el qual l'Elena i la Marina juguen amb objectes de diferents formes.

Exemple 4 (comportament d'un objecte) – Comportament en el joc de les formes

Les sis figures tenen una àrea, per tant, són capaces de calcular-la a partir dels valors dels seus atributs. Si bé és cert que les figures 1, 3, 4 i 6 usaran la fórmula $b * h/2$, les figures

2 i 5 usaran πr^2. Les sis figures també podran calcular el seu perímetre amb les seves respectives fórmules. A més, les figures 1, 3, 4 i 6 podran calcular les coordenades del seu baricentre. Així doncs, l'àrea i el perímetre són dos mètodes que comparteixen totes les figures, mentre que el baricentre és exclusiu dels objectes 1, 3, 4 i 6. De la mateixa manera, els objectes 1, 3, 4 i 6 poden rotar, mentre que els objectes 2 i 5, no.

1.3. Què és una classe?

El concepte *classe* està íntimament relacionat amb el concepte *objecte*.

Definició de classe

Podem definir informalment una classe com una plantilla (o esquelet o plànol) a partir de la qual es creen els objectes.

Vegem la relació que hi ha entre classe i objecte amb un exemple.

Exemple 5 (relació entre classe i objecte) – El televisor del Josep

El David va a casa del seu amic Josep i s'adona que té el mateix televisor que ell. El mateix? El mateix no. Són dos objectes diferents, el David té el seu televisor a casa seva i el Josep el seu a casa seva. Que potser no pots tocar-los i veure que són dos objectes diferents? Això sí, són dos objectes (dos televisors) que són de la mateixa marca i model; per tant, semblen el mateix objecte, ja que cadascun d'aquests dos televisors ha estat muntat a partir d'un mateix plànol, esquelet o plantilla i, consegüentment, tots dos tenen els mateixos components, connexions i funcionalitats. Aquest plànol, esquelet o plantilla és, en termes de programació orientada a objectes, una classe (la classe `Television`, que representa el concepte abstracte de televisor, és a dir, les característiques i accions comunes dels televisors), mentre que cada televisor és un objecte d'aquesta classe.

El mateix ocorre amb l'exemple següent.

Exemple 6 (relació entre classe i objecte) – El plànol d'una casa

L'Elena és arquitecta i li han fet un encàrrec. Li han demanat que dissenyi una casa unifamiliar amb la qual urbanitzar tot un barri. En aquest escenari, el plànol que l'Elena dibuixi amb tots els detalls de la casa seria en POO la classe, mentre que cada casa que es construeixi a partir d'aquest plànol, per molt que s'assemblin, seria en POO un objecte. De fet, en el plànol de l'Elena posa que el `terra` de l'habitació pot ser de `parquet` o de `gres`. Així doncs, hi haurà cases per a les quals el valor de l'atribut `terra` serà `gres` i d'altres el valor del qual serà `parquet`. No obstant això, a partir del mateix plànol (i.e. classe) s'han creat totes les cases (i.e. objectes).

Finalment:

Exemple 7 (relació entre classe i objecte) – La reunió familiar

Si som en una reunió familiar, cada persona de la família és única i s'hi pot interactuar. És a dir, cada persona d'aquesta reunió és un objecte. Així doncs, la `Marina`, la seva mare `Elena` i el seu pare `David` són, cadascun d'ells, objectes. Encara més, si el besavi i l'avi paterns de la `Marina` es diuen tots dos `Manuel`, cadascun d'ells és un objecte diferent, ja que, si bé es diuen igual, no són la mateixa persona (és a dir, el mateix objecte). El besavi i l'avi, com la resta d'integrants de la família, són elements amb els quals es pot interactuar de manera individual. Això sí, tots els membres de la família són del mateix tipus `Person` (que és la classe a la qual pertanyen). En el cas d'aquesta família, hi ha dos objectes que comparteixen el valor `Manuel` per a l'atribut `nom`.

Així doncs, en aquest punt ha de quedar molt clar que el *teu* gos, el *teu* televisor i el *teu* bolígraf són objectes diferents del *meu* gos, el *meu* televisor i el *meu* bolígraf, encara que siguin de la mateixa raça, model i color, respectivament. Encara més, si tens dos gossos de la mateixa raça, per molt iguals que siguin, cadascun és únic i, per tant, són dos objectes diferents.

Vídeo d'interès

Per acabar d'entendre la relació entre els conceptes *classe* i *objecte*, us recomanem que vegeu el vídeo «Classe, objecte, atribut i mètode» que trobaràs a l'aula de l'assignatura.

1.4. Abstracció

Com vam dir en el mòdul anterior, per a dissenyar un programa basat en el paradigma orientat a objectes hem de seguir un enfocament *bottom-up*. Això significa anar dels objectes a les classes; és a dir, a partir dels objectes del problema, haurem d'obtenir les classes d'aquests objectes. Per a fer-ho, haurem de seguir un procés d'abstracció. És per això que:

> Una *classe* és l'abstracció o definició d'un conjunt d'objectes similars de la vida real.

Per **abstracció** entenem:

> **Definició d'abstracció**
>
> El procés mental d'identificar les entitats (és a dir, objectes), així com els seus estats (atributs) i comportaments (mètodes) que són rellevants per al problema que estem tractant. Així doncs, també consisteix a ignorar tots els aspectes que són irrellevants en el context en el qual estem treballant.

Per tant:

> Una *classe* descriu les característiques i comportaments comuns d'un conjunt d'objectes similars en un context o problema determinat.

Per a abstreure les classes a partir d'objectes concrets, cal **entendre molt bé el problema que volem resoldre** i **quedar-se solament amb l'estrictament necessari**. Així doncs, el context ho és tot, el context determinarà quines classes hi ha i quins atributs i mètodes definim per a cadascuna d'elles.

Exemple 8 (abstracció) – Obrint un compte d'estalvis

El David ha anat al banc per obrir un compte d'estalvis. La persona que atén al David necessita demanar-li certa informació personal per a poder gestionar la creació del nou compte. De la informació següent que enumerem, què creus que el banc necessita saber?:

- Nom
- Al·lèrgies alimentàries
- Cognom
- Adreça postal

- DNI
- Grup de música favorit
- Telèfon de contacte
- Nombre de fills

Efectivament, el David com a objecte té molta informació, però realment necessita el banc totes aquestes dades? La resposta és «no». Per a obrir un compte, el banc no necessita conèixer les al·lèrgies alimentàries del David (si les té), ni el seu grup de música favorit ni el nombre de fills.

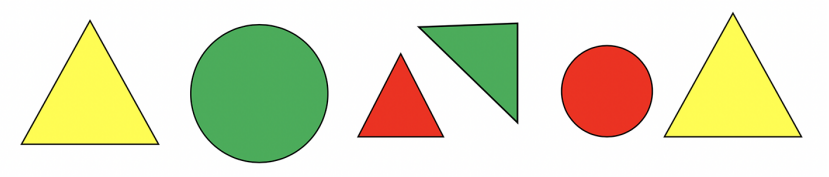
Ara imagina que, en comptes d'obrir un compte d'estalvis, el David ha de sotmetre's a una operació quirúrgica. Llavors l'hospital segurament necessitarà les mateixes dades que en el cas del banc més alguna d'addicional com, per exemple, les al·lèrgies alimentàries (ara sí), el grup sanguini, etc.

En tots dos contextos podríem haver dit que el David era un objecte de la classe `Person`, però les dades o atributs que necessitaríem serien diferents (i els mètodes, també). Per tant, en ambdós contextos, la classe `Person`, tot i dir-se igual, l'haguéssim definit de manera diferent, és a dir, amb diferents atributs i mètodes.

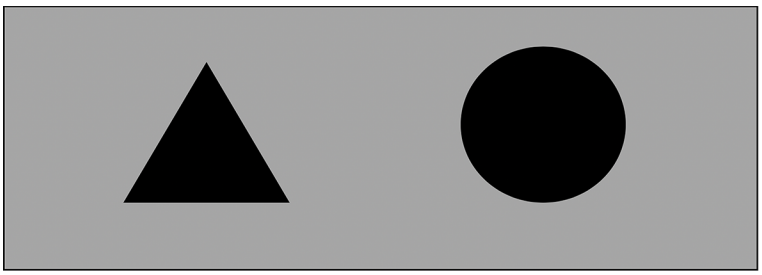
Vegem un altre exemple a partir del joc de les formes a què jugaven la Marina i l'Elena.

Exemple 9 (abstracció) – Encaixant formes

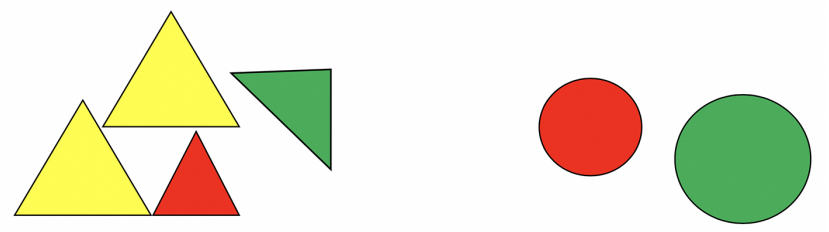
A continuació, l'Elena demana a la Marina que agrupi els objectes següents.



per a introduir-los en la caixa següent amb dos orificis:



Al cap d'una estona, la petita Marina els agrupa de la manera següent:



Quin criteri ha seguit? Ha usat la *forma* com a criteri d'agrupació. Mitjançant un procés d'abstracció, la Marina s'ha adonat que, d'una banda, hi ha uns objectes que tenen tres costats i uns altres que no en tenen cap. S'ha adonat que aquesta característica permet resoldre el problema al qual s'enfronta millor que, per exemple, el color dels objectes.

Ara, la seva mare Elena assenyala a la Marina els objectes de l'esquerra i li pregunta:

–Marina, què són?

La Marina es queda pensativa i contesta:

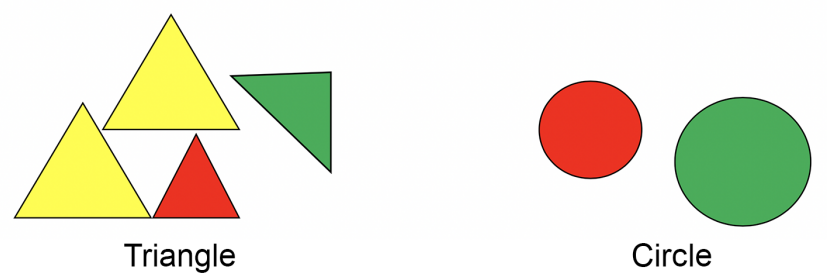
–Triangles! –diu amb llengua de drap.

–Molt bé! –exclama l'Elena sorpresa. I aquests altres? –pregunta l'Elena assenyalant el grup d'objectes de la dreta.

–Rodones –contesta la Marina, després de dubtar uns segons.

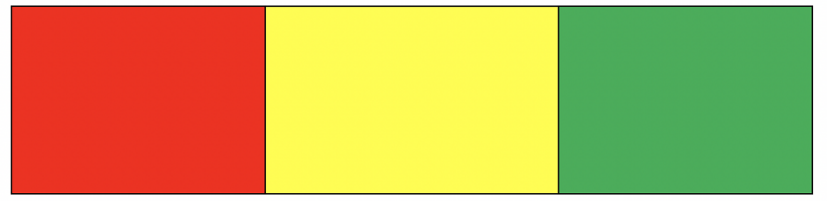
–Genial, Marina, són rodones, cercles! –diu l'Elena.

La Marina ha fet un pas més en el procés d'abstracció i ha vist que diferents objectes comparteixen «alguna cosa» en comú. Aquesta «cosa» és una **classe**. Així, als objectes de l'esquerra els ha assignat la classe `Triangle`, i als de la dreta, la classe `Circle`.

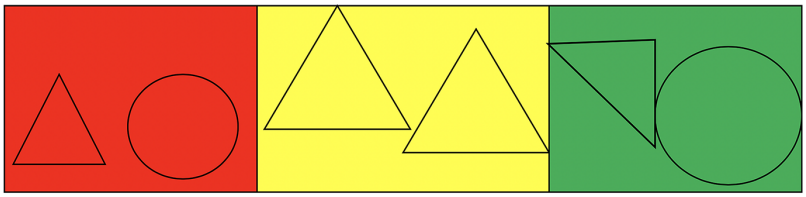


Per tant, de la classe `Triangle` hi ha quatre objectes, i de la classe `Circle` n'hi ha dos. Totes dues classes són diferents. Com ja hem dit, en el cas dels triangles i dels cercles, se'n pot calcular el perímetre i l'àrea, però els primers són els únics dels quals es pot calcular el baricentre o que es poden girar. Igualment, hi ha atributs comuns com el color, però també n'hi ha de diferents: longitud dels costats enfront del radi.

Si el problema que s'hagués presentat a la Marina hagués estat desar els sis objectes a la caixa següent:



llavors els hauria agrupat de manera diferent, i les classes, juntament amb els atributs i mètodes, segurament haurien estat unes altres (potser `Red`, `Yellow` i `Green`).



És molt important adonar-se que els mateixos objectes es poden agrupar de manera diferent (és a dir, abstreure en diferents classes) depenent del problema que calgui resoldre.

L'**abstracció** és un dels pilars en els quals es fonamenta el paradigma de la programació orientada a objectes. L'abstracció es fa en la fase de disseny i consisteix a centrar-se solament en les dades i comportaments dels objectes que són rellevants per a solucionar el problema al qual ens enfrontem. Gràcies a ai-

xò, simplifiquem la complexitat del problema obtenint una representació més simple i abstracta de la realitat i, alhora, augmentem l'eficiència. L'abstracció se centra en què fan uns objectes, no en com ho fan. Vegem-ne un exemple:

Exemple 10 (abstracció) – Definint un cotxe en anglès

Com a propòsit d'aquest any, el David s'ha apuntat a classes d'anglès, la seva assignatura pendent. En una d'aquestes classes, la professora li pregunta «què és un cotxe?» (*what is a car?*). El David balboteja i comença a parlar amb el reduït vocabulari que té: «És un vehicle que es pot moure». De primeres, el David descriu l'objecte en termes més abstractes dient que es mou, però sense dir com: usant pneumàtics, volant, navegant, etc. Solament es mou, no necessitem més detalls per a tenir una primera idea de l'objecte. Després, el David diu: «és terrestre i té quatre rodes» i «en general, té cinc seients». Refina l'abstracció de l'objecte *cotxe* per diferenciar-lo d'altres objectes com poden ser una *moto* o un *camió*, però sense entrar en gaires detalls. Així doncs, un cotxe és «un vehicle terrestre que té quatre rodes; en general, té cinc seients i es pot moure». Com són les rodes i com estan posades?, i els seients?, com es mou?, amb benzina o amb electricitat?, com passa la força del motor a les rodes? Tant és, això és com ho fa, no què fa.

Finalment, vegem més exemples en els quals abstraiem les característiques o atributs i accions o mètodes de diferents entitats (o objectes) de la vida real. Una vegada més, ens adonarem que qualsevol entitat o objecte té un estat i un comportament:

- Els *gossos* tenen un estat (atributs: nom, raça, color, etc.) i un comportament (mètodes: bordar, moure la cua, enterrar ossos, etc.).

- Els *cotxes* també tenen un estat (velocitat actual, marxa actual, color, longitud, amplada, etc.) i un comportament (arrencar, pujar marxa, baixar marxa, encendre intermitent, etc.).

- D'igual manera, els *televisors* tenen un estat (encès o apagat, canal actual, volum actual, etc.) i un comportament (encendre, apagar, canviar a un canal concret, incrementar el nombre de canals, disminuir el nombre de canals, augmentar el volum, disminuir el volum, sintonitzar, etc.).

- També les *factures* tenen un estat (cobrada o no, import total, paga i senyal abonada, etc.) i un comportament (canviar de no estar cobrada a estar cobrada i viceversa, modificar el valor de l'import total, etc.).

- Fins i tot quelcom una mica més abstracte o intangible com són els *contactes* del telèfon tenen un estat (nom, cognom, telèfon, correu electrònic, etc.) i un comportament (introduir un atribut –és a dir, nom, cognom, telèfon, correu electrònic, etc.–, modificar el valor d'un atribut i consultar el valor d'un atribut).

- Què ens dius dels triangles amb els quals jugaven la Marina i l'Elena? Els *triangles* tenen un estat (tres costats amb una longitud cadascun –depenent d'això el triangle serà isòsceles, escalè, etc.–, un color, etc.) i un comportament (està girat, el càlcul de la seva àrea, canviar el valor de la longitud d'un costat, etc.).

Fins ara, solament hem abstret, per a cada conjunt d'objectes similars (per exemple gossos, cotxes, etc.), estats i comportaments comuns (o, dit d'una manera més senzilla, atributs i mètodes). De moment no ens hem preocupat de com implementarem aquests estats i comportaments. El «com» el veurem amb **l'encapsulació**.

Vídeo d'interès

Per acabar d'entendre el concepte d'*abstracció*, us recomanem que vegeu el vídeo «Abstracció i paradigma bottom-up» que trobaràs a l'aula de l'assignatura.

Finalment, si ens fixem en els exemples anteriors, ens adonarem que hi ha dos tipus de mètodes:

1) Els que fan accions que du a terme l'entitat real (per exemple, bordar en el cas del gos, arrencar el motor en el cas d'un cotxe, l'acció d'encendre's d'un televisor, etc.).

2) Els que operen directament sobre els atributs de l'objecte. D'una banda, hi ha els mètodes que modifiquen el valor dels atributs de l'objecte (per exemple, el número de telèfon d'un contacte o el canal actual d'un televisor) i que, per tant, canvien l'estat de l'objecte. D'altra banda, hi ha els mètodes que consulten el valor dels atributs de l'objecte (per exemple, el número de telèfon d'un contacte o el canal actual d'un televisor). Aquests mètodes de modificació i consulta són anomenats, en l'àmbit de la programació, ***setter*** i ***getter***, respectivament.

2. Encapsulació: del disseny a la implementació

Si hem dit que l'abstracció se centra en el disseny, l'encapsulació se centra en la implementació.

Definició d'encapsulació

És el procés de tancar o empaquetar els atributs i mètodes dins de classes, que són entitats més grans i més abstractes.

L'objectiu principal de l'encapsulació és fer que un sistema complex sigui més senzill de manejar per part de l'usuari final. Gràcies a l'encapsulació s'aconsegueix:

- Protegir les dades i el codi ocultant els detalls interns de la implementació.
- Facilitar l'ús de les classes i la reutilització del seu codi.
- Fer transparent el manteniment de les classes (per exemple, fer canvis interns) per a l'usuari final.

Abans de veure com aconsegueix els beneficis anteriors, vegem com es defineix una classe seguint el concepte d'*encapsulació* i com es creen objectes a partir de la classe.

2.1. Membres d'una classe

Com hem vist, un objecte està format per atributs que n'estableixen l'estat i mètodes que en limiten el comportament. Hem dit que, gràcies a l'encapsulació, els atributs i mètodes d'un conjunt d'objectes similars els agrupem en una **classe**. Al seu torn, el binomi format pels **atributs** i els **mètodes** es denomina **membres d'una classe**. Així doncs, tant un atribut com un mètode són membres d'una classe.

2.1.1. Atributs

Els atributs, també anomenats *camps* (*fields*), són com variables que codifiquen l'estat d'un objecte.

Classe sense atributs

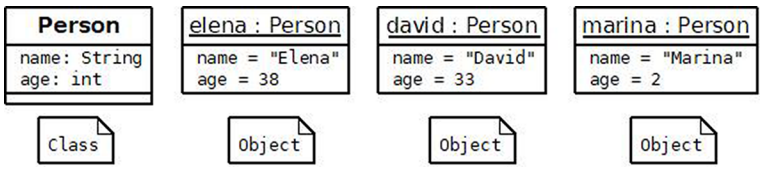
Una classe pot no tenir atributs.

Si tenim la classe `Person` amb els atributs `name` i `age` –de tipus cadena de caràcters i enter, respectivament–, cada objecte que es defineixi del tipus `Person` tindrà aquests dos atributs.

Tipus d'atributs

Un atribut pot ser de tipus bàsic o primitiu (`int`, `char`, etc.) o d'un tipus de classe concreta, per exemple, `Person`. El tipus dels atributs es defineix com qualsevol altra variable.

L'estat de cada objecte `Person` en un instant determinat dependrà dels valors que s'assignin a aquests dos atributs, tal com es veu en la figura següent:



Encara que dos objectes comparteixin el mateix estat –és a dir, mateixos valors per a tots els atributs–, aquests dos objectes són diferents. Solament cal pensar que pot haver-hi dues persones anomenades `David` amb `33` anys d'edat al món i, òbviament, són persones (o objectes) diferents.

2.1.2. Mètodes

Els mètodes implementen el comportament d'un objecte o, dit d'una altra manera, les funcionalitats que un objecte és capaç de realitzar. Si fem una analogia amb la programació estructurada, els mètodes serien com les funcions (retornin alguna cosa o no). Per aquest motiu, un mètode, a més de pel nom també es caracteritza pels paràmetres que rep i pel valor que retorna. La descripció d'aquests elements es coneix com a **firma del mètode o signatura del mètode**. En pseudocodi seria:

```
functionName(paramName1:type, …, paramNameN:type):returnType
```

En el cas de la classe `Person`, alguns mètodes podrien ser:

```
talk(text:vector[30] of char):void
walk(speed:integer):void
```

El patró que segueix la signatura dels mètodes depèn de cada llenguatge de programació.

En aquest punt, cal esmentar el concepte de **sobrecàrrega**.

La sobrecàrrega es produeix quan dos o més mètodes tenen el mateix nom, però es diferencien en la llista de paràmetres. Aquesta diferenciació ha de donar-se en el nombre de paràmetres, en el tipus dels paràmetres o en l'ordre dels paràmetres.

Un exemple de sobrecàrrega del mètode `talk` podria ser:

```
talk(text:vector[30] of char):void
talk(text:vector[30] of char, speed:integer):void
```

Sobrecàrrega en PHP

En PHP, la sobrecàrrega de manera nativa no existeix. Per a «simular» una sobrecàrrega, s'ha d'usar la funció màgica (*magic function*) anomenada `__call()`.

El compilador decideix quin mètode (quin dels dos `talk`) s'invoca comparant els arguments de la crida amb els paràmetres de la signatura.

No obstant això, la sobrecàrrega següent donaria error:

```
talk(text:vector[30] of char, speed:integer):void
talk(text:vector[30] of char, volume:integer):void
```

Per què? Perquè en cridar `talk("Elena", 50)`, el compilador no sabria quin dels dos mètodes cridar: el `50` és velocitat o volum? Tu ho saps? El compilador tampoc.

Per a solucionar aquest error, es podria canviar l'ordre dels paràmetres en la segona signatura, per exemple:

```
talk(volume:integer, text:vector[30] of char):void
```

2.2. Constructor i destructor

Les classes tenen dos tipus de mètodes especials anomenats **constructor** i **destructor** que no es consideren membres d'una classe com a tals. No se'ls considera membres d'una classe perquè ni el constructor ni el destructor s'hereten.

Vegeu també

El concepte d'*herència* s'estudia en el mòdul «Herència (relacions entre classes)» d'aquesta assignatura.

La majoria dels llenguatges de programació orientats a objectes implementen el mètode constructor, fins i tot alguns obliguen a codificar-ne explícitament un. No ocorre el mateix amb el destructor, la codificació del qual es pot obviar en molts llenguatges, per exemple, en Java.

2.2.1. Constructor

El constructor és el mètode especial que hem de cridar per a crear un objecte. Amb aquesta crida, l'objecte és situat en la memòria i s'inicialitzen els atributs declarats en la classe. En la majoria de llenguatges, el constructor té les característiques següents:

1) Normalment, el nom del constructor és el mateix que el de la classe.

Nom del constructor diferent del nom de la classe

Des de la versió 5.3.3 de PHP, el constructor es defineix mitjançant un mètode anomenat `__construct()`; en canvi, en Python, s'ha de fer servir el mètode especial `__init_()_`.

2) El constructor no té tipus de retorn, ni tan sols `void` (el tipus `void` vol dir que no retorna res).

3) En la majoria de llenguatges, el constructor pot rebre arguments amb la finalitat d'inicialitzar els atributs de la classe per a l'objecte que s'està creant en aquest moment.

4) En general, és públic, però alguns llenguatges permeten que sigui protegit o privat. Veurem els conceptes *públic*, *protegit* i *privat* en l'apartat 2.5 d'aquest mòdul.

Tenint en compte les quatre característiques anteriors, un exemple de constructor per a la classe `Person` podria ser:

```
public Person(String nameParam){//Java
     name = nameParam;
     age = 33;
}
```

Hi ha llenguatges que permeten crear més d'un constructor, com C++, C# i Java, entre altres. En aquests casos, el constructor sense paràmetres se sol anomenar **constructor per defecte o predeterminat**, mentre que els que tenen paràmetres s'anomenen **constructors parametritzats o amb arguments**. Com es pot apreciar, dir constructor «per defecte» i «parametritzat» és el mateix que dir que es fa una sobrecàrrega del constructor. A causa de la sobrecàrrega, l'única limitació quan es vol (i es pot) definir més d'un constructor és que no poden declarar-se diversos constructors amb el mateix nombre i el mateix tipus de paràmetres (és a dir, les mateixes limitacions que hem explicat per a la sobrecàrrega de mètodes).

En els llenguatges en què solament es pot codificar un constructor, per exemple, Python i PHP, aquest s'anomena simplement constructor.

Cal destacar que en molts llenguatges no és obligatori que una classe tingui un constructor per defecte. Depenent del context, pot interessar-nos que tots els constructors d'una classe siguin amb arguments.

Finalment, cal dir que en molts llenguatges no és obligatori definir explícitament un constructor per a una classe. En aquests llenguatges, si no es defineix cap constructor per a la classe, el mateix compilador crearà un constructor per defecte –és a dir, sense arguments– que no farà res d'especial més enllà de situar l'objecte en memòria. No obstant això, en el moment en què el programador implementa un constructor (per defecte o amb arguments), el compilador no afegeix automàticament el constructor per defecte.

2.2.2. Destructor

El destructor és un mètode especial que es crida durant l'execució del programa quan l'objecte s'està destruint, eliminant o alliberant de la memòria. Podem dir que és l'últim mètode que es crida abans que desaparegui l'objecte de memòria. En molts llenguatges, aquest mètode es crida de manera automàtica quan ocorren una sèrie de circumstàncies. Així mateix, cal tenir en compte que:

Crida al destructor en C++

Per a eliminar un objecte en C++ s'ha d'usar l'operador `delete` seguit del nom de l'objecte, p. ex. `delete david`. L'operador `delete` cridarà el destructor de la classe `Person` a la qual pertany l'objecte `david`.

1) No tots els llenguatges obliguen a implementar un mètode destructor. En aquests casos, si no se'n codifica un explícitament, el compilador en crea un per defecte.

2) Per norma general, una classe té solament un destructor.

3) En alguns llenguatges no té tipus de retorn, ni tan sols `void`. En altres, generalment té `void` com a tipus de retorn.

4) No té paràmetres.

5) En general, és públic.

La manera en què es declara el mètode destructor varia segons el llenguatge. Per exemple, en C++ i C#, el nom del destructor és el mateix que el de la classe precedit pel símbol `~`, per exemple `~Person()`. En altres llenguatges s'usa un mètode especial que es comporta com un destructor, encara que no és un destructor pròpiament dit; per exemple, en Java s'utilitza el mètode especial `finalize()`.

Algunes tasques habituals que se solen fer dins d'un destructor i per les quals n'haurem d'implementar explícitament un són:

- Tancar fitxers, *sockets* o connexions a bases de dades que l'objecte ha obert.
- Alliberar recursos compartits que l'objecte té bloquejats (molt típic en programació distribuïda i concurrent).
- Alliberar memòria, per exemple, punters que l'objecte ha demanat.

2.3. Creant instàncies d'una classe

Com ja hem comentat, els objectes són exemplars d'una classe i, per tant, es creen a partir de la definició de la seva classe. A l'hora de crear un objecte hem de seguir els passos següents:

1) Declarar una variable per a l'objecte que volem crear. Aquesta declaració, en molts llenguatges, haurà de tenir un tipus (que serà el nom d'una classe) i un identificador (és a dir, un nom). En altres llenguatges, no fa falta indicar-ne el tipus (és a dir, la classe).

```
ClassName instanceName; //Java, C#
instanceName; //Python, PHP (no es defineixen els tipus explícitament)
```

Per exemple:

```
Person david; //Java, C#
david //Python
$david; //PHP, les variables van precedides de $
```

2) Crear l'objecte en memòria (també anomenat **instanciar** l'objecte). Per a fer-ho es crida un dels constructors que s'hagi definit dins de la classe. Hi ha llenguatges que utilitzen una paraula especial per a cridar el constructor, per exemple, la paraula reservada `new`.

Instància o objecte?

A causa que l'acció de crear un objecte a partir d'una classe s'anomena *instanciar*, moltes vegades els objectes s'anomenen *instància*. Així doncs, `david` és una «instància» o un «objecte» de `Person`.

```
david = new Person("David"); //Java, C#
david = Person("David") //Python
$david = new Person("David"); //PHP
```

Els passos 1 i 2 es poden agrupar en un únic pas (alguns llenguatges obliguen a fer-ho així, per exemple C++):

```
Person david = new Person("David"); //Java, C#
Person david("David"); //C++
val david = new Person("David"); //Scala
david = Person("David") //Python
$david = new Person("David"); //PHP
```

Si ens quedéssim en el pas 1, és a dir, solament en la declaració, i no féssim la instanciació, llavors l'objecte no seria construït en memòria i, en general, el compilador li assignaria el valor `null`.

Una vegada s'ha cridat el constructor, tenim l'objecte `david` de tipus `Person` creat en memòria amb els atributs i mètodes de la classe `Person` copiats. A partir d'aquest moment ja podem accedir als seus atributs i mètodes. Els atri-

buts i mètodes d'un objecte concret, per exemple, `david`, s'anomenen **membres de la instància**. Això és així perquè aquest objecte o instància té la seva pròpia còpia dels atributs i dels mètodes, els quals ocupen una zona de memòria diferent dels d'un altre objecte `Person`. Així, si instanciéssim un segon objecte anomenat `elena`, en modificar-ne l'atribut `name`, no estaríem modificant l'atribut `name` de l'objecte `david`.

2.4. Missatge

Quan els objectes volen interactuar entre ells utilitzen **missatges**. Un missatge és la manera que hi ha d'accedir als atributs i mètodes d'un objecte (és a dir, als membres de la instància). La forma d'un missatge, en la majoria de llenguatges, segueix la sintaxi següent:

```
instanceName.member
```

en què `member` pot ser un atribut o un mètode. Per exemple:

```
elena.name = "Elena";
david.talk("Hola!!");
elena.talk(david.name);
```

En la majoria dels llenguatges per a crear un missatge (és a dir, accedir a un atribut o mètode) s'utilitza l'operador punt (`.`). En altres llenguatges s'utilitzen altres símbols. Per exemple, en PHP s'utilitza `->`, és a dir, `$david->talk()`.

2.5. Ocultació d'informació

Un dels mecanismes íntimament relacionats amb l'encapsulació és l'**ocultació d'informació**. De fet, moltes persones fan dels termes *encapsulació* i *ocultació d'informació* sinònims i, no obstant això, el primer inclou el segon. O, dit d'una altra manera, l'ocultació d'informació és una conseqüència directa de l'encapsulació. Per *informació* s'entén tant els atributs com els mètodes.

Terminologia

En altres explicacions veuràs el concepte d'*ocultació* referenciat com a «visibilitat dels membres d'una classe» o «restricció d'accés a atributs i mètodes» o, especialment en documentació centrada en la programació, «modificadors d'accés».

L'objectiu que es persegueix amb l'ocultació és:

> Fer que una classe es comporti com una caixa negra que proporciona un conjunt de serveis al programador que la utilitza, ocultant-li tots els detalls que no interessa que sàpiga o que hi tingui accés. És a dir, el programador usuari solament ha de conèixer i poder accedir directament als atributs i mètodes estrictament necessaris, res més. D'aquesta manera, se separa o diferencia el que un objecte pot fer de la implementació real de com l'objecte ho fa.

Vegem dos exemples per entendre el concepte d'*ocultació*.

Exemple 11 (ocultació d'informació) – La pastilla

Quan estem malalts i el metge ens recepta una pastilla, les dues úniques coses que ens interessen *a priori* com a usuaris són:

1) què guareix aquesta pastilla,

2) com s'administra (oralment o pel recte, freqüència, quantitat, etc.).

És a dir, usem la pastilla i ja està, no ens preocupa amb quins ingredients està elaborada. Sabem que la pastilla blava és la de la tos i la vermella la de la tensió; en coneixem l'aspecte (interfície o exterior), la utilitat i la forma d'administració, però no sabem res sobre la seva composició (és a dir, del seu interior).

Exemple 12 (ocultació d'informació) – Càpsules de cafè

El mateix ocorre amb les càpsules de cafè que tan de moda s'han posat. D'una banda, sabem que el color de la càpsula –que és la interfície– ens indica el sabor i la intensitat del cafè i, d'una altra, sabem que la manera d'utilitzar la càpsula és posant-la d'una determinada manera dins de la cafetera compatible amb aquestes càpsules. En aquest punt, poc sabem sobre la composició exacta del cafè que hi ha a l'interior, però com a usuaris tampoc no ens interessa gaire, sabem que el cafè resultant és bo i ens el prenem.

El fet d'encapsular la pastilla i la càpsula de cafè fa que la seva utilització per part de l'usuari final sigui senzilla. Vegem un exemple més extens:

Exemple 13 (ocultació d'informació) – Cotxe

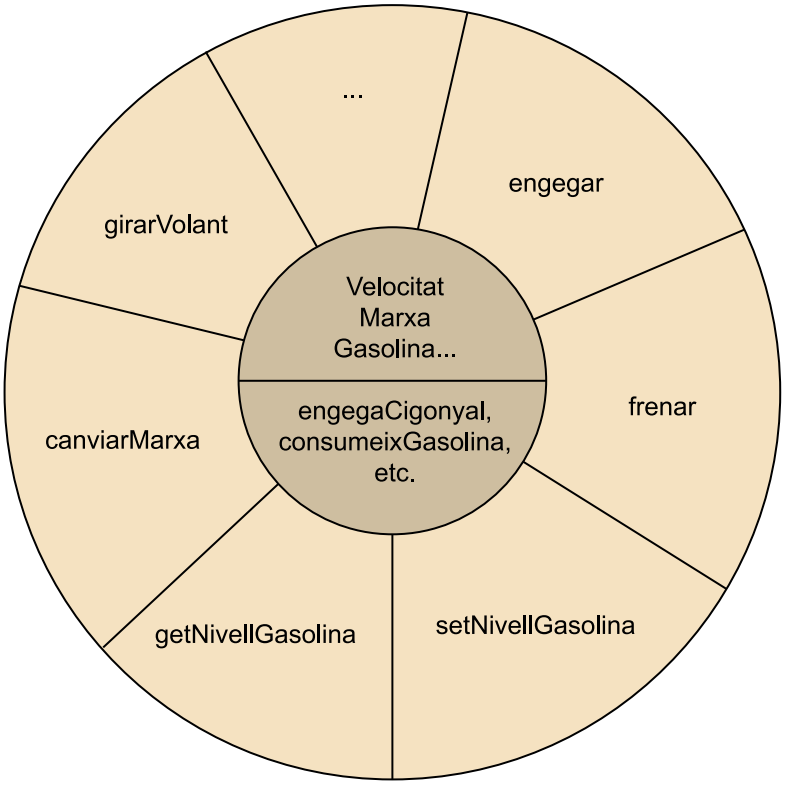
Si bé és cert que quan obtenim el permís de conduir de cotxe se suposa que tenim unes nocions de mecànica, la realitat és que la majoria de conductors no tenen ni idea de com funciona el seu cotxe per dins. Això no els limita a l'hora de conduir el vehicle.

El conductor sap com s'utilitza el volant, com es canvia de marxa, perquè serveix cada pedal i quan ha d'usar-lo, sap on és la palanca que activa els intermitents, els llums, etc. En definitiva, el conductor coneix la interfície del vehicle, no la seva implementació; és a dir, no sap quins mecanismes s'activen perquè en prémer el botó del parabrisa aquest s'engegui. D'igual manera, sap que ha d'introduir la clau en el contacte i girar-la perquè el cotxe s'engegui, però desconeix quines accions realitza la mecànica del cotxe per arrencar i engegar-se.

Si veiéssim el cotxe com una classe de la POO, podríem dir que els mètodes `arrencar()`, `frenar()`, `girarVolant(dretaOEsquerra:boolean, graus:float)`, etc. són la part visible i, per tant, coneguda pel conductor (és a dir, pel programador que usa la classe `Car`). No obstant això, quan el conductor crida el mètode `arrencar()`, per dins, aquest mètode crida molts altres mètodes ocults que el conductor desconeix –per exemple, `arrancaCigonyal`, `consumeixGasolina`, etc.– i que solament coneix el mecànic (és a dir, el programador que ha dissenyat i codificat la classe `Car`).

Així mateix, la classe `Car` té atributs que com a conductors podem consultar i modificar mitjançant mètodes *getter* i *setter*. Per exemple, el nivell de gasolina del dipòsit és un atribut ocult que consultem mitjançant el mètode `getNivellGasolina()`, que ens retorna el valor en el panell d'informació del vehicle en forma de nombre o d'agulla. També som capaços de modificar-lo quan repostem en una gasolinera.

De manera gràfica i resumida, un cotxe (classe `Car`) vist com una classe POO seria:



En la part exterior hi hauria els mètodes i atributs visibles, i en la interior, els ocults.

Hi ha diferents nivells d'ocultació (o «modificadors d'accés», *access modifiers* o *access level modifiers*). Podem restringir l'accés als atributs i mètodes en diferent mesura. En general, la majoria dels llenguatges de programació orientats a objectes defineixen tres modificadors d'accés (va ser el llenguatge C++ qui els va introduir) que estableixen tres nivells d'accés o ocultació diferents: públic, privat i protegit. Abans d'explicar en què consisteixen aquests tres modificadors d'accés, cal emfatitzar tres aspectes clau:

1) Quan parlem de visibilitat o ocultació d'un atribut o mètode d'una classe, l'hem d'entendre en relació amb la resta de les classes.

2) Una classe sempre pot accedir a tots els atributs i mètodes definits en ella.

3) La interacció entre dues classes es pot fer o bé mitjançant la interacció de dos objectes d'aquestes dues classes (interactuen usant missatges, com veurem més endavant) o bé mitjançant el mecanisme d'herència (ho veurem en l'últim mòdul d'aquests materials).

2.5.1. Públic

Si un atribut o mètode és públic en una classe `A`, s'hi pot accedir des de fora de la classe `A`. Dit d'una altra manera, qualsevol altra classe pot accedir a l'atribut o mètode usant en el seu codi un objecte de tipus o classe `A`. Si volem definir un element com a públic, en la majoria de llenguatges usarem la paraula reservada `public`.

```
class Person{
     public int age = 5;
     public int getAge(){
         return age;
     }
}
```

En l'exemple anterior estem definint un atribut anomenat `age` que és de tipus `int` (enter), el valor d'inicialització del qual és `5` i el seu nivell d'accés és públic. A més, estem implementant un mètode públic anomenat `getAge` que no rep paràmetres, però sí retorna un enter, concretament, el valor de l'atribut `age`.

Exemple 14 (ocultació d'informació) – Public

```
class MainApp{
  public void main(){
   Person david = new Person("David");
   david.getAge(); //retorna 5
   david.age = 36;
   david.getAge(); //retorna 36
  }
}
```

En una classe que fa de programa principal anomenada `MainApp` tenim un mètode `main` que utilitza un objecte anomenat `david` de la classe `Person`. Si imaginem que hem definit la classe `Person`, l'atribut `age` i el mètode `getAge()` com a públics, llavors hi podem accedir directament des de qualsevol lloc del nostre codi.

2.5.2. Privat

Quan un atribut o mètode és privat en una classe, només s'hi pot accedir des de dins de la mateixa classe que el defineix (la qual s'anomena *classe contenidora*), és a dir, que cap altra classe pot accedir a ell directament. Si volem definir un element com a privat, en la majoria de llenguatges usarem la paraula reservada `private`.

`Private` en C++

En C++ les classes declarades com a amigues (*friend classes*) també poden accedir als atributs o mètodes declarats com a privats en la classe de la qual és amiga.

```
     private int age = 5;
     public int getAge(){
         return age;
     }
```

Imagineu que en la classe `Person` hem fet la implementació anterior, és a dir, hem definit solament l'atribut `age` com a privat i, mentrestant, hem mantingut el mètode `getAge()` com a públic.

Exemple 15 (ocultació d'informació) – Private

```
class MainApp{
  public void main(){
   Person david = new Person("David");
   david.getAge(); //retorna 5
   david.age = 36; //error, no s'assigna
   david.getAge(); //si l'execució
   //continués, retornaria 5
   david.age; //error
  }
}
```

En aquest cas, el mètode `getAge()` codificat en la classe `Person` pot seguir accedint a l'atribut `age` perquè aquest atribut, encara que privat, està definit en la mateixa classe que `getAge()`.

No obstant això, des d'una altra classe, com `MainApp`, no podem accedir directament a l'atribut `age`. Si volem saber-ne el valor, haurem d'usar el mètode `getAge()`, que és públic.

`private` en Python

En Python, realment no existeixen els membres «privats» i, per tant, tampoc la paraula reservada `private`. No obstant això, hi ha la convenció d'usar la tècnica de *name mangling* (o *name decoration*), que consisteix a precedir el nom de l'atribut o mètode amb doble *underscore* (_), per exemple `self.__name`. Això fa que l'accés a l'atribut o mètode no sigui tan directe i sembli privat.

Accés a membres `private` en classes imbricades

Els diferents llenguatges de programació orientats a objectes defineixen diferents comportaments a l'hora de poder accedir directament o no als membres privats d'una classe des d'una classe imbricada dins aquesta.

2.5.3. Protegit

Quan un atribut o mètode és protegit en una classe, a aquest sol s'hi pot accedir des de dins de la mateixa classe que el defineix (classe contenidora), així com des de classes filles o derivades (o subclasses) de la classe contenidora.

El concepte de *classe filla* o *derivada* (o *subclasse*) es veurà en el mòdul «Herència (relacions entre classes)» quan vegem el mecanisme d'herència.

`protected` en Python

En Python no existeix la paraula reservada `protected`. En el seu lloc es precedeix l'atribut o mètode amb un *underscore* (_), per exemple, `self._name`.

2.5.4. Hi ha més modificadors d'accés?

Sí, però depèn del llenguatge de programació. Per exemple:

- En C# hi ha el modificador d'accés `internal`, que limita l'accés a l'`assembly` (unitat física de composició, equivalent a un `.jar`, `.dll` o binari) actual. A més a més, permet combinar dos modificadors o nivells: `protected internal` i `private protected`.

- En Java hi ha el nivell d'accés `package-private`, que no té paraula clau (*keyword*), simplement s'assumeix en no escriure cap nivell d'accés.

2.5.5. Solament podem assignar modificadors d'accés als membres d'una classe?

La resposta és «no». Hi ha diferents elements d'un programa orientat a objectes, a part dels atributs i mètodes, als quals es pot assignar un modificador d'accés. Això depèn del llenguatge. Normalment, aquests elements es divideixen en dos dominis o nivells:

- **Nivell alt:** classes, interfícies, `namespace`, `enum`, etc.
- **Nivell de membre:** atributs i mètodes (incloent constructors i destructors).

Vegeu també

El concepte d'*interfície* es veurà en el mòdul «Herència (relacions entre classes)».

Així mateix, els modificadors d'accés que es poden aplicar varien segons el llenguatge i el domini o nivell de l'element al qual li aplicarem. Normalment, els elements de nivell alt (és a dir, classes, interfícies, etc.) tenen menys opcions. Per exemple, en Java, una classe o interfície quan és un element de nivell alt no pot ser declarada `protected` ni `private`; solament pot ser `public` o `package-private`. No obstant això, quan la classe o interfície està imbricada dins d'una segona classe, llavors també podem usar `protected` o `private`, ja que la classe o interfície ha deixat de ser un element de nivell alt i es converteix en un «membre» més d'una altra classe, és a dir, com si fos un atribut o un mètode d'aquesta altra classe.

Per la seva banda, en C#, els `namespaces` són implícitament públics i no es permet assignar cap modificador d'accés, per la qual cosa el nivell d'accés públic no pot ser modificat. Així mateix, C# solament permet assignar els modificadors `internal` i `public` als elements d'alt nivell si aquests no estan imbricats, i `internal` és l'accés per defecte (i.e. si no s'indica res). En cas d'estar imbricats, llavors sí que permet assignar-los altres modificadors d'accés, els quals dependran del tipus d'element (classe, interfície, `struct`, etc.) al qual s'imbricarà.

2.5.6. I si no indiquem el modificador d'accés?

Si a un element no li assignem un modificador d'accés, aleshores el nivell que s'aplica o assumeix depèn del llenguatge de programació que estiguem usant. A més, depèn de què estem delimitant: un atribut, un mètode, una classe, una interfície, etc. Vegem-ne alguns exemples:

- **Classes:** C# les considera `internal`; mentre que Java, `package-private`. C++, Python i TypeScript no conceben un altre nivell d'accés que no sigui `public`. Per a Scala, l'única manera de dir que alguna cosa és `public` és no indicant un modificador d'accés.

- **Atributs i mètodes d'una classe:** TypeScript, Python i Scala, per exemple, els consideren `public`, mentre que C++ i C# els considera `private`. Per la seva banda, Java els considera `package-private`.

2.5.7. Quin modificador d'accés s'assigna habitualment a cada element?

A continuació indiquem quins són els nivells o modificadors d'accés més habituals per a cada element, la qual cosa no vol dir que no se'n puguin aplicar altres depenent del problema que cal resoldre.

a) Classes i interfícies. En general, es declaren com a públiques, però en alguns llenguatges també poden ser privades o protegides (solament quan estan imbricades). En molts llenguatges, el nivell d'accés `protected` per a classes i interfícies no té sentit si la classe o interfície no està imbricada i sol no estar permès. En canvi, en llenguatges com Scala, sí que pot haver-hi classes que siguin `private` o `protected` a nivell alt (és a dir, sense estar imbricades).

```
public class Person{}
```

b) Atributs. Usualment els programadors els declaren privats, encara que poden ser públics i protegits. Fer-los protegits és interessant quan hi ha herència entre classes.

```
private int age;
```

c) Mètodes. El més habitual és declarar-los o bé públics o bé privats. Els declararem públics quan vulguem que es puguin usar des de fora de la classe (per exemple, mitjançant un objecte d'aquesta classe), i privats, quan siguin d'ús intern de la classe. No obstant això, igual que els atributs, també poden ser protegits.

```
public int getAge(){}
private void doSomething(int param){}
```

d) Constructors. Típicament són declarats com a públics. No obstant això, també poden ser privats i protegits. Aquest últim cas és interessant quan la classe que té el constructor protegit és una superclasse d'una altra classe.

```
public Person(){}
```

Vegeu també

El concepte de *superclasse* es veurà en el mòdul «Herència (relacions entre classes)» quan vegem el mecanisme d'herència.

2.6. Protecció de les dades

A continuació intentarem exemplificar la utilitat de restringir l'accés al codi, concretament als atributs, i veure com n'és d'útil usar els modificadors d'accés per a aconseguir un comportament estable i desitjat (segur) dels nostres programes. També quedarà de manifest la importància d'usar mètodes *getter* i *setter* per a accedir als atributs.

Exemple 16 (protecció de les dades) - Per què es comporten de manera diferent?

Imagina que a un amic li han proporcionat ja feta una classe `A` amb un atribut públic anomenat `valueA` de tipus `int`. A més, aquesta classe té un constructor per defecte i dos mètodes públics, un que és un *getter* `getValueA()` que retorna el valor de `valueA` i un *setter,* anomenat `setValueA(int val)`, que rep un enter i l'assigna a l'atribut `valueA`.

Ell et diu que ha desenvolupat una classe `B` que té dos objectes de tipus `A` i que en ella fa el següent:

```
public class B{
     public A objectA1, objectA2;

     public B(){
          objectA1 = new A();
          objectA1.valueA = 100; //Assignem directament perquè és públic
          print(objectA1.getValueA()); //Imprimeix 100
          print(objectA1.valueA); //Imprimeix 100

          objectA2 = new A();
          objectA2.setValueA(100); //Assignem a través del setter
          print(objectA2.getValueA()); //Imprimeix 300
          print(objectA2.valueA); //Imprimeix 300
     }
}
```

T'explica que no entén per què amb l'objecte `objectA1` el programa imprimeix `100` i amb `objectA2` imprimeix `300`. Sabries dir-li per què pot ser?

La resposta és que amb `objectA1` està accedint directament a l'atribut i l'està modificant. En canvi, el mètode `setValueA(int val)`, el que fa és multiplicar per tres el valor que se li passa com a paràmetre i després assignar-lo a l'atribut `valueA`. Evidentment, quan es fa l'assignació directament (com es fa amb `objectA1`), aquesta multiplicació no es du a terme.

Gràcies a l'exemple anterior, veiem que l'encapsulació no solament oculta informació o restringeix l'accés als elements, sinó que també assegura que les dades siguin tractades correctament forçant al fet que s'usin els mètodes que hem programat. Amb les dades/atributs encapsulats amb un modificador d'accés `private`, qualsevol càlcul o comprovació que la classe faci sobre les dades serà realitzat, ja que no es pot accedir a ells directament.

Seguim amb l'exemple anterior per acabar de veure com n'és d'útil l'encapsulació:

Exemple 17 (protecció de les dades) – Per què de nou no es comporta igual?

Imagina que en comptes d'assignar 100, ara assignem -100.

```
objectA1.valueA = -100; //S'assigna sense problemes
objectA2.setValueA(-100); //Dona un error
```

Per què la segona línia dona un error? Doncs perquè segurament dins del mètode `setValueA(int val)` s'està comprovant que no s'assigni un valor negatiu, perquè per als objectes d'aquesta classe està prohibit. Imagina que `valueA` fos `age`, és obvi que ningú té una edat negativa. Amb l'assignació directa no podríem controlar-ho mitjançant el codi de la classe, sinó que ho hauríem de controlar, allà on fem l'assignació, de la manera següent (la qual cosa ens fa repetir trossos de codi similars milers de vegades en el nostre programa, i això no és eficient):

```
int val = -100;

if(val>=0){
      objectA1.valueA = val;
}else{
      print "ERROR";
}
```

2.7. Facilitat d'ús i reutilització

Si partim de la classe `A` de l'Exemple 16, veurem que podem usar aquesta classe en qualsevol dels nostres programes. Així, si la classe `A` fos de veritat la classe `Customer`, allà on tinguéssim clients, per exemple, un restaurant, hotel, botiga, etc., podríem usar aquesta classe per a crear objectes de tipus `Customer`. Gràcies a l'encapsulació podem veure una classe com un element autocontingut que és *plug and play*. Potser per a cada context calgui fer ajustos (per exemple, afegir algun mètode o atribut), però el gruix del que fa un client ja ho tindríem implementat.

2.8. Transparència als canvis

Si en una classe fem petits canvis dins d'un mètode, aquests canvis haurien de ser transparents per a l'usuari final (és a dir, el programador que usa la classe). Així doncs, el programador que usa la classe no hauria de modificar el seu codi, ja que el mètode modificat seguirà funcionant i, per tant, les crides a aquesta seran correctes.

Imagina que el mètode `setValueA(int val)` de la classe `A` d'abans, a més de comprovar que el valor passat per paràmetres és positiu i assignar a l'atribut `valueA` el triple del valor passat, també incrementa un atribut privat que acabem de crear per saber quantes vegades s'ha cridat `setValueA(int val)` per a l'objecte en qüestió. Aquest canvi és tan lleu que no s'haurà de modificar el codi del programa que utilitza la classe `A`.

Si hi ha canvis més grans, llavors caldrà veure com afecta. Per exemple, eliminar un mètode és un canvi important, per això, el que se sol fer, abans d'eliminar un mètode, és marcar-lo com `deprecated` o `obsolete`, segons el llenguatge, en la versió de la classe just anterior a la versió en la qual s'elimina i indicar que s'utilitzi el nou mètode que el substitueix. D'aquesta manera, el programador sap que de moment pot continuar usant el mètode, però que en versions futures de la classe aquest mètode no existirà i el seu programa deixarà de funcionar si l'utilitza.

Així doncs, l'encapsulació també ens ajuda a poder fer canvis en les classes i que els canvis, si són lleus, siguin transparents per als usuaris finals. Per tant, podem dir que un altre objectiu de l'encapsulació és:

Ocultar al món exterior el treball intern dels objectes perquè pugui canviar-se més tard sense afectar els usuaris externs.

Vídeo d'interès

Per entendre millor el concepte d'encapsulació, com també els nivells bàsics d'ocultació (és a dir, els modificadors d'accés), us recomanem que vegeu el vídeo «Encapsaluació» que trobaràs a l'aula de l'assignatura.

3. Elements estàtics (*static*)

3.1. Atributs i mètodes estàtics

Hem vist que, gràcies a l'abstracció i l'encapsulació, podem agrupar en una classe els atributs i mètodes que comparteixen diferents objectes. Així mateix, hem comentat que quan instanciem un objecte d'una classe, aquest objecte se situa en memòria tenint la seva pròpia còpia dels atributs i mètodes de la classe a la qual pertany. És per això que diem que aquests atributs i mètodes són membres de la instància. Cada nou objecte de la classe tindrà els seus membres de la instància, independents de la resta d'objectes de la mateixa classe.

La pregunta ara és la següent: podem tenir un atribut o un mètode que sigui compartit per tots els objectes de la classe? La resposta és «sí». Molts llenguatges de programació orientats a objectes –com C++, Java, C#, etc.– inclouen el modificador `static`, amb el qual podem forçar que l'atribut o mètode sigui un **membre de la classe** i no de la instància. Això implica que aquest atribut o mètode sigui comú a tots els objectes i, per tant, ocupi una única zona de memòria i s'eviti que hi hagi una còpia d'aquest atribut o mètode per a cada objecte que es crea. El fet que un atribut o mètode sigui estàtic continua permetent que cada objecte de la classe pugui accedir a l'atribut o mètode, però també ens permet accedir a l'atribut o mètode sense necessitat de crear un objecte de la classe.

Membres de la classe en altres llenguatges

Hi ha llenguatges, com ara Python, que no tenen el modificador `static`, però permeten declarar atributs o mètodes com a estàtics. En el cas de Python, per exemple, els atributs inicialitzats en la seva declaració dins de la classe són considerats *static*, mentre que si són inicialitzats en un constructor o mètode, llavors són *membres de la instància*. Per als mètodes s'ha d'usar el decorador `@staticmethod`.

Exemple 18 (*static*) – Atribut estàtic I

```
public class A{
     private int id;
     private static int counter  = 0;

     public A(){ //constructor
          id = counter;
          counter = counter + 1;
     }
}
```

L'atribut `counter` és un membre de la classe, no de la instància. Així doncs, és compartit per tots els objectes de la classe `A`. Si creem `2` objectes de tipus `A`, el valor de `id` del primer objecte serà `0` i el del segon `1`; i, una vegada creat el segon objecte, el valor de l'atribut `counter` serà `2` per als dos objectes, perquè `counter` és de la classe.

Vegem un altre exemple, però ara amb dues classes:

Exemple 19 (*static*) – Atribut estàtic II

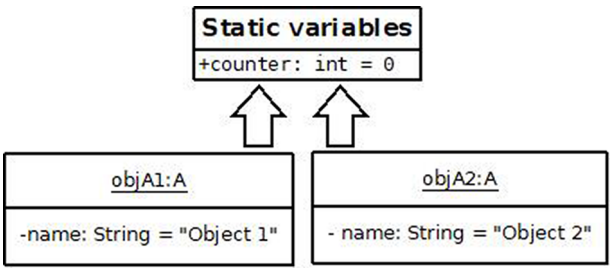
```
public class A{
     public static int counter  = 0;
     private String name;

     public A(String n){
          name = n;
     }
}

public class B{
     private A objA1;
     private A objA2;

     public B(){//constructor
          A.counter = A.counter+1;
          objA1 = new A("Object 1");
          print objA1.counter; //1
          print A.counter; //1
          objA1.counter = 5;
          print A.counter; //5
          objA2 = new A("Object 2");
          print objA2.counter; //5
     }
}
```

L'atribut `counter` és un membre de la classe `A`, no de la instància. Així doncs, és compartit per tots els objectes de la classe `A`.

En el constructor de la classe `B` accedim a l'atribut `counter` de la classe `A` mitjançant el nom de la classe `A` i mitjançant dos objectes de la classe `A`, `objA1` i `objA2`. Com que l'atribut `counter` solament existeix una vegada en memòria i és de la classe –la qual cosa significa que és compartit per tots els objectes de la classe `A` –, és igual com hi accedim, perquè sempre estarem modificant la mateixa «còpia» de l'atribut `counter`.

En aquests casos es recomana accedir a l'atribut amb el format `className.staticField` perquè quedi clar que és un membre de la classe i no de la instància.

Visualment seria així:



Els objectes `objA1` i `objA2` comparteixen la variable estàtica `counter`, però tots dos objectes tenen el seu atribut d'instància `name`, és a dir, la seva pròpia còpia de l'atribut `name`.

Ara vegem un exemple de mètode estàtic. D'igual manera que els atributs estàtics, els mètodes estàtics també són compartits per tots els objectes de la classe. Així mateix, s'hi pot accedir mitjançant un objecte de la classe o mitjançant el nom de la classe. Tal com hem dit per als atributs estàtics, es recomana accedir als mètodes estàtics amb el nom de la classe.

Exemple 20 (*static*) – Mètode estàtic

```
public class A{
     private static int counter  = 0;

     public static int getCounter(){
          return counter;
     }
     public static void setCounter(int c){
          counter = c;
     }
}

public class B{
     private A objA; //objecte de la instancia

     public B(){//constructor
          A.setCounter(5);
          objA = new A();
          print objA.getCounter(); //5
          print A.getCounter(); //5
          objA.counter = 5; //error
          print A.counter; //error
     }
}
```

En aquest exemple hem declarat l'atribut `counter` com a `private`. Així doncs, l'única manera d'accedir-hi és mitjançant els mètodes *getter* i *setter*.

En general, hi ha unes regles que se solen complir en els llenguatges de programació que permeten l'ús del modificador `static` amb atributs i mètodes:

- Un mètode de la instància (és a dir, no estàtic) pot accedir directament tant a atributs i mètodes estàtics com no estàtics.

- Un mètode de la classe (és a dir, estàtic) no pot accedir directament a un atribut o mètode de la instància (és a dir, no estàtic). Per a poder accedir a aquests necessita usar un objecte de la classe. Això és obvi perquè es pot accedir a un mètode estàtic sense necessitat d'un objecte i, per tant, en aquest cas no sabria, per exemple, quin valor tenen els atributs no estàtics que depenen de la instanciació d'un objecte.

- Els mètodes estàtics no poden ser sobreescrits, però poden ser sobrecarregats.

- Un mètode abstracte no pot ser declarat estàtic.

Vegeu també

Els conceptes de *sobreescriptura* i mètode *abstracte* s'explicaran en el mòdul «Herència (relacions entre classes)».

Quan és útil declarar estàtic un atribut o mètode? En el cas d'un atribut, serà útil declarar-lo estàtic quan la seva finalitat sigui comptar el nombre d'instàncies o objectes que hem declarat d'una classe. També serà útil quan guardi un valor que sigui comú a tots els objectes de la classe, per exemple, el valor del bitllet dels autobusos d'una companyia (tots els objectes `Bus` cobren el mateix preu per viatge). Per què caldria guardar el mateix valor tantes vegades com objectes hi hagi? Això seria una pèrdua de memòria innecessària.

D'altra banda, és necessari tenir molta cura amb l'ús d'atributs `static`, ja que podem caure en l'error d'usar-los com a variables globals –és a dir, que estiguin disponibles en qualsevol part del nostre programa–, amb els problemes que ja sabem que això implica.

En el cas dels mètodes, aquests es declaren estàtics quan l'operació o acció que duen a terme és independent de la creació d'una instància o objecte.

3.2. Constructor estàtic

En alguns llenguatges com C# també podem fer estàtic un constructor. En el cas de C#, si creem un constructor estàtic, aquest es cridarà automàticament la primera vegada que es creï un objecte d'aquesta classe. Aquest constructor no pot tenir arguments ni se li ha d'assignar un modificador d'accés. Un constructor estàtic sol usar-se per a inicialitzar qualsevol atribut estàtic de la classe o fer alguna acció que necessiti fer-se solament una vegada per als objectes de la classe. A més, el constructor estàtic no s'hereta.

Exemple 21 – Constructor estàtic

```
public class A{
     static A(){ print "hola"; }
     public A(int a){ print a;}
     public void main(){
          //Imprimirà "hola" i després 5;
          A obj1 = new A(5);
          //Imprimirà 6.
          A obj2 = new A(6);
     }
}
```

Atès que el primer objecte de la classe `A` que instanciem és `obj1`, en crear `obj1` amb el constructor no estàtic es crida primer el constructor estàtic i després el constructor públic. En canvi, per a `obj2`, en tractar-se ja del segon objecte de la classe `A`, solament es crida el constructor no estàtic.

3.3. Classe estàtica

En alguns llenguatges –per exemple, Java, PHP, C++ i C#– és possible declarar una classe com a estàtica usant el modificador `static`. El comportament d'una classe estàtica depèn del llenguatge de programació que estiguem usant. Per exemple, vegem les diferències entre Java i C#:

	Java	C#
Quines classes poden ser estàtiques?	Solament les classes que són dins d'una altra (és a dir, les classes imbricades, *nested classes*) poden ser estàtiques.	Qualsevol classe pot ser declarada estàtica.
Pot ser instanciada?	Sí	No

	Java	C#
Tipus de constructor	Solament constructor d'instància (i.e. el normal), ja que en Java no hi ha la possibilitat de declarar un constructor estàtic.	Solament constructor estàtic.
Pot ser heretada?	Sí	No
Pot heretar d'una altra classe?	Sí	No, solament d'`Object`.
Tipus d'atributs o mètodes que pot contenir	Tant estàtics com no estàtics.	Solament estàtics.

Així doncs, veiem que en C# el modificador `static` aplicat a una classe força tots els seus membres (i.e. atributs i mètodes) a ser estàtics; mentre que en Java significa que la classe (que és una *nested class*) és un membre estàtic de la classe en la qual està imbricada, com pot ser qualsevol altre atribut o mètode de la classe contenidora. Vegem-ne un exemple (el codi no és Java, però s'hi assembla):

Exemple 22 – Classe imbricada estàtica (estil Java)

```
public class A{
     private int a = 5;
     public static class B(){} //classe B nested en la classe A
}

public class C{
     public C(){
         A.B objB = new A.B(); //podem fer això.
     }
}
```

Altres llenguatges, com Scala i Kotlin, no tenen el modificador `static`, però permeten crear un comportament similar mitjançant altres tècniques, per exemple, usant *companion objects*.

4. Representació d'una classe i un objecte en UML

4.1. Classe

Per a definir una classe (i en definitiva, un programa) de manera formal i gràfica, se sol utilitzar el llenguatge UML (*Unified Modeling Language*). Aquest llenguatge inclou molts tipus de diagrames, per exemple: casos d'ús, diagrames de seqüència, etc. Nosaltres ens centrarem en el **diagrama de classes**.

Usar un diagrama basat en un llenguatge de modelització com UML permet llegir i entendre el que representa una classe sense que sigui necessari veure el codi.

UML és un llenguatge conceptual, per la qual cosa el **diagrama de classes** hauria de ser **el més independent possible del llenguatge de programació que s'utilitzi en l'etapa de codificació**. D'aquesta manera, un mateix diagrama de classes serveix per a qualsevol llenguatge de programació.

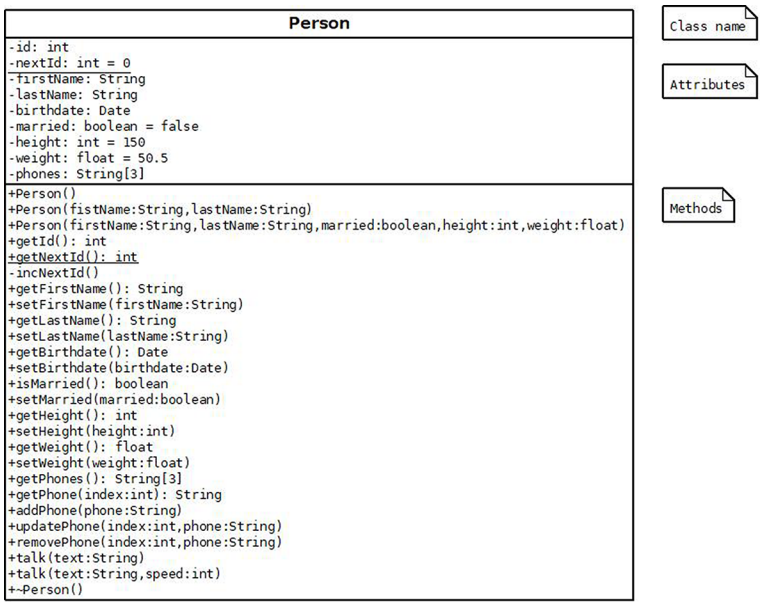
No obstant això, és molt freqüent trobar els diagrames de classes adaptats al llenguatge de programació que s'utilitzarà en l'etapa de codificació. Per exemple, si hem de desenvolupar el programa en Java, llavors sovint no s'escriu el destructor en el diagrama de classes, ja que en aquest llenguatge no existeix com a tal. En Java, si s'utilitzés, apareixeria `finalize()`.

També sol ocórrer que un diagrama de classes no pugui representar-se exactament en un llenguatge de programació. Per exemple, quan vegem les classes associatives, ens adonarem que la majoria de llenguatges no ens permeten implementar-les. En aquests casos, haurem de ser creatius i intentar traduir el diagrama de classes a codi el millor que sapiguem.

Vegeu també

El concepte de *classe associativa* s'estudia en el mòdul «Associacions (relacions entre objectes)» d'aquesta assignatura.

Una vegada introduït què és UML, vegem com es representa una classe en un diagrama de classes. En aquest tipus de diagrames, una classe es representa com una caixa amb tres compartiments ben diferenciats:



1) Nom de la classe. El compartiment superior conté el nom de la classe. Aquest s'escriu en negreta i centrat. Per conveni, els noms de les classes s'escriuen amb la primera lletra en majúscula i en singular. Si el nom de la classe està format per diverses paraules, cada paraula ha de començar per majúscula, per exemple, `BankAccount`, `PrintStream`, `VipMember`, `HttpProxyServer`, etc.

2) Atributs. El compartiment del mig conté la definició dels atributs. Per a cada atribut, com a mínim, s'indica la visibilitat (o modificador d'accés), el nom i el tipus al qual pertany. A més, és possible indicar-ne el valor per defecte (d'inicialització). En el diagrama anterior, l'atribut `married` és de tipus `boolean` i el seu valor per defecte és `false`. Fixa't que els *arrays* s'indiquen amb el format tipus `[longitud]`; per exemple, `phones: String[3]` per a dir que és un *array* amb tres `String` (cadenes de caràcters).

Per conveni, el nom dels atributs és un substantiu o sintagma nominal. A més, la nomenclatura habitual a l'hora d'escriure el nom dels atributs és *lower camel case*, és a dir, escriure en minúscula la primera paraula i, si el nom de l'atribut està compost per dues o més paraules, llavors a partir de la segona paraula la primera lletra de cadascuna s'escriu en majúscula; per exemple, `lastName` (està composta de `last` + `name`), `height`, `resourceNumber`, `xMin`, `yTopLeft`, etc.

Com a tipus d'atribut solen utilitzar-se els tipus primitius habituals (és a dir, `int`, `float`, `double`, `char`, etc.), com també `String` (és la manera usual d'indicar una cadena de caràcters) o una classe proporcionada pel mateix llenguatge de programació (no creada per nosaltres); per exemple, `birthdate: Date` (aquest atribut és un objecte de la classe `Date`. Aquesta classe existeix tant en l'estàndard UML com en molts llenguatges de programació, p. ex. Java).

3) Mètodes. L'últim compartiment conté els mètodes de la classe, incloent els constructors i el destructor. En el cas de l'exemple, la classe `Person` té un constructor per defecte, dos constructors amb arguments i un destructor. Per a poder diferenciar el constructor per defecte del destructor, aquest últim es fa precedir del símbol ~; per exemple, `~Person()`.

Atès que els mètodes defineixen accions, s'ha establert per conveni que el nom dels mètodes sigui un verb o sintagma verbal, a excepció dels constructors i el destructor, ja que s'han d'anomenar exactament igual que la classe. Igual que els atributs, el nom dels mètodes segueix un estil *lower camel case*, és a dir, la primera paraula del nom s'escriu en minúscula i les següents comencen per majúscula; per exemple, `getHeight()`, `getNextValue()`, etc.

Els paràmetres dels mètodes segueixen la mateixa convenció de nom que els atributs i, a més, se n'ha d'indicar el tipus. Així mateix, els mètodes que retornen alguna cosa han d'indicar el seu tipus; per exemple, `getLastName():String`. En cas de no retornar res, l'habitual és no escriure un tipus de retorn; per exemple `talk(text:String)`. No obstant això, en alguns diagrames de classes es pot llegir `void` com a tipus de retorn, que vol dir que el mètode no retorna res.

Nomenclatura per a *getter* i *setter*

Per conveni, els mètodes de tipus *getter* se solen anomenar `get + nom de l'atribut`; per exemple, `getHeight` per a anomenar el *getter* de l'atribut `height`. Excepcionalment, els atributs de tipus `boolean` se solen anomenar `is + nom de l'atribut`; per exemple, `isMarried`. Per la seva banda, tots els *setter* se solen anomenar `set + nom de l'atribut`; per exemple, `setHeight` i `setMarried`.

D'altra banda, ens falta explicar com es representen els diferents nivells o modificadors d'accés o visibilitat en un diagrama de classes UML.

- El símbol + indica que és públic.
- El símbol – indica que és privat.
- El símbol # indica que és protegit.
- El símbol ~ indica que el seu nivell d'accés és de tipus paquet (`package`). El seu equivalent en Java seria el modificador d'accés `package-private`.

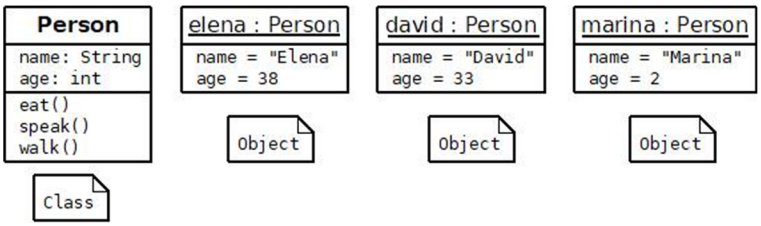
Finalment, cal dir que per indicar que un atribut o un mètode és *static* ho fem subratllant tota la definició de l'atribut o tota la signatura del mètode. Mira l'atribut `nextId` i el mètode `getNextId()` de la classe `Person` del diagrama de classes anterior.

En el cas de les classes estàtiques, no hi ha un estàndard, per la qual cosa es dona llibertat al dissenyador del diagrama de classes. És freqüent indicar que una classe és estàtica anteposant l'estereotip `<<static>>` abans del nom de la classe. Quan la classe solament conté atributs i mètodes estàtics, llavors és habitual usar l'estereotip `<<utility>>`, en comptes de `<<static>>`.

4.2. Objecte

Per a mostrar un objecte en un diagrama de classes, aquest es representa com una caixa amb dos compartiments. El primer d'aquests està format per un text d'una línia, tot ell subratllat, que inclou el nom de la instància, a continuació dos punts (:) i finalment el nom de la classe a la qual pertany.

El segon compartiment és l'estat de l'objecte, és a dir, els atributs que són membres de la instància amb els seus valors corresponents. Els mètodes no es posen perquè ja s'entén que són els de la classe a la qual pertanyen.



Resum

Classe

- Per a definir una classe (és a dir, els seus atributs i mètodes) s'ha de fer un exercici d'abstracció seguint un procés *bottom-up* (i.e. identificar primer els objectes i després les classes).

- Una classe és un element de *software* que representa un conjunt d'entitats del món real que són similars, p. ex., `Student`, `Ball`, etc.

- Per conveni, el nom d'una classe és un substantiu o sintagma nominal en singular escrit en *upper camel case* (o *Pascal case*), p. ex., `Student`, `Ball`, `VipAuthor`.

- Una classe encapsula en un únic element les dades (en forma d'atributs) i comportaments (en forma de mètodes) comuns de les entitats que representa.

- Per conveni, el nom d'un atribut és un substantiu o sintagma nominal, mentre que el nom d'un mètode és un verb o sintagma verbal. Tots dos casos s'escriuen en *lower camel case* (o *dromedary case*); p. ex., `numLifes`, `getParametersValues()`, etc.

- Gràcies a l'encapsulació, les classes són elements autocontinguts fàcils de reutilitzar.

- La majoria de llenguatges tenen, com a mínim, tres nivells de visibilitat (o modificadors d'accés): `public`, `protected` i `private`. Segons el llenguatge utilitzat, aquests es poden aplicar a diferents elements del programa: atributs, mètodes, classes, etc.

- En un diagrama de classes UML, una classe es representa com una caixa de tres compartiments: nom de la classe, atributs i mètodes.

- Els modificadors d'accés en un diagrama de classes UML es representen amb un `+ (public)`, `# (protected)`, `- (private)` i `~ (package)`.

- En general, quan el modificador `static` s'aplica a:
  - una classe, el seu comportament depèn del llenguatge de programació;

  - un atribut o mètode, aquest es converteix en membre de la classe (no de la instància) i, per tant, s'hi pot accedir sense necessitat de cre-

ar un objecte de la classe, solament usant `className.fieldName` o `className.methodName()`.

Objecte

- Un objecte és un cas particular o concret d'una classe, per exemple, `pau`, `mateu` i `sofia` són entitats concretes de la classe `Student`. Així doncs, `pau`, `mateu` i `sofia` són objectes de la classe `Student`.

- L'acció de crear un objecte i allotjar-lo en la memòria es diu *instanciar*. Per aquest motiu, els objectes també s'anomenen sovint *instàncies*.

- Per a instanciar un objecte s'ha de cridar un dels constructors de la classe. La manera de fer aquesta crida varia segons el llenguatge de programació utilitzat.

- Una vegada instanciat l'objecte en la memòria, aquest tindrà els seus propis atributs i mètodes, els quals anomenarem *membres de la instància*.

- Els valors que assignem als atributs de la instància o objecte constitueixen l'estat de l'objecte en un moment determinat.

- Els mètodes constitueixen el comportament o accions que pot tenir o fer un objecte. Aquests utilitzaran els valors que tinguin els atributs de la instància a què pertanyen.

- Per a cridar un membre de la instància, hem d'usar missatges. En la majoria de llenguatges, el patró per a crear un missatge és `instanceName.fieldName` o `instanceName.methodName()`.

- El destructor és un mètode especial que es crida just abans que l'objecte quedi eliminat de la memòria per sempre. Com es defineix aquest mètode depèn de cada llenguatge de programació. En la majoria de llenguatges, el destructor és cridat automàticament quan ocorren una sèrie de circumstàncies.

Activitats

Exercici 1
Pensa en un caixer automàtic. Aquest haurà de controlar els diners de què disposa i com els té distribuïts. Per exemple, si té 100 €, un client no podrà treure 200 € per molt que els tingui en el seu compte corrent. Així mateix, els 100 € els pot tenir en dos bitllets de 50 €, en deu bitllets de 10 € o mitjançant la combinació d'un bitllet de 50 € i cinc bitllets de 10 €. Tota aquesta gestió la farem amb la classe `Cash` (en català, 'Dispensador'). La classe `Cash` la definim de la manera següent:

a) Té tres dipòsits de bitllets (*notes*), un per als bitllets de 10 €, un altre per als de 20 € i un altre per als de 50 €.

b) Té un mètode anomenat `getBalance` que permet saber la quantitat de diners en euros que té en cada moment el caixer.

c) Té un mètode públic anomenat `toString` que retorna el text següent (`String`):

```
Amount: A €
[Notes 10 €: N10, Notes 20 €: N20, Notes 50 €: N50]
```

En què `A` és la quantitat de diners que té el dispensador en aquell moment i `N10`, `N20` i `N50` són el nombre de bitllets de 10 €, 20 € i 50 €, respectivament, que hi ha en aquell moment.

d) El constructor per defecte inicialitza els atributs de la classe a zero (`=0`) si són numèrics, a `false` si són `boolean` i a `null` si són d'un altre tipus.

e) Té un constructor amb arguments que té tants arguments com atributs té la classe.

f) Inclou tots els mètodes *getter* i *setter* dels atributs de la classe. Tots són públics.

g) Té un mètode anomenat `withdraw`, que rep per paràmetre una quantitat sense decimals de diners en euros, comprova si el caixer té aquesta quantitat entre tots els seus dipòsits i retorna un *array* de tres caselles (una per tipus de bitllet) amb la quantitat de bitllets de cada tipus que dispensa per a la quantitat sol·licitada. A més, actualitza la quantitat de diners que queda en cada dispensador després de fer l'operació. Per exemple, si es demanessin 60 € i el caixer tingués cinc bitllets de 10 €, zero de 20 € i un de 50 €, llavors retornaria un *array* en què la casella 0 (= 10 €) seria 1, la casella 1 (= 20 €) seria 0 i la casella 2 (= 50 €) seria 1. Després de l'operació, quedarien 4, 0 i 0 bitllets, respectivament. En cas de no poder donar bitllets, per la raó que sigui, imprimeix el text «`There are not enough notes`».

Dibuixa el diagrama de classes UML de la classe `Cash` indicant el nivell d'accés o visibilitat dels atributs i mètodes, com també els tipus dels atributs de la classe, dels paràmetres dels mètodes i dels valors de retorn dels mètodes.

Exercici 2
Ens demanen que creem una classe per a gestionar la informació de les diferents sucursals o locals que té una cadena de restaurants anomenada UocDonald's. Aquesta classe l'anomenarem `Restaurant` i la definirem de la manera següent:

a) Té un atribut que permet identificar numèricament la sucursal.

b) Té altres set atributs que són: nom de la sucursal (`name`), una adreça, que serà el carrer més el número (`address`), el nom de la ciutat en la qual es troba (`city`), un codi postal (`zip`), un telèfon (`phone`), un correu electrònic de contacte (`email`) i un *flag* que diu si la sucursal és oberta o no (`open`).

c) Té un mètode públic anomenat `toString` que retorna el text següent(`String`):

```
[ID: I, Address: A, Phone: P, Email: E, Open: O]
```

En què `I` és l'identificador de la sucursal, `A` és la seva adreça (i.e. carrer i número) més la ciutat i el codi postal, P és el número de telèfon, `E` és el correu electrònic i `O` és el valor que indica si la sucursal és oberta o no.

d) Té un constructor per defecte que inicialitza els atributs de la classe a zero (`=0`) si són numèrics, a `false` si són de tipus `boolean` i a `null` si són d'un altre tipus.

e) Té un constructor amb arguments que té tants arguments com atributs té la classe.

f) Inclou tots els mètodes *getter* i *setter* dels atributs de la classe. Tots són públics.

Dibuixa el diagrama de classes UML de la classe `Restaurant` indicant el nivell d'accés o visibilitat dels atributs i mètodes, com també els tipus dels atributs de la classe i el seu valor d'inicialització (si s'indica en l'enunciat), els tipus dels paràmetres dels mètodes i dels valors de retorn dels mètodes.

Exercici 3
Ens demanen que creem la classe `Room`, la qual modelarà la informació i el comportament de les sales que formen part del museu MuseUOC. La classe `Room` la definim de la manera següent:

a) Té un atribut que permet identificar la sala. Aquest identificador està format sempre per tres dígits, en què el primer fa referència a la planta on se situa la sala, per exemple, `0` per a la planta baixa, `1` per a la primera planta, etc. Els altres dos dígits fan referència al número de la sala en aquella planta. Així doncs, una sala amb identificador `102` significaria que és la sala número `2` de la primera planta. El museu no té plantes baixes (i.e. soterranis).

b) Té quatre atributs més que són els següents: nom de la sala (`name`), un *flag* que diu si la sala és oberta o no (`open`), els metres quadrats (`m2`) i l'aforament màxim (`capacity`).

c) Té un mètode públic anomenat `toString` que retorna el text següent (`String`):

```
[ID: I, Name: N, m2: M, Capacity: C, Open: O]
```

En què `I` és l'identificador de la sala, `N` és el seu nom, `M` són els metres quadrats, `C` és l'aforament de la sala i `O` és el valor que indica si la sala és oberta o no.

d) Té un constructor per defecte que inicialitza els atributs de la classe a zero (`=0`) si són numèrics, a `false` si són de tipus `boolean` i a `null` si són d'un altre tipus.

e) Té un constructor amb arguments que té tants arguments com atributs té la classe.

f) Inclou tots els mètodes *getter* i *setter* dels atributs de la classe. Tots són públics.

Dibuixa el diagrama de classes UML de la classe `Room` indicant el nivell d'accés o visibilitat dels atributs i mètodes, com també els tipus dels atributs de la classe i el seu valor d'inicialització (si s'indica en l'enunciat), els tipus dels paràmetres dels mètodes i dels valors de retorn dels mètodes.

Exercici 4
Per a modelar la informació i el comportament d'un usuari de la biblioteca BibliUOCteca, usarem la classe `Member`, la qual definim de la manera següent:

a) Té un atribut que permet identificar l'usuari. Aquest identificador està format sempre per deu dígits. El primer usuari és el `0000000000`.

b) Té quatre atributs més que són: correu electrònic (`email`), adreça (`address`), un *flag* que diu si l'usuari està bloquejat o no (`blocked`) i la data de naixement (`birthday`).

c) Té un mètode públic anomenat `toString` que retorna el text següent (`String`):

```
[ID: I, E-mail: E, Address, A: S, Birthday: D, Blocked: B]
```

En què `I` és l'identificador del membre, `E` és el seu correu electrònic, `A` és la seva adreça, `D` és la seva data de naixement i `B` és el valor que indica si l'usuari està bloquejat o no.

d) Té un constructor per defecte que inicialitza els atributs de la classe a zero (`=0`) si són numèrics, a `false` si són de tipus `boolean` i a `null` si són d'un altre tipus (per exemple, `Date`, etc.).

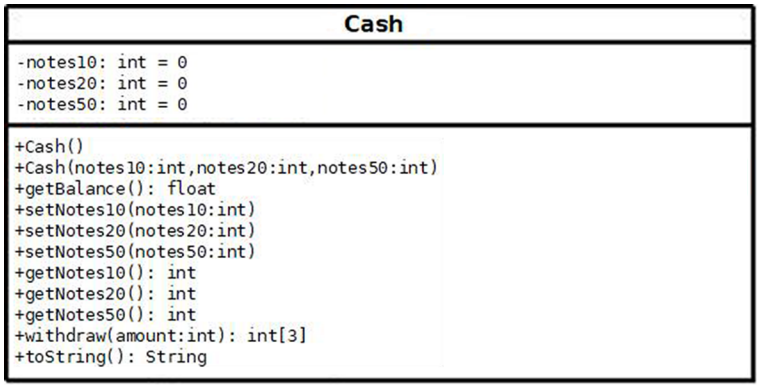
i) Té un constructor amb arguments que té tants arguments com atributs té la classe.

f) Inclou tots els mètodes *getter* i *setter* dels atributs de la classe. Tots són públics.

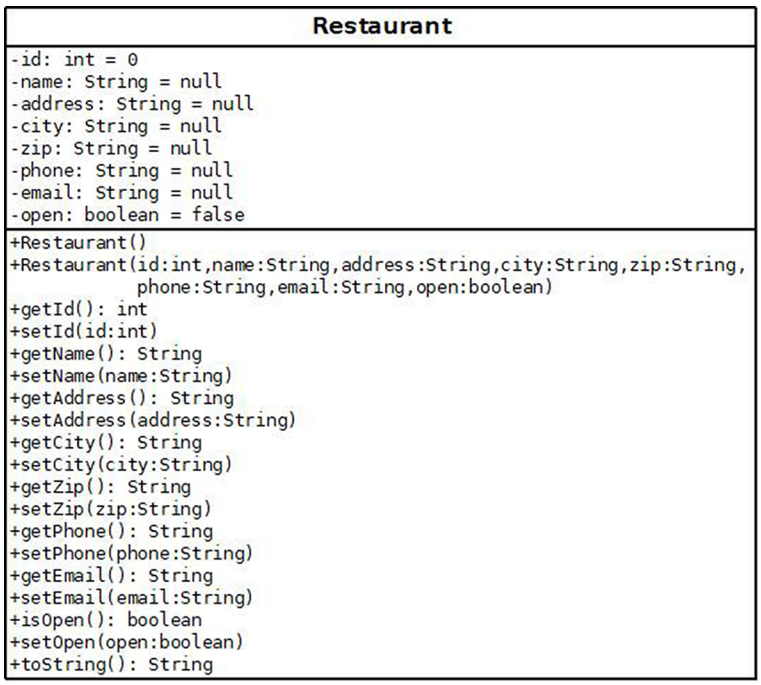
Dibuixa el diagrama de classes UML de la classe `Member` indicant el nivell d'accés o visibilitat dels atributs i mètodes, com també els tipus dels atributs de la classe i el seu valor d'inicialització (si s'indica a l'enunciat), els tipus dels paràmetres dels mètodes i dels valors de retorn dels mètodes.

Solucionari

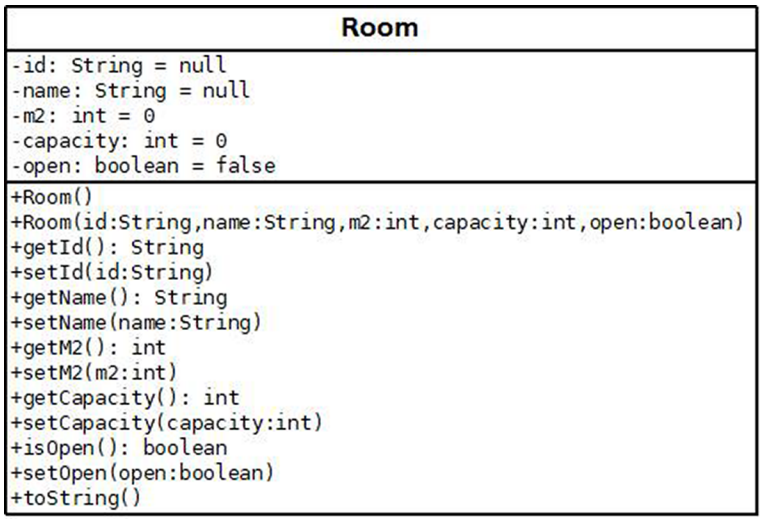
Solució exercici 1



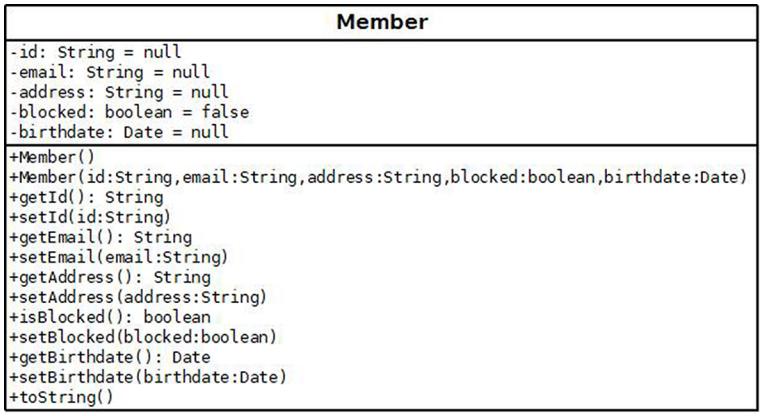
Solució exercici 2



Solució exercici 3



Solució exercici 4



Bibliografia

Booch, G.; Maksimchuk, R.; Engle, M.; Young, B. J.; Conallen, J.; Houston, K. (2007). *Object-Oriented Analysis and Design with Applications*. Addison-Wesley Professional. ISBN: 978-0201895513.

Griffiths, D.; Griffiths, D. (2019). *Head First Kotlin*. O'Reilly Media, Inc. ISBN: 978-1491996690.

Hunt, A.; Thomas, D. (2019). *The Pragmatic Programmer: your journey to mastery, 20th Anniversary Edition* (2a. ed.). Addison-Wesley Professional. ISBN: 978-0135956977.

Microsoft (s. f.). «C# Guide» [en línia]. Microsoft. <https://docs.microsoft.com/en-us/dotnet/csharp/>

Phillips, D. (2018). *Python 3 Object-Oriented Programming* (3a. ed.). Packt Publishing. ISBN: 978-1789615852.

Pollice, G.; West, D.; McLaughlin, B. (2006). *Head First Object-Oriented Analysis and Design*. O'Reilly Media, Inc. ISBN: 978-0596008673.

Robinson, S.; Nagel, C.; Watson, K.; Glynn, J.; Skinner, M.; Evjen, B. (2004). *Professional C#* (3a. ed.). Wrox. ISBN: 978-0764557590.

Sharp, J. (2007). *Microsoft Visual C# 2008 Step by Step*. Microsoft Press. ISBN: 978-0735624306.

Weisfeld, M. (2019). *The Object-Oriented Thought Process* (5a. ed.). Addison-Wesley Professional. ISBN: 978-0135182130.