Tipus de classe i interfície

David García Solórzano

PID_00235249

UOC
Universitat Oberta de Catalunya

Índex

Introducció

Aquest és el segon i últim mòdul que tracta sobre les relacions entre objectes i classes. En aquest mòdul hi aprofundirem una mica més en el paradigma de la programació orientada a objectes (POO).

Concretament veurem cinc tipus de classes que tenen un comportament especial que afecta la relació que aquestes classes especials tenen amb les classes «normals» vistes fins ara.

Una d'aquestes classes especials –la classe associativa– té conseqüències en el pla conceptual o de disseny, mentre que les altres quatre –abstracta, final, estàtica i parametritzada– ens afectaran no només en el comportament que tenen, sinó també directament en la sintaxi que cal utilitzar per a declarar-les.

Per acabar, presentarem la interfície, un element similar a una classe, però que té diferències importants que hem de conèixer.

Objectius

L'objectiu principal d'aquest mòdul és explicar com les classes (i, en conseqüència, els objectes) es relacionen entre si dins d'un programa basat en el paradigma de la programació orientada a objectes (POO). D'acord amb aquest objectiu més general, aquest mòdul es proposa acomplir els dos objectius següents:

1. Conèixer diferents tipus de classes especials que poden ser-nos útils en determinats dissenys.

2. Saber què és una interfície des del punt de vista del paradigma de la POO i la seva utilitat.

1. Tipus de classe

En aquest apartat veurem cinc tipus de classes «especials»:

- Classe associativa
- Classe abstracta
- Classe final o segellada
- Classe estàtica
- Classe genèrica o parametritzada

La definició de cadascuna d'elles té conseqüències en la manera en què es relacionen amb la resta de classe «normals». Dit d'una altra manera, aquestes classes tenen implicacions i aplicacions diferents en el disseny de programes.

El primer tipus de classe –la classe associativa– segueix una sintaxi idèntica a les classes «normals» que hem vist fins ara, però representa un canvi en el pla conceptual/teòric. En canvi, els altres quatre tipus de classe que explicarem obliguen el programador a utilitzar una paraula clau o una sintaxi especial en la definició per a distingir-les de les classes «normals».

1.1. Classe associativa

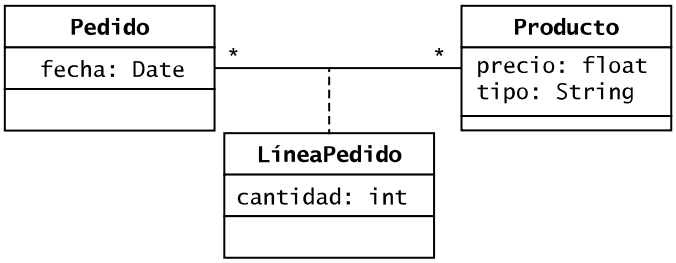
Com s'ha comentat, una classe associativa té sentit des d'un punt de vista teòric/conceptual o, millor dit, del disseny del nostre programari. No hi ha una sintaxi especial que ens permeti distingir a simple vista mitjançant el codi que una classe és associativa o no.

Les classes associatives apareixen en les associacions/relacions binàries en què ens cal guardar informació específica de la relació/associació (per això el seu nom).

Dit d'una altra manera, una classe associativa és aquella classe que està associada a una relació i ens permet guardar informació sobre aquesta relació.

Exemple 1

Imaginem que modelem el comportament d'una comanda d'una botiga en línia. El diagrama de classes fet amb UML podria ser el següent:



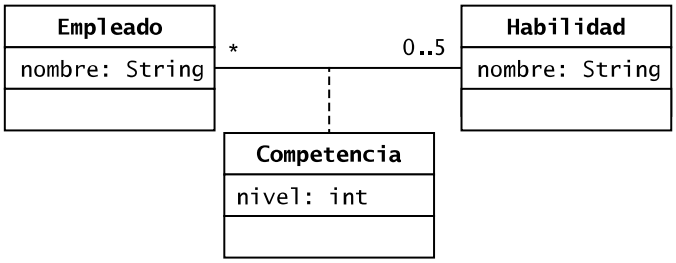
Amb la classe associativa *LineaPedido* guardem la quantitat d'un *Producto* concret per a un *Pedido* concret. És a dir, guardem informació que fa referència a la relació *Pedido-Producto*.

Com veiem en l'exemple anterior, en un diagrama de classes UML es fa servir una línia discontínua per a representar que una classe és associativa dins d'una relació. Així mateix, com es pot constatar, no hi ha cardinalitat ni navegabilitat per a aquesta classe.

Per acabar d'entendre el concepte de classe associativa, vegem-ne tres exemples més:

Exemple 2

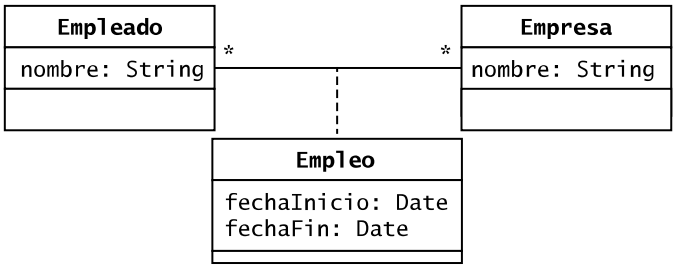
Modelem el nivell de competència que tenen els empleats respecte a una habilitat:



En aquest cas hem decidit que un empleat només pot tenir cinc habilitats. La cardinalitat podria haver estat diferent si ens haguessin proporcionat informació contextual.

Exemple 3

Modelem quant de temps passa un empleat en les empreses on treballa:

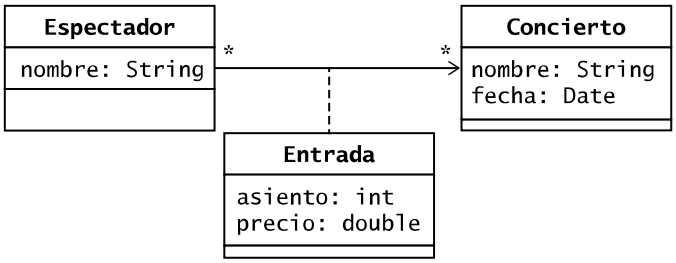


Exemple 4

Modelem que un espectador té la seva entrada per a un concert concret:

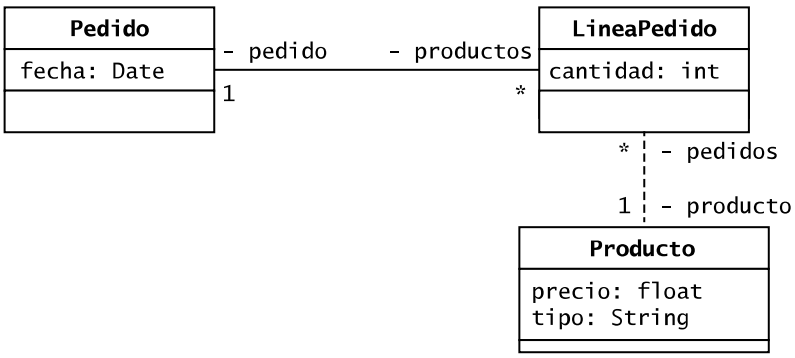
Vegeu també

Podeu veure el codi de l'exemple de les entrades al concert a l'exemple 501 de la col·lecció d'exemples de l'assignatura.



Un espectador assisteix al concert i fruit d'aquesta relació apareix la seva entrada que actua com a classe associativa. Hem posat navegabilitat unidireccional perquè és l'espectador qui sap del concert i de l'entrada; el concert no sap res de l'espectador.

Pel que fa a com es codifica una classe associativa, cal dir que els llenguatges de programació no proporcionen un mecanisme directe per a implementar-la. Per aquest motiu, l'habitual és implementar-la com una classe normal que posseeix dues associacions binàries amb les classes de la relació, tal com es mostra en la figura següent.

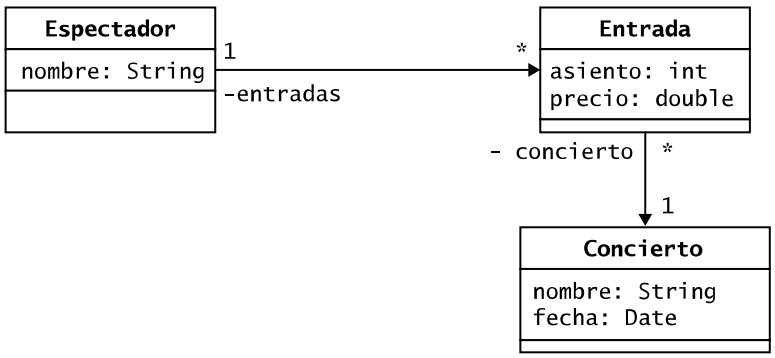


Fixeu-vos que cada objecte de la classe associativa (és a dir, *LineaPedido*) es relaciona amb només un objecte de cadascuna de les classes de la relació (o sigui, *Pedido* i *Producto*). És a dir, només hi ha un objecte del tipus de la classe associativa *LineaPedido* per relació *Pedido-Producto*.

Així doncs, ha de quedar clar que només hi pot haver un objecte de la classe associativa per relació.

Així mateix, és important entendre que si la relació entre dos objectes de l'associació binària original es destrueix (en aquest cas, *Pedido-Producto*), per exemple perquè eliminem un dels dos objectes, llavors cal eliminar l'objecte de la classe associativa (en aquest cas, *LineaPedido*), ja que només té sentit que hi hagi l'objecte de la classe associativa si hi ha la relació.

Per acabar, cal comentar que hem de tenir present la navegabilitat a l'hora de codificar. Així doncs, en l'exemple de les entrades al concert en el qual la navegabilitat és unidireccional, el diagrama «equivalent» seria el següent:



1.2. Classe abstracta

La classe abstracta té molt a veure amb l'herència.

> Una classe abstracta es defineix com aquella que declara l'existència de tots els seus mètodes, però no els implementa tots.

Una de les utilitats de definir una classe abstracta és actuar com a superclasse (o classe pare) per a unificar atributs i mètodes de les subclasses, evitant la repetició de codi i unificant processos.

Per a definir una classe abstracta, farem servir la paraula reservada *abstract*:

```
public abstract class Persona{…}
```

> Arribats a aquest punt, és important saber que una classe abstracta no es pot instanciar però sí que es pot heretar. És a dir, no podrem fer servir la paraula *new* per a crear objectes del tipus de la classe abstracta.

Si sabem que una classe abstracta no es pot instanciar, té sentit que tingui definit un o diversos constructors? Doncs sí, ja que si una classe hereta de la classe abstracta, el seu constructor, el primer que ha de fer és sol·licitar el constructor de la classe pare (abstracta) mitjançant la funció especial *super*.

Una classe abstracta sol contenir, com a mínim, un mètode abstracte. No obstant això, no estem obligats que la nostra classe abstracta contingui mètodes abstractes.

Un mètode és abstracte si:

- Té la paraula *abstract* al davant.

- Només consta de la signatura del mètode. És a dir, no té claus i, per tant, no implementa el codi del mètode.

És important saber que un mètode abstracte només pot existir dins d'una classe abstracta. És a dir, si declarem que un mètode és abstracte, estem obligats que la classe que el conté sigui abstracta.

Exemple

```
public abstract class Persona{
   protected String nombre;
   public Persona(){}
   public abstract String getNombre();
   public void setNombre(String n){nombre = n;};
}
```

En l'extracte de codi anterior veiem que el mètode *getNombre* és abstracte i no té implementat el seu codi. Serà una classe filla/derivada/subclasse qui ho faci.

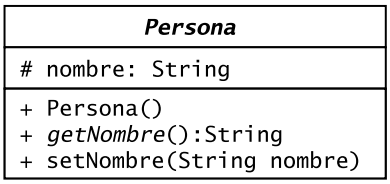
Si una classe filla no implementa el codi d'aquest mètode abstracte, la classe filla haurà de ser una classe abstracta.

Per la seva banda, la classe *Persona*, en contenir un mètode abstracte, ha de ser declarada com a abstracta.

Vegeu també

Aquest exemple el teniu codificat en l'exemple 502 de la col·lecció d'exemples de l'assignatura.

En UML representem una classe abstracta escrivint el nom de la classe en cursiva i els mètodes abstractes també en cursiva.



En el cas representat, el mètode *setNombre* s'ha implementat en la classe *Persona* (encara que el pot sobreescriure una subclasse), mentre que el mètode *getNombre* ha estat declarat com a abstracte (està en cursiva) i, per això, ha de ser implementat en una subclasse que hereti directament o transitivament de *Persona*.

1.3. Classe final o segellada

En Java es denomina classe final, mentre que a C#, per exemple, es diu classe segellada (en anglès, *sealed*). De fet, hem de fer servir la paraula reservada *final* (en Java) o *sealed* (en C#) davant de la classe per a indicar que volem que es comporti com a tal. Així doncs, per a definir una classe final en Java farem:

```
public final class Cliente{…}
```

Una classe final/segellada pot ser instanciada, però no heretada.

Vegeu també

Teniu un exemple de classe final en l'exemple 503 de la col·lecció d'exemples de l'assignatura.

Una classe final/segellada és just el cas contrari de la classe abstracta. Pot ser **útil per a crear classes immutables**. Per exemple, si tenim una classe que defineix tot el comportament dels nombres complexos, com que no es poden fer més ampliacions del camp numèric, no té sentit permetre l'herència.

En Java, a més d'una classe podem definir com a *final* un mètode o un atribut:

- **Mètode**: si el mètode d'una classe (sigui aquesta *final* o no) és *final*, llavors aquest mètode no pot ser sobreescrit per una classe filla. Això és útil per a assegurar-se que una subclasse no modifiqui el codi d'un mètode crucial.

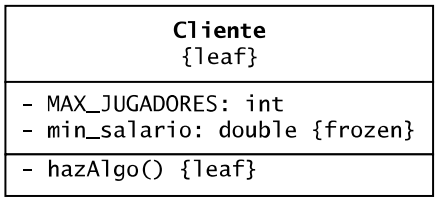
- **Atribut**: un atribut final fa que aquest atribut només pugui ser inicialitzat una vegada. Aquesta inicialització només es pot fer a dos llocs: (1) en el constructor de la classe o (2) en el moment de la declaració dins de la classe. Si optem per la via del constructor, el compilador ens obligarà a inicialitzar l'atribut final en tots els constructors de la classe. Així doncs, un atribut final actua com una constant un cop se li ha assignat un valor.

En un diagrama de classes UML:

- Una classe final s'indica escrivint la propietat *{leaf}* sota el nom de la classe. D'aquesta manera, diem que la classe és la fulla d'un arbre o, dit d'una altra manera, l'última classe d'una jerarquia. Per tant, no es pot heretar d'ella. Que no es pugui heretar la classe és el mateix que dir que és *final*.

- Per a indicar que un mètode és *final*, podem posar la propietat *{leaf}* al costat dret del mètode. Aquesta pràctica no és estàndard, però tenim una certa flexibilitat a l'hora de dibuixar els nostres diagrames de classes.

- Sol ser una pràctica molt estesa distingir un atribut final de la resta d'atributs escrivint segons la convenció de nomenclatura de les constants dels llenguatges de programació, és a dir, tot en majúscules i el símbol "_" (*underscore*) per a representar l'espai que separa paraules. Altres especificacions fan servir la paraula *{readOnly}* o *{frozen}* o *{const}* al costat dret de l'atribut. Totes dues maneres es poden utilitzar simultàniament si es vol.



1.4. Classe estàtica

Ja coneixem el concepte d'extensió –o més conegut per la seva paraula clau, *static*– per als mètodes i atributs d'una classe. Doncs bé, una classe també pot ser estàtica.

Vegeu també

En el mòdul «Encapsulació i extensió» es tracta el concepte d'*extensió*.

Perquè una classe sigui estàtica, tots els membres (és a dir, atributs i mètodes) de la classe han de ser estàtics.

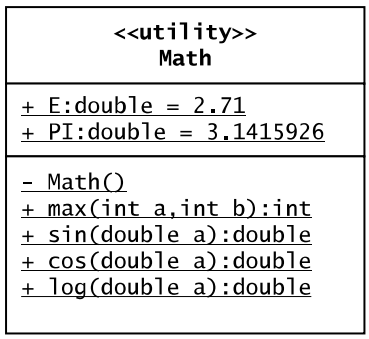
Això implica que una classe estàtica hauria d'actuar com una classe abstracta en el sentit que una classe estàtica no hauria de poder ser instanciada. És a dir, no hauríem de poder fer servir la paraula *new* per a crear objectes del tipus de la classe estàtica.

Pel que fa referència a la seva implementació en Java, no cal especificar res especial sobre la classe, només n'hem de declarar els membres com a estàtics per mitjà de la paraula reservada *static*. Així mateix, **sol ser una pràctica habitual declarar el constructor per defecte com a *private*** per a assegurar-se que aquesta classe no pot ser instanciada. Si no ho féssim, la classe podria ser instanciada, la qual cosa aniria en contra de la definició de classe estàtica.

Classe estàtica en altres llenguatges

Queda a les vostres mans buscar com s'implementa una classe estàtica en altres llenguatges.

D'altra banda, per a representar una classe estàtica en un diagrama de classes UML anteposarem l'***estereotip*** **<<*utility*>>**, encara que no seria necessari, ja que podem suposar que la classe és estàtica perquè tots els seus membres són estàtics.



La classe *Math* de Java és el típic exemple de classe estàtica (o *utility*). Proporciona atributs i mètodes que ens poden ser útils per a moltes operacions matemàtiques, com per exemple, saber quin és el major/menor de dos nombres, calcular el sinus d'un angle, etc. No té sentit instanciar aquesta classe, ja que no deixa de ser un conjunt d'eines (és a dir, *toolbox*) matemàtiques que fem servir sense més i no té una correspondència amb un objecte/entitat del món real.

Enllaços d'interès

Podeu veure la documentació de la classe *Math* de Java en l'enllaç següent: https://docs.oracle.com/javase/7/docs/api/java/lang/Math.html.

Una altra classe estàtica en Java és la classe *Collections*, la qual té mètodes utilitzats freqüentment sobre col·leccions, com ara l'ordenació (*sort*) dels elements que conté una col·lecció.

Enllaços d'interès

I sobre la classe Collections de Java a: http://docs.oracle.com/javase/1.5.0/docs/api/java/util/Collections.html.

Com totes dues classes no es poden instanciar, per a accedir als seus membres ho farem anteposant el nom de la classe en el missatge, per exemple:

```
Math.max(5,8);
```

Per acabar, cal indicar que, al contrari del que hem dit abans, en realitat sí que hi ha la possibilitat de crear classes estàtiques en Java per mitjà de la paraula *static* en la declaració de la classe:

```
public class ClaseNormal {
      //TODO
    public static class ClaseEstatica {
      //TODO
    }
}
```

Vegeu també

Podeu veure la implementació d'una classe estàtica (no imbricada) en l'exemple 504 de la col·lecció d'exemples de l'assignatura.

però, tal com es veu en el fragment de codi anterior, aquesta classe estàtica ha d'estar nidificada dins d'una altra classe. Si es declara una classe usant la paraula *static*, llavors no cal declarar constructors per a aquesta classe i tots els seus membres han de ser estàtics.

Classe nidificada

El concepte de classe nidificada (és a dir, *nested class*) queda fora de l'abast d'aquesta assignatura.

1.5. Classe genèrica o parametritzada

Sovint tenim classes que funcionen com una col·lecció (o contenidor) d'objectes. Dos exemples que coneixeu són la pila (*stack*, *LIFO*) i la cua (*queue*, *FIFO*). En un escenari podem tenir una pila (o cua) de persones (és a dir, objectes de tipus/classe *Persona*), però en un altre escenari podria contenir televisors (és a dir, objectes de tipus/classe *Televisor*). En tots dos contextos el funcionament de la pila (o cua) seria el mateix, l'únic que canviaria és el tipus/classe d'objecte que emmagatzema.

Davant aquesta situació, molts programadors implementen una classe pila (o cua) que només accepta objectes *Persona* anomenada, per exemple, *PilaPersona*, i fan una altra classe exacta (copiant directament el codi) en la qual es

reemplaça *Persona* per *Televisor* anomenada, per exemple, *PilaTelevisor*. Com es pot intuir, això no és pràctic ni escalable. No seria millor tenir una única classe genèrica? La resposta és clarament *sí*.

Alguns programadors implementen la classe, en aquest cas pila o cua, com una classe que guarda objectes de la classe/tipus *Object*. D'aquesta manera, quan recuperem un element de la pila o cua mitjançant un mètode de tipus *getter*, només hem d'aplicar a l'element retornat de tipus *Object* un càsting del tipus de la classe que sabem que hem desat. Aquest disseny és una millora respecte a tenir tantes classes pila (o cua) com tipus d'objectes volem desar. Aquesta manera d'implementar una classe genèrica la vam veure a l'apartat 2 del mòdul «Encapsulació i extensió». No obstant això, no és la pràctica més adequada, ja que el procés de *càsting* penalitza el rendiment del programa.

Vegeu també

El procés de *càsting* s'estudia en el mòdul «Associació i herència».

És per això que va aparèixer un tipus de classe que permet al programador indicar en el moment de la instanciació, i segons el context en què estem, el tipus d'objecte que ha d'emmagatzemar aquesta classe. Aquest tipus de classe és conegut com a **classe genèrica** o **parametritzada**. Per a implementar classes genèriques, els llenguatges de programació orientats a objectes han creat diferents mecanismes. Per exemple, en C++ aquest mecanisme es coneix com a *templates* i en Java, com a *generics*.

Gràcies a les classes genèriques o parametritzades, podem programar abstraient-nos dels tipus de dades. Per a fer una classe genèrica o parametritzable en Java, posarem un o diversos àlies de les classes que podran ser parametritzables entre els símbols <>.

Exemple

```
public class Impresora<T>{
   private T objeto; //el tipus/classe T es decidirà
   //quan s'instanciï cada objecte de tipus Impresora

   public Impresora(T objetoNuevo){
      objeto = objetoNuevo;
   }
   public void setObjeto(T objetoNuevo){
      objeto = objetoNuevo;
   }
   public T getObjeto (){
      return objeto;
   }
   public String getTexto (){
      //el mètode "toString()" s'hereta de la classe Object
      return objeto.toString();
   }
}
```

En el programa principal tindríem:

```
Impresora<Integer> impresora1 = new Impresora<Integer>();
Impresora<Persona> impresora2 = new Impresora<Persona>();
Impresora<Televisor> impresora3 = new Impresora<Televisor>();
```

En el primer cas, *impresora1*, emmagatzemaríem un objecte de tipus/classe *Integer*, en el segon, de tipus *Persona* i en el tercer, de tipus *Televisor*. En fer servir una classe parametritzable només hem implementat una classe, no tres. Per tant, estalviem codi en el nostre programa, i millorem el manteniment i l'escalabilitat, a més del rendiment en evitar fer un procés de *càsting*.

Vegeu també

Aquest exemple el teniu codificat en l'exemple 505 de la col·lecció d'exemples de l'assignatura.

La classe *Impresora* de l'exemple anterior podria haver tingut més d'una classe parametritzable:

```
public class Impresora<T,S,X>{...}
```

També podem limitar els tipus amb els quals es pot parametritzar la nostra classe. Per exemple, pensem en una classe que suma dos nombres; no tindria sentit poder parametritzar la nostra classe amb el tipus *String* (que és text). Per a això farem servir la paraula reservada ***extends***:

```
public class Suma<T1 extends Number, T2 extends Number>{...}
```

Així doncs, des del programa principal podríem fer:

```
Suma suma1 = new Suma<Integer,Double>(2,5.5);
```

però no:

```
Suma suma1 = new Suma<Float,String>(3.5,"8");
```

També podem fer que un tipus sigui d'una classe i una o diverses interfícies. Per a això farem servir &:

```
public class A<T1 extends Number & Interfaz1 & Interfaz2>{...}
```

En aquest punt és important saber que en Java (consulteu altres llenguatges):

Vegeu també

Veurem el concepte *interfície* més endavant en aquest mòdul.

1) Els tipus parametritzables no poden ser tipus primitius (és a dir, *int*, *float*, *double*, etc.), sempre han de ser classes. No podem fer:

```
Suma suma1 = new Suma<int,double>(2,5.5);
```

2) No es permeten declarar *arrays* de *generics*. No podem fer:

```
ArrayList<String> strLista[] = new ArrayList<String>[3]
```

El més semblant a crear un *array* genèric que podem fer és:

```
Object [] arr = new String[3];
```

3) Tampoc no es pot sobrecarregar un mètode el paràmetre formal del qual sigui un *generics*:

```
public void imprimirLista(ArrayList<Persona> listaPersonas) {...}

public void imprimirLista(ArrayList <Televisor> listaTelevisores) {...}
```

La classe *ArrayList* de Java és genèrica. Per això, per al compilador de Java, els dos mètodes anteriors són el mateix, és a dir, tenen el mateix nom, així com el mateix nombre i tipus de paràmetres. És a dir, hi ha un error en fer la sobrecàrrega del mètode i no podrem compilar el nostre programa. De fet, per al compilador de Java, els dos mètodes anteriors són en realitat aquest:

```
public void imprimirLista(ArrayList<T> lista) {...}
```

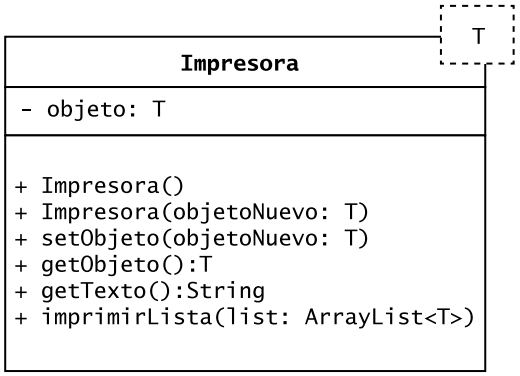
i és aquesta signatura la que haurem de fer servir.

4) Hi ha la possibilitat d'utilitzar el comodí *?* quan fem servir una classe genèrica i no sabem el tipus que hi emprarem, per exemple en la signatura d'un mètode o en una declaració:

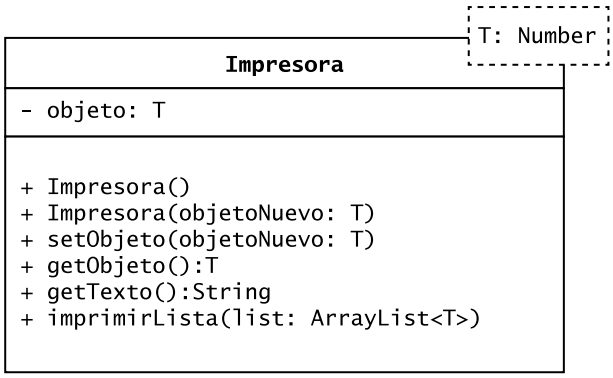
```
public void metodoDeUnaClaseDistintaAImpresora(Impresora<?> imp){...}
public Impresora<? extends File> fichero;
```

És millor evitar l'ús del comodí *?*, però també és important saber que existeix.

En UML, les classes genèriques o parametritzades –conegudes en l'àmbit de l'UML també com a classes plantilla (*template*)– s'identifiquen de la manera següent:



Si es vol indicar la informació d'acotació que determina *extends* i el símbol *ampersand* (&) en la declaració de la classe genèrica, o classes genèriques, farem servir els dos punts seguits de la informació:



En el cas anterior, la classe *impresora* només accepta objectes que siguin de la classe *Number* o derivades de *Number*.

Per acabar, cal comentar que els llenguatges orientats a objectes proporcionen al programador moltes classes que són parametritzables. En el cas de Java, tenim *ArrayList*, *LinkedList*, etc. Com veiem, la majoria són classes el comportament de les quals és implementar una col·lecció (o contenidor).

Enllaç d'interès

Com a curiositat, cal dir que *ArrayList*, *LinkedList*, etc. són classes que implementen la interfície *List* que també està parametritzada. Podeu veure l'enllaç https://docs.oracle.com/javase/7/docs/api/java/util/List.html.

Vegeu també

En l'apartat següent veurem què és una interfície.

2. Interfície

2.1. Concepte de interfície

En aquest apartat veurem la **interfície**, un element de la POO que no és una classe, però que s'hi assembla bastant. De fet, en algun lloc llegireu que és com una classe 100% abstracta, és a dir, no implementa el codi de cap dels mètodes que declara, sinó que només en defineix/declara les signatures. De la mateixa manera, **tant la interfície com la classe abstracta no poden ser instanciades**.

De fet, es diu que una interfície és *com* una classe 100% abstracta perquè no és exactament el mateix, òbviament. Algunes diferències significatives entre una interfície i una classe abstracta són:

- Una interfície no pot tenir atributs llevat que siguin estàtics (és a dir, *static*) o constants (és a dir, *final*), mentre que una classe abstracta sí que pot tenir qualsevol tipus d'atribut. A més, en una **interfície, els atributs han de ser públics i inicialitzats amb un valor en el moment de la declaració**.

- Una interfície no és heretada per una classe, sinó que una classe *implementa* una interfície. Per la seva banda, les classes abstractes s'hereten.

- En llenguatges com Java o C#, una classe pot implementar múltiples interfícies, però només pot heretar d'una classe base (o superclasse o classe pare). Això vol dir que **en Java o C#, l'herència múltiple se «simula» mitjançant la implementació de múltiples interfícies**.

> Així doncs, podem dir que una interfície defineix l'esquelet d'una classe delegant-ne la implementació en una nova classe que implementa aquesta interfície.

Diem que una interfície és un esquelet d'una classe perquè aquesta classe només declara les signatures dels mètodes, no conté res de codi.

2.2. Casos d'ús

La utilitat de les interfícies es fa patent quan es treballa en equips de desenvolupament on molts programadors desenvolupen una aplicació a la vegada. És molt útil que l'encarregat del disseny de l'aplicació proporcioni, a més del diagrama de classes, interfícies a l'equip de desenvolupament perquè enten-

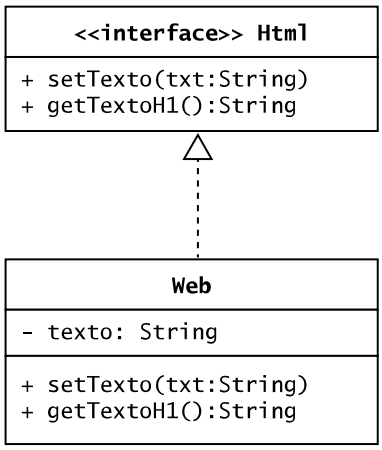
guin com s'ha de comportar un determinat objecte (és a dir, classe) i evitar així que el programador s'oblidi d'implementar els diferents mètodes (és a dir, comportaments) definits pel dissenyador. De fet, si una classe implementa una interfície i aquesta no sobreescriu un dels mètodes declarats en la interfície –encara que només sigui obrir i tancar claus–, el compilador dóna un error.

En un diagrama de classes UML també es poden definir les interfícies. Aquestes es representen mitjançant una caixa amb dues parts:

1) nom de la interfície amb l'***estereotip*** ***<<interface>>*** anteposat i

2) la signatura dels mètodes.

És com una classe, però com que en general no té atributs, eliminem la part destinada a la declaració dels atributs. A més, el símbol d'*implementar* és com el d'*heretar*, però la línia és discontínua.



Interfície en el diagrama de classes

Una interfície rarament té atributs. Si els tingués, la seva representació en el diagrama de classes seria idèntica a la d'una classe però amb l'estereotip <<interface>>.

Representació dels mètodes d'una interfície en el diagrama de classes

En molts diagrames de classes veureu els mètodes de les interfícies en cursiva, com si fossin abstractes.

Representació dels mètodes implementats d'una interfície en el diagrama de classes

Fixeu-vos que a la classe filla *Web* es representen els dos mètodes de la interfície, ja que els hem hagut de sobreescriure obligatòriament en implementar la interfície *Html*, ja que en la interfície, per definició, els mètodes no estan codificats.

2.3. Traduint a codi...

Una interfície en Java (i en C#) es defineix de la manera següent:

```
public interface Html {
   public void setTexto(String txt);
   public String getTextoH1();
}
```

Vegeu també

Aquest exemple el teniu codificat en l'exemple 506 de la col·lecció d'exemples de l'assignatura.

Els llenguatges que accepten interfícies tenen diferents sintaxis per a indicar que una classe implementa una interfície.

En Java:

```
public class Web implements Html{
   String texto;
   public void setTexto(String txt){
      texto = txt;
   }
   public String getTextoH1(){
      return "<h1>"+texto+"</h1>";
   }

}
```

En C#:

```
public class Web : Html{
//Aquí aniria el codi dels mètodes de la classe Web, la qual cosa implica implementar els mètodes
//definits en la interfície HTML
}
```

2.4. Herència i polimorfisme

Cal tenir en compte que una interfície pot heretar d'una altra interfície. En Java i C#, a causa de la restricció de l'herència simple, només podrà heretar d'una interfície.

És a dir:

```
public interface Interfaz2 extends Interfaz1{
   //Aquí van les signatures dels mètodes
}
```

Una interfície no pot implementar una altra interfície

Una interfície A no pot implementar una interfície B, perquè la interfície A, per a ser interfície, només pot indicar les signatures dels mètodes i, per tant, no pot codificar els mètodes de la interfície B.

A més, cal tenir present que la declaració d'una interfície comporta la creació d'un nou tipus que podem utilitzar per a declarar objectes.

D'aquesta manera, si es defineix un atribut/variable el tipus del qual és una interfície, se li pot assignar un objecte que sigui una instància d'una classe que implementa la interfície.

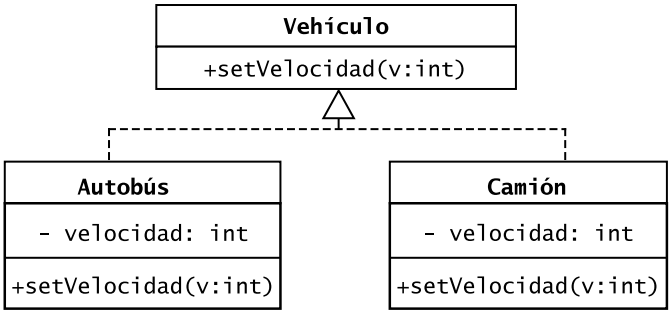
Vegeu també

Aquest exemple el teniu codificat en l'exemple 507 de la col·lecció d'exemples de l'assignatura.

Això vol dir que utilitzant interfícies com tipus, es pot aplicar el mecanisme de polimorfisme a classes que no estan relacionades pel mecanisme d'herència, però sí pel mecanisme d'implementació d'una interfície.

Exemple

Si tenim el programa següent:



Podem fer:

```
Vehiculo v1 = new Autobus();
Vehiculo v2 = new Camion();
v1.setVelocidad(10); //executa el mètode setVelocidad de Autobus
v2.setVelocidad(10); //executa el mètode setVelocidad de Camion
Vehiculo [] vehiculos = new Vehiculo[3]; //A la declaració de la array no s'usa el
//constructor de la classe/interfície, per tant, funciona.
vehiculos[0] = new Autobus();
vehiculos[1] = new Camion();
vehiculos[2] = v1;
//La següent línia de codi ens donarà error perquè no podem instanciar una interfície
Vehiculo v = new Vehiculo();
```

Resum

En aquest mòdul hem ampliat els coneixements sobre com es relacionen les classes i objectes entre si.

En l'apartat 1 hem vist nous tipus de classe. El primer d'ells, la **classe associativa**, és una classe nova en el pla conceptual que serveix per a guardar informació sobre la relació d'associació binària (subapartat 1.1).

A continuació, hem parlat de quatre classes especials –la **classe abstracta** (subapartat 1.2), **final** (subapartat 1.3), **estàtica** (subapartat 1.4) i **parametritzada** (subapartat 1.5)– que impliquen, a més de comportaments nous, canvis en la sintaxi a l'hora de declarar-les. És convenient conèixer-les ja que ens permeten dissenyar millor els nostres programes.

Per acabar, en l'apartat 2 s'ha presentat el concepte d'**interfície** i la seva utilitat a l'hora de jerarquitzar les classes del nostre programa i controlar, al mateix temps, que un equip de desenvolupadors acompleixi el disseny realitzat per a l'aplicació.

Bibliografia

Booch, G.; Jacobson, I.; Rumbaugh, J. (1999). *El lenguaje de modelado unificado UML*. Madrid: Addison-Wesley Iberoamericana.

Casas, J.; Conesa, J. (2013). *Diseño conceptual de bases de datos en UML*. Barcelona: Editorial UOC.

Meyer, B. (1997). *Object-oriented software construction*. Santa Bárbara: Prentice Hall.

Gamma, E.; Helm, R.; Johnson, R.;Vlissides, J. (1995). *Design patterns. Elements of reusable object-oriented design*. Addison Wesley.

Rumbaugh, J.; Blaha, M.; Premernali, W.; Eddy, F.; Lorensen, W. (1996). *Modelado y diseño orientado a objetos*. Madrid: Prentice Hall.

Valeri, M.; Naccarato, G. (2006). *Java 5 – Novedades del lenguaje*. Lulu.com. ISBN: 9781430301288.