TEMA 2. RELACIONES ENTRE CLASES. HERENCIA ENTRE CLASES.

Introducción al tema:

En el Tema 1 nos hemos ocupado de cómo se debe implementar una clase en un lenguaje orientado a objetos y las mejoras que esto suponía con respecto a la implementación de TAD's en lenguajes estructurados. En este tema nos vamos a ocupar de hacer que esas mismas clases se comuniquen entre sí, por medio de lo que llamaremos "relaciones entre clases".

En primer lugar (Sección 2.1) veremos cómo distintas clases se pueden comunicar entre sí y colaborar por medio del paso de mensajes. Para ello deberemos recuperar las nociones de parte pública y parte privada de una clase. En la Sección 2.2 pasaremos a enunciar dos de los tipos de relaciones entre clases más conocidos (asociación y agregación) y a ver cómo se pueden representar los mismos en los lenguajes de programación. Esto nos servirá para ver cómo los lenguajes de especificación (por ejemplo, UML) tienen una sintaxis más rica que la mayor parte de los lenguajes de POO, que ofrecen un número limitado de mecanismos para implementar relaciones entre clases.

Después de algunos ejemplos de relaciones entre clases, pasaremos a ver la parte central del Tema 2, que son las relaciones de especialización/generalización (también conocidas como herencia). A ello dedicaremos el resto del Tema 2. En particular, la Sección 2.3 la emplearemos en enunciar ejemplos de relaciones de especialización/generalización basados en los ejemplos hasta ahora introducidos en la asignatura. Nuestro propósito será observar que una relación entre clases que permita agrupar clases con una serie de características (atributos y métodos) comunes nos va a permitir reutilizar el código de nuestros programas (y simplificar el mantenimiento del mismo). A partir de estos ejemplos, en la Sección 2.4 enunciaremos formalmente la noción de herencia entre clases. En la Sección 2.5 recuperaremos algunas de las ventajas principales de utilizar relaciones de herencia entre clases que ya habrán ido apareciendo a lo largo del Tema y las ilustraremos con ejemplos. La Sección 2.6 la dedicaremos a introducir aquellas relaciones de herencia que no heredan íntegramente el comportamiento de sus superclases, sino que lo modifican por medio de la redefinición de algunos de sus métodos. En la Sección 2.7 recuperaremos los modificadores de acceso que introdujimos en la Sección 1.8 del Tema 1 y añadiremos el modificador de acceso "protegido" (o protected) explicando su comportamiento en relaciones de herencia.

La Sección 2.8 la dedicaremos a conocer la representación de relaciones entre clases en diagramas UML (tanto las de asociación como las de agregación, y

por supuesto, las de generalización/especialización). Finalmente, en la Sección 2.9 veremos la sintaxis en Java y en C++ que permite definir relaciones de herencia; aquí prestaremos especial atención a la definición de los constructores de subclases así como a la declaración y definición de objetos de las mismas. En realidad, a lo largo del Tema 2 habremos ido introduciendo las notaciones propias de UML, Java y C++ para definir relaciones entre clases, así que las Secciones 2.8 y 2.9 servirán como compendio de la notación que ya habremos introducido.

Los objetivos principales del Tema son que los alumnos conozcan los diversos tipos de relaciones entre clases existentes, que sean capaces de identificar por sí mismos relaciones de generalización/especialización, que utilicen los mecanismos de representación, declaración y definición de relaciones de herencia entre clases en UML, Java y C++, y que conozcan y apliquen las nociones de redefinición de métodos y de atributos “protegidos” en ejemplos de programación.

Entre los objetivos del Tema no se encuentra la identificación de relaciones de asociación y de agregación entre clases que los alumnos aprenderán en asignaturas más dirigidas al diseño de sistemas software.

Una posible referencia sobre relaciones entre clases y su implementación en Java son los capítulos 8, 16 y 17 de “Practical Object-Oriented development with UML and Java”, Richard C. Lee, William Tepfenhart, en Prentice Hall (2002).

2.1 COMUNICACIÓN ENTRE DISTINTAS CLASES

En el Tema 1 pusimos especial énfasis en comprender cómo los usuarios y clientes de una clase se comunicaban con la misma. Para posibilitar esta comunicación de una clase con el “exterior” (en el Tema 1 entendíamos por “exterior” el único usuario o cliente que definíamos, el programa principal) utilizábamos la parte pública (o interfaz) de la misma (tal y como introdujimos en la Sección 1.8).

En el Tema 2 aumentaremos el ámbito de lo que entendemos como usuarios o clientes de una clase también a otras clases. Veremos (en la Sección 2.2 y 2.3) que dos (o más clases) pueden tener una relación entre ellas de asociación, agregación o de especialización/generalización.

En el primer caso y en el segundo, la comunicación entre clases será posible sólo a través de la parte pública de las clases correspondientes (o, en Java, a las partes declaradas como “package”). Conviene recordar ahora que el ámbito de visibilidad de un método o atributo público es cualquier otra clase o fragmento de código que haya en nuestra aplicación.

En las relaciones de herencia veremos que además existe un modificador de acceso especial, llamado “protected” que permite que un atributo o método de una clase sea visible desde las clases que heredan de la misma.

En cualquier caso, nunca una clase podrá acceder a un atributo que sea privado en otra, independientemente de cómo éstas estén relacionadas.

Manual de buenas prácticas: incidiendo en la idea de ocultación de información que presentamos en la Sección 1.9, el programador de una clase debe, por lo general, declarar los atributos (o estado de la misma) como privados (private) y hacer que sean métodos los que accedan a esa información. De esta forma facilitará que si se modifica la implementación de la misma, los usuarios (ya sea el programa principal, u otras clases) no perciban ni sufran las consecuencias de estos cambios.

2.2 CLASES QUE CONTIENEN OBJETOS COMO ATRIBUTOS: ALGUNOS EJEMPLOS CONOCIDOS

En esta Sección presentaremos dos tipos de relaciones entre clases. En primer lugar hablaremos de relaciones de agregación entre clases (Sección 2.2.1), y luego pasaremos a hablar de relaciones de asociación (Sección 2.2.2).

Una vez más, insistimos en la idea de que el objetivo de este curso no es tanto que el alumno identifique tipos de relaciones entre clases, sino que comprenda los mecanismos de comunicación entre la mismas. Por eso esta Sección tendrá un contenido basado en ejemplos que permitan ver estos tipos de relaciones en ejemplos reales.

2.2.1 RELACIONES DE AGREGACIÓN

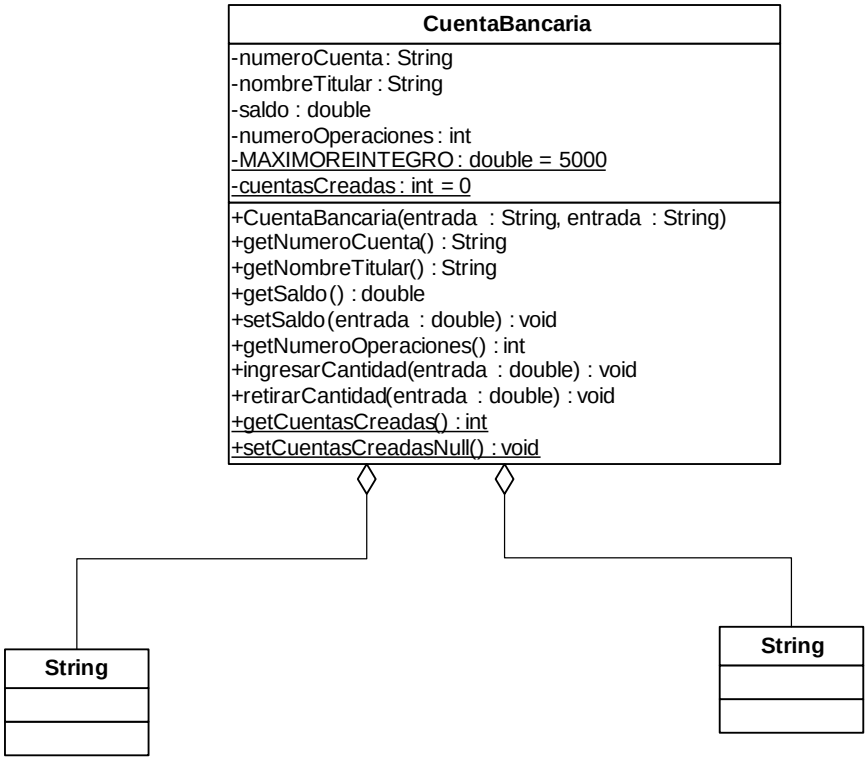
Las relaciones de agregación se basan en la idea de observar o entender un objeto como una composición de otros objetos. Desde nuestro punto de vista, las relaciones de agregación se entenderán como relaciones en las cuales una serie de clases aparecen como tipos de los atributos de otra clase.

Estas relaciones se conocen también como relaciones “todo - partes”. El “todo” está representado por la clase que aglutina a las otras clases, y las “partes” están dadas por las diversas clases que aparecen.

La mejor forma de identificar si nos encontramos ante una relación de agregación es preguntarnos si la clase que queremos definir “tiene un” (en inglés, “has - a”) atributo de la otra clase que estemos usando (de ahí que en ciertas referencias se definan como relaciones “has - a”).

En realidad, hemos tratado ciertos ejemplos de agregación (en Java) desde que comenzamos el curso, ya que hemos usado la clase propia de Java String (http://java.sun.com/javase/6/docs/api/java/lang/String.html) para definir nuevas clases.

Utilizaremos alguno de esos ejemplos para ilustrar la sintaxis en UML y Java de las relaciones de agregación.

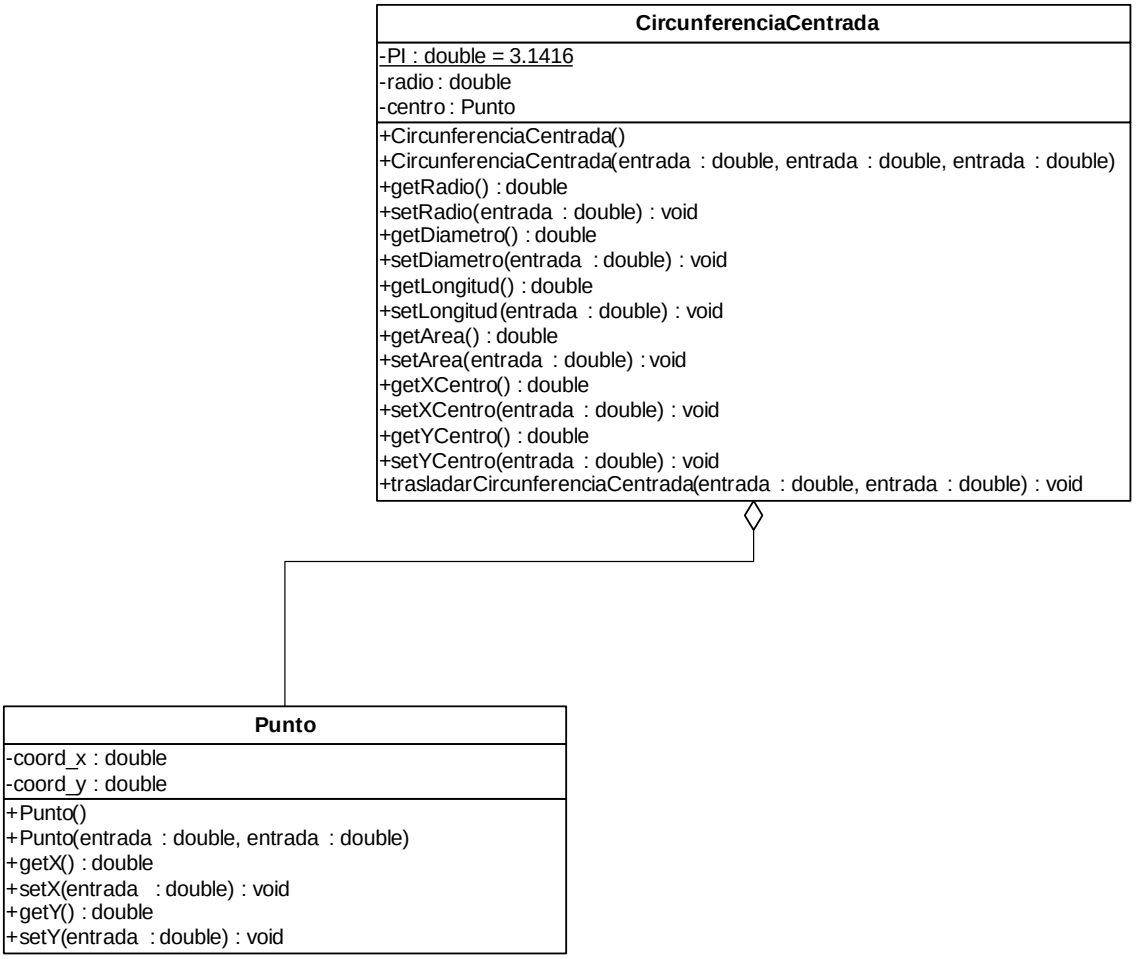


En la anterior imagen podemos observar cómo se representan en UML las relaciones de agregación. En la misma definimos la clase “CuentaBancaria” como el resultado de la agregación de dos atributos de la clase “String” (http://java.sun.com/javase/6/docs/api/java/lang/String.html) de Java, uno de los cuales representa el número de cuenta y el otro el nombre del titular de la misma.

La forma de representar una relación de agregación en UML es mediante una línea que sale de la clase que está siendo agregada (clase “parte”) hacia la clase resultante (clase “todo”), terminada en un rombo vacío en la clase que definimos por agregación.

Como explicábamos antes, se puede comprobar que el objeto “CuentaBancaria” “tiene un” atributo de la clase “String” (pero no “es un” objeto de la clase String, como veremos más adelante en las relaciones de herencia).

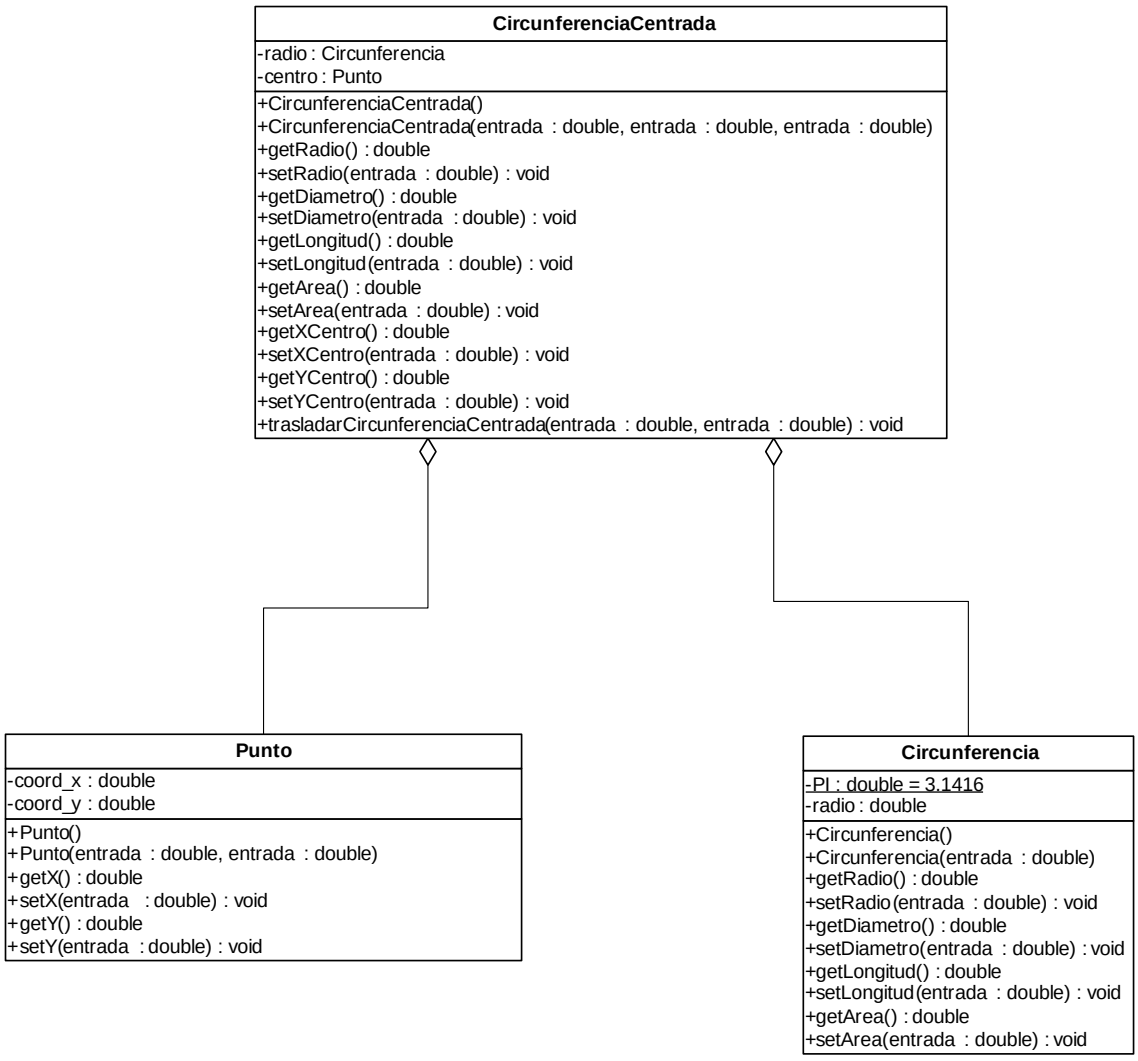
Podemos recuperar también ahora un ejemplo que aparecía en la Práctica 4.



En el mismo definimos la clase “CircunferenciaCentrada” como el resultado de la agregación de la clase “Punto”, que permite representar el centro de la circunferencia, a la clase “CircunferenciaCentrada”. De nuevo, se puede examinar la validez de la relación entre ambas clases por medio del test: Una “CircunferenciaCentrada” “tiene - un” atributo de tipo “Punto”.

Además, el diagrama de clases UML nos indica también cómo debemos implementar posteriormente la relación de agregación que introduce. En este caso, se hará por medio de un atributo de la clase “Punto” dentro de la clase “CircunferenciaCentrada”.

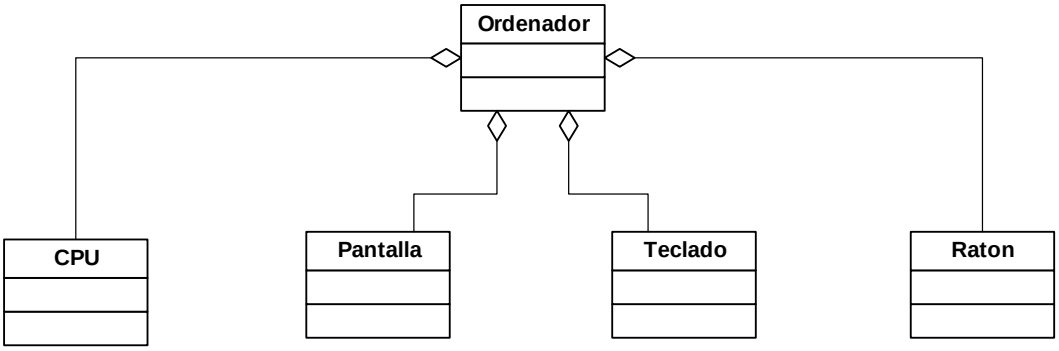
El anterior ejemplo admite diversas interpretaciones. Por ejemplo, un análisis distinto del mismo ejemplo podría haber dado lugar a la siguiente representación:



La anterior representación se basa en la idea de considerar una “CircunferenciaCentrada” como el resultado de agregar un atributo de la clase “Punto” y un atributo de la clase “Circunferencia”. Esta representación no tiene por qué ser errónea, pero podemos observar como gran parte de los métodos que hemos requerido de la clase “CircunferenciaCentrada” se encuentran replicados en la clase “Circunferencia”.

Más adelante veremos cómo una representación más adecuada será considerar la clase “CircunferenciaCentrada” como una clase que “tiene un” Punto (relación de agregración) y que “es una” “Circunferencia” (relación de herencia, que introduciremos en la Sección 2.3), lo cual nos permitirá acceder directamente a todos los métodos de la clase “Circunferencia”.

Otro ejemplo clásico de agregación es el siguiente (del cual no daremos detalles de implementación):



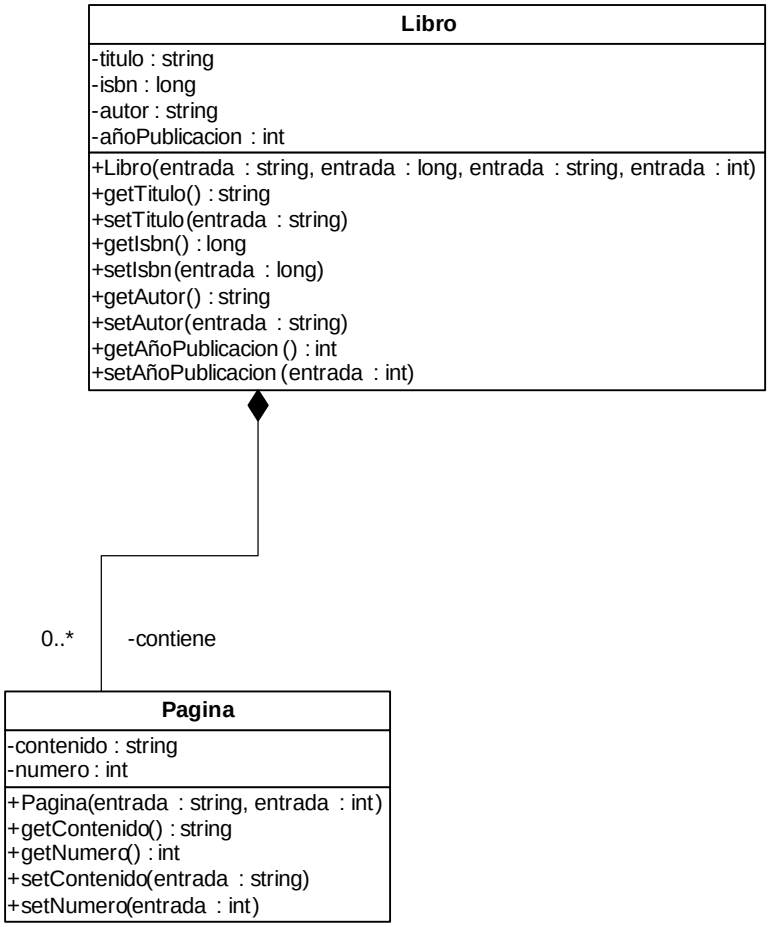
Un ordenador es el resultado de “agregar” una serie de componentes, como una “CPU”, una “Pantalla”, un “Teclado” y un “Raton” (y quizá algunos adicionales). De nuevo podemos aplicar el “test” de un objeto de la clase “Odenador” “tiene – una” “CPU” y “tiene – una” “Pantalla”, y “tiene – un” “Teclado” y “tiene – un” “Raton”.

En los ejemplos anteriores también se puede observar cómo los objetos de la clase “Punto” pueden existir independientemente de la clase “CircunferenciaCentrada”, o también objetos de las clases “CPU”, “Pantalla”, “Teclado” y “Raton” pueden existir con independencia de la clase “Ordenador”.

Basado en esta idea, existe un tipo particular de agregación, llamada **composición (o también agregación fuerte)**, en la cual los objetos agregados no tienen sentido fuera del objeto resultante. También se puede entender la composición como una relación en la que, los objetos siendo agregados, deben dejar de existir cuando lo hace el objeto compuesto.

Veamos lo anterior con algunos ejemplos.

Ejemplo 1: Un primer ejemplo que podemos considerar de composición es la relación que existe entre un libro y sus páginas:



En el mismo podemos observar tres nuevos elementos con respecto a los diagramas de clases anteriores que hemos presentado en UML:

1. En primer lugar, la relación de composición de denota con un rombo negro (en lugar de con uno blanco, como hacíamos en las relaciones de agregación).

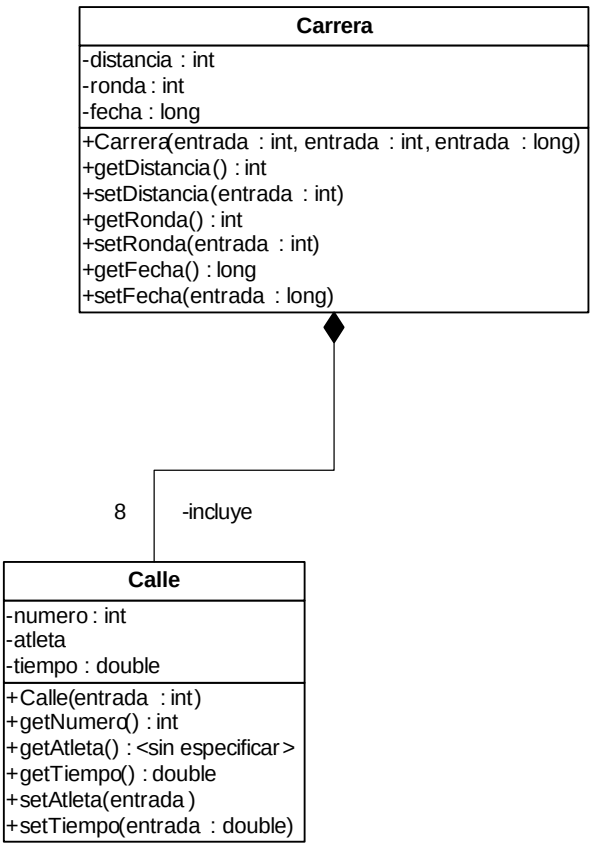
2. En segundo lugar, hemos añadido la etiqueta “0..*” a la flecha. Esto quiere decir que la relación tiene una multiplicidad entre 0 y cualquier número. La forma de entenderlo es que un objeto de la clase “Libro” puede contener cualquier número de objetos de la clase “Pagina”.

3. Finalmente, hemos etiquetado la relación con la palabra “contiene”, que nos da una idea de cómo se relacionan ambas clases.

Volviendo a la idea de por qué la relación es de composición (o una agregación fuerte), los objetos de la clase “Pagina” tienen sentido en tanto en cuanto formen parte de la clase “Libro”, que les asigna un “autor”, un “isbn” ... . Fuera de esta clase, un objeto de la clase “Pagina” no nos ofrecería una información de mucha utilidad.

En el diagrama de clases UML anterior se puede observar cómo no hemos incluido ninguna información de cómo debe ser implementada la relación de composición que hemos representado.

Ejemplo 2: Un segundo ejemplo de composición sería la relación existente entre una “Carrera” de atletismo y las distintas “Calles” que componen la misma. Veamos una posible representación en UML de la misma.



El gráfico anterior muestra la relación existente entre una carrera atlética y las distintas clases que componen la misma.

Tal y como está representado el gráfico anterior, la carrera estará compuesta por un número de calles entre 1 y 8. Como se puede observar, el atributo “atleta” lo hemos dejado sin asignarle un tipo (por el momento).

De nuevo, se puede entender la relación anterior como una composición, ya que los objetos de la clase ”Calle”, con un atleta y un tiempo determinado, no tienen sentido o significado fuera de la clase “Carrera”.

Sin embargo, no hemos proporcionado ninguna información de cómo debe ser implementada (programada) la relación anterior. Cuando representemos las anteriores relaciones en Java y C++ veremos diversas formas de programarlas.

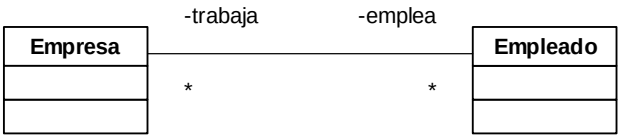
2.2.2 RELACIONES DE ASOCIACIÓN

Diremos que dos (o más) clases tiene una relación de **asociación** cuando una de ellas tenga que requerir o utilizar alguno de los servicios (es decir, acceder a alguna de las propiedades o métodos) de las otras.

Como se puede observar, las relaciones de asociación son más débiles que las relaciones de agregación, en el sentido de que no requieren (aunque en ocasiones lo implementemos así) que creemos un objeto nuevo a partir de otros objetos, sino únicamente que los objetos interactúen entre sí.

Las relaciones de asociación crean enlaces entre objetos. Estos enlaces no tienen por qué ser permanentes (es más, en la mayoría de los casos, no lo serán). Los objetos deben tener entidad fuera de la relación (a diferencia de las relaciones de composición).

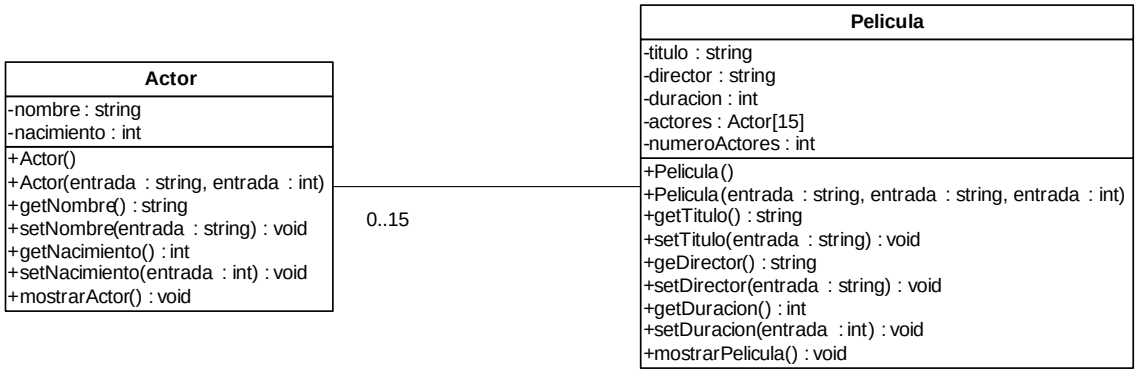
Un ejemplo clásico de relación de asociación es la relación “empresa – empleado”. Podemos definir una clase “Empresa” y una clase “Empleado” que nos permitan representar esta relación:



El anterior gráfico UML representa una relación de asociación. Las relaciones de asociación se representan en UML como una línea que une las dos clases siendo relacionadas.

En el ejemplo además vemos que esta relación se ha declarado de multiplicidad indefinida en ambos extremos, lo que quiere decir que un empleado puede trabajar en varias empresas (casos de pluriempleo) y, por supuesto, que una empresa puede estar formada por más de un empleado.

Este ejemplo nos recuerda también al ejemplo de la clase “Pelicula” y la clase “Actor” que presentamos en la Práctica 4, y cuya representación en UML es:



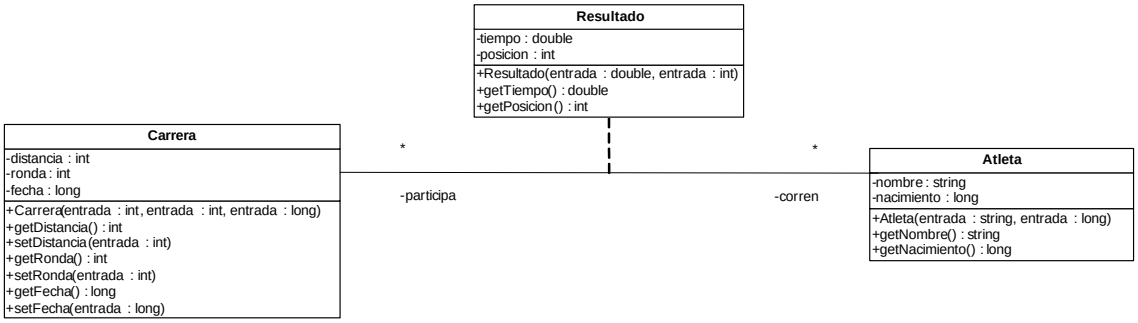
En el anterior diagrama UML hemos representado una relación de asociación entre la clase “Actor” y la clase “Pelicula”. Además, la hemos definido de tal forma que un objeto de la clase “Pelicula” podrá contener, como máximo, 15 actores.

Como podemos ver en el diagrama anterior, también se proporciona información de cómo ha de ser representada dicha relación de asociación. En

este caso, la relación se representará por medio de un “array” de objetos de la clase “Actor” incluido como un atributo en la clase “Pelicula”.

También podemos encontrar relaciones de asociación en las que no sea suficiente con representar la relación entre dos clases, sino que además debamos contemplar la aparición de propiedades adicionales vinculadas a la relación de asociación.

El siguiente ejemplo nos servirá para ilustrar esta situación. Tenemos una clase “Atleta” y una clase “Carrera”. Podemos apreciar que la relación puede ser múltiple en ambos extremos, ya que un objeto de la clase “Atleta” puede tomar parte en diversas carreras, del mismo modo que un objeto de la clase “Carrera” puede ser participado por múltiples atletas.



Pero aparte de la información anterior, también nos podría interesar conocer el resultado de los atletas en diversas carreras. Para ello definimos lo que se conoce como “Clase de Asociación”, que nos permite vincular dos objetos particulares de dos clases (en este caso, de las clases ”Atleta” y “Carrera”) y además añadir cierta información relevante al sistema que no tiene por qué estar vinculada a ninguna de ellas (en este caso, el resultado de un atleta en una carrera).

2.2.3 IMPLEMENTACIÓN DE RELACIONES DE AGREGACIÓN Y ASOCIACIÓN EN C++ Y JAVA

Los lenguajes de programación C++ y Java no tienen construcciones específicas para los distintos casos de relaciones que hemos visto en las Subsecciones anteriores. Esto quiere decir que hemos de utilizar formas más sencillas de implementar relaciones entre clases.

Algunas ideas de cómo implementar estas relaciones se pueden encontrar en el Capítulo 17 de “Practical Object Oriented Development with Java and UML”, de Richard Lee y William Tepfenhart, Prentice Hall (2002), en el Cap. 17.

En primer lugar comenzaremos por implementar alguno de los ejemplos de agregación que hemos visto. Por las peculiaridades anteriormente señaladas de las relaciones de agregación (son relaciones “todo - partes” en las cuales las distintas partes deben tener entidad propia independiente de la clase agregadora), la forma natural de representar relaciones de agregación es representar las distintas clases involucradas como clases independientes; las

distintas partes de la agregación se representan entonces como atributos de la clase agregadora.

Ejemplo 1: Recuperamos ahora el ejemplo en el que la clase "CircunferenciaCentrada" era definida por medio de la agregación de la clase "Punto" a una serie de atributos que nos permitían definir una circunferencia.

En general, las relaciones de agregación se trasladan a C++ y a Java por medio de la definición de atributos en la clase agregadora (en el "todo") que corresponden a las distintas "partes" que componen dicha clase.

De todos modos, C++ admite la representación de los atributos de las clases agregadas (en este caso la clase "Punto") **como un objeto** dentro de la clase agreadora ("CircunferenciaCentrada") o también como un **puntero a un objeto** de dicha clase ("Punto").

En nuestro caso, la clase "CircunferenciaCentrada" dispondrá de un atributo de tipo "Punto". Recuperamos en primer lugar la representación de la clase "Punto" (implementado por medio de coordenadas rectangulares):

```
//Fichero Punto.h

#ifndef PUNTO_H
#define PUNTO_H 1

class Punto {

    //Atributos de instancia
    private:
    double coord_x;
    double coord_y;

    //Constructores de instancia

    public:

    Punto ();
    Punto (double, double);

    //Métodos de instancia

    double getX ();
    void setX (double);
    double getY ();
    void setY (double);

};

#endif
```

```
//Fichero Punto.cpp

#include "Punto.h"

Punto::Punto (){
   this->coord_x = 0.0;
   this->coord_y = 0.0;
};

Punto::Punto (double coord_x, double coord_y){
    this->coord_x = coord_x;
    this->coord_y = coord_y;
};

double Punto::getX (){
    return this->coord_x;
};

void Punto::setX (double nueva_coord_x){
    this->coord_x = nueva_coord_x;
};

double Punto::getY (){
    return this->coord_y;
};

void Punto::setY (double nueva_coord_y){
    this->coord_y = nueva_coord_y;
};
```

2.2.3.1 REPRESENTACIÓN DE LA CLASE AGREGADA POR MEDIO DE UN OBJETO DE DICHA CLASE

```
//Fichero CircunferenciaCentrada.h
//El único detalle que debemos tener en cuenta es incluir la “declaración” de la
//clase “Punto” en la declaración de la clase “CircunferenciaCentrada”

#ifndef CIRCUNFERENCIA_CENTRADA_H
#define CIRCUNFERENCIA_CENTRADA_H 1

#include "Punto.h"

class CircunferenciaCentrada{
     //Constantes
     private:
     const static double PI = 3.1416;
     //Atributos
     private:
     double radio;
     Punto centro;
```

```
    //Constructores de instancias
    public:
    CircunferenciaCentrada ();
    CircunferenciaCentrada (double, double, double);

    //Métodos de instancia
    double getRadio ();
    void setRadio (double);
    double getDiametro ();
    void setDiametro (double);
    double getLongitud ();
    void setLongitud (double);
    double getArea ();
    void setArea (double);
    double getXCentro();
    void setXCentro (double);
    double getYCentro();
    void setYCentro(double);
    void trasladarCircunferenciaCentrada(double, double);
    };

#endif

//Fichero CircunferenciaCentrada.cpp

#include <cmath>
#include <iostream>
#include "CircunferenciaCentrada.h"

using namespace std;

//Constructores

CircunferenciaCentrada::CircunferenciaCentrada () {
   this->radio = 0.0;
   this->centro = Punto ();//No es necesario, ya que, internamente,
                           //en la propia cabecera del método,
                           //se ha construido “this->centro”
};

CircunferenciaCentrada::CircunferenciaCentrada (double radio, double
coord_x, double coord_y){
        this->radio = radio;
        this->centro = Punto (coord_x, coord_y);
        //Internamente ya se ha construido “this->centro” antes de entrar en el
        //cuerpo del constructor.
        //Bastaría con haber hecho “this->centro.setX(coord_x);” y
        //”this->centro.setY(coord_y);”
};
```

```
//Métodos

double CircunferenciaCentrada::getRadio (){
	return this->radio;
};

void CircunferenciaCentrada::setRadio (double nuevo_radio){
	this->radio = nuevo_radio;
};

double CircunferenciaCentrada::getDiametro (){
	return (2 * this->radio);
};

void CircunferenciaCentrada::setDiametro (double nuevo_diametro){
	this->radio = nuevo_diametro / 2;
};

double CircunferenciaCentrada::getLongitud (){
	return (this->radio * 2 * PI);
};

void CircunferenciaCentrada::setLongitud (double nueva_longitud){
	this->radio = nueva_longitud / (2 * PI);
};

double CircunferenciaCentrada::getArea (){
	return (this->radio * this->radio * PI);
};

void CircunferenciaCentrada::setArea (double nuevo_area){
	this->radio = sqrt (nuevo_area )/ PI;
};

double CircunferenciaCentrada::getXCentro (){
	return (this->centro.getX());
};

void CircunferenciaCentrada::setXCentro (double nueva_coord_x){
	this->centro.setX (nueva_coord_x);
};

double CircunferenciaCentrada::getYCentro (){
	return this->centro.getY();
};

void CircunferenciaCentrada::setYCentro (double nueva_coord_y){
	this->centro.setY (nueva_coord_y);
};
```

```
void CircunferenciaCentrada::trasladarCircunferenciaCentrada (double trasl_x, double trasl_y){
    this->centro.setX(this->centro.getX() + trasl_x);
    this->centro.setY(this->centro.getY() + trasl_y);
};
```

Notas:

1. Lo primero que debemos tener en cuenta es que al utilizar la clase “Punto” en la declaración de la clase “CircunferenciaCentrada” hemos de añadir el fichero de cabeceras “Punto.h” en el fichero de cabeceras “Circunferencia.h”.

2. En segundo lugar, la implementación de la clase agregada se hace por medio de un atributo en la clase “CircunferenciaCentrada”, al cual hemos llamado “centro”, de tipo “Punto”.

3. En tercer lugar, es importante destacar que en los constructores propios de la clase “CircunferenciaCentrada”, antes de que se entre en el cuerpo de dichos constructores, C++ se ha asegurado de que los objetos agregados (en este caso, “centro”) han sido adecuadamente construidos. Por tanto, C++ ha invocado internamente al constructor “Punto()” sobre “this->centro”. El propio lenguaje se encarga de inicializar todos los atributos representados como objetos. En caso de que no hubiese un constructor sin parámetros de las clases agregadas (en este caso, de la clase “Punto”), obtendríamos un error de compilación. El comportamiento de C++ es similar al que veíamos en el Tema 1, cuando insistíamos en el comportamiento del siguiente comando:

```
Punto p1;
```

En C++, el lenguaje automáticamente llamaba al constructor sin parámetros de la clase “Punto” sobre “p1”, para asegurarnos la construcción correcta de dicho objeto.

4. Por último, hemos de tener cuidado en la definición de la clase “CircunferenciaCentrada” de hacer uso, exclusivamente, de aquellos métodos que hayan sido definidos como visibles en la clase “Punto” (o la interfaz de la misma, que en este caso es la parte pública). Se puede observar esta situación, por ejemplo, en los constructores de la clase “CircunferenciaCentrada”.

Por lo demás, la definición de un programa principal o cliente de las clases anteriores no tendría ninguna particularidad.

2.2.3.2 REPRESENTACIÓN DE LA CLASE AGREGADA POR MEDIO DE UN PUNTERO A UN OBJETO DE DICHA CLASE

Implementaremos sólo la clase “CircunferenciaCentrada”, haciendo uso de la clase “Punto” introducida en el apartado previo.

```
//Fichero CircunferenciaCentrada.h
//El único detalle que debemos tener en cuenta es incluir la “declaración” de la
//clase “Punto” en la declaración de la clase “CircunferenciaCentrada”

#ifndef CIRCUNFERENCIA_CENTRADA_H
#define CIRCUNFERENCIA_CENTRADA_H 1

#include "Punto.h"

class CircunferenciaCentrada{
    //Constantes
    private:
    const static double PI = 3.1416;
    //Atributos
    private:
    double radio;
    Punto * centro;

    //Constructores de instancias
    public:
    CircunferenciaCentrada ();
    CircunferenciaCentrada (double, double, double);

    //Métodos de instancia
    double getRadio ();
    void setRadio (double);
    double getDiametro ();
    void setDiametro (double);
    double getLongitud ();
    void setLongitud (double);
    double getArea ();
    void setArea (double);
    double getXCentro();
    void setXCentro (double);
    double getYCentro();
    void setYCentro(double);
    void trasladarCircunferenciaCentrada(double, double);
    };

#endif

//Fichero CircunferenciaCentrada.cpp

#include <cmath>
#include <iostream>
#include "CircunferenciaCentrada.h"

using namespace std;

//Constructores
```

```
CircunferenciaCentrada::CircunferenciaCentrada () {
   this->radio = 0;
   this->centro = new Punto ();//Es responsabilidad del programador de la
                               //clase construir correctamente los atributos
                               //representados por punteros
};

CircunferenciaCentrada::CircunferenciaCentrada (double radio, double
coord_x, double coord_y){
        this->radio = radio;
        this->centro = new Punto (coord_x, coord_y);
        //Es responsabilidad del programador de la clase construir cada uno de
        //los atributos representados por punteros
};

//Métodos

double CircunferenciaCentrada::getRadio (){
        return this->radio;
};

void CircunferenciaCentrada::setRadio (double nuevo_radio){
        this->radio = nuevo_radio;
};

double CircunferenciaCentrada::getDiametro (){
        return (2 * this->radio);
};

void CircunferenciaCentrada::setDiametro (double nuevo_diametro){
        this->radio = nuevo_diametro / 2;
};

double CircunferenciaCentrada::getLongitud (){
        return (this->radio * 2 * PI);
};

void CircunferenciaCentrada::setLongitud (double nueva_longitud){
        this->radio = nueva_longitud / (2 * PI);
};

double CircunferenciaCentrada::getArea (){
        return (this->radio * this->radio * PI);
};

void CircunferenciaCentrada::setArea (double nuevo_area){
        this->radio = sqrt (nuevo_area )/ PI;
};
```

```
double CircunferenciaCentrada::getXCentro (){
        return (this->centro->getX());
};

void CircunferenciaCentrada::setXCentro (double nueva_coord_x){
        this->centro->setX (nueva_coord_x);
};

double CircunferenciaCentrada::getYCentro (){
        return this->centro->getY();
};

void CircunferenciaCentrada::setYCentro (double nueva_coord_y){
        this->centro->setY (nueva_coord_y);
};

void CircunferenciaCentrada::trasladarCircunferenciaCentrada (double trasl_x,
double trasl_y){
    this->centro->setX(this->centro->getX() + trasl_x);
    this->centro->setY(this->centro->getY() + trasl_y);
};
```

Comentarios: sólo incidiremos aquí en la diferencia con la anterior representación de los atributos en C++ por medio de objetos. Si los atributos son representados por punteros, es responsabilidad del programador reservar memoria para dichos punteros, y construir adecuadamente los objetos correspondientes. Si utilizamos la representación por medio de objetos, será el propio lenguaje el que internemente se encarge de dicha construcción.

2.2.3.4 REPRESENTACIÓN DE LA CLASE AGREGADA EN JAVA

El mismo ejemplo puede ser resuelto en Java de una forma bastante similar a como lo hemos hecho en C++ con punteros. Veamos primero la representación de la clase “Punto” en el mismo.

```
//Fichero Punto.java

public class Punto{

        private double coord_x;
        private double coord_y;

        public Punto (){
                this.coord_x = 0.0;
                this.coord_y = 0.0;
        }

        Punto (double coord_x, double coord_y){
                this.coord_x = coord_x;
```

```
		this.coord_y = coord_y;
	}

	double getX (){
		return this.coord_x;
	}

	void setX (double nueva_coord_x){
		this.coord_x = nueva_coord_x;
	}

	double getY (){
		return this.coord_y;
	}

	void setY (double nueva_coord_y){
		this.coord_y = nueva_coord_y;
	}
}

//Fichero CircunferenciaCentrada.java

import java.lang.Math;

public class CircunferenciaCentrada{

	private final static double PI = 3.1416;
	private double radio;
	private Punto centro;

	//Constructores

	public CircunferenciaCentrada () {
		this.radio = 0.0;
		this.centro = new Punto();
	}

	public CircunferenciaCentrada (double radio, double coord_x, double
coord_y){
		this.radio = radio;
		this.centro = new Punto (coord_x, coord_y);
	}

	//Métodos

	public double getRadio (){
		return this.radio;
	}

	public void setRadio (double nuevo_radio){
```

```
            this.radio = nuevo_radio;
        }

        public double getDiametro (){
            return (2 * this.radio);
        }

        public void setDiametro (double nuevo_diametro){
            this.radio = nuevo_diametro / 2;
        }

        public double getLongitud (){
            return (this.radio * 2 * PI);
        }

        public void setLongitud (double nueva_longitud){
            this.radio = nueva_longitud / (2 * PI);
        }

        public double getArea (){
            return (this.radio * this.radio * PI);
        }

        public void setArea (double nuevo_area){
            this.radio = Math.sqrt (nuevo_area )/ PI;
        }

        public double getXCentro (){
            return (this.centro.getX());
        }

        public void setXCentro (double nueva_coord_x){
            this.centro.setX (nueva_coord_x);
        }

        public double getYCentro (){
            return this.centro.getY();
        }

        public void setYCentro (double nueva_coord_y){
            this.centro.setY (nueva_coord_y);
        }

        public void trasladarCircunferenciaCentrada (double trasl_x, double
trasl_y){
            this.centro.setX(this.centro.getX() + trasl_x);
            this.centro.setY(this.centro.getY() + trasl_y);
        }
}
```

Notas:

1. Lo más reseñable de la anterior implementación en Java con respecto a la anterior en C++ es la ausencia del comando “include”. Debido a las reglas de visibilidad entre clases propias del lenguaje, al ser ambas clases públicas y encontrarse en el mismo “package” (o carpeta) ambas pueden comunicarse a través de su parte pública.

2. Por lo demás, la representación de la relación de agregación se hace introduciendo la clase agregada como un atributo en la clase agregadora que queremos definir.

3. Es importante observar que, al igual que nos sucedía en C++ cuando hemos utilizado punteros para representar los atributos agregados, el programador de la clase debe encargarse en los constructores de la clase “CircunferenciaCentrada” de reservar memoria para los atributos agregados (“centro”) y llamar a los constructores correspondientes (“Punto()” ó “Punto(double, double)”).

La implementación en Java es similar a la que hemos hecho en C++ cuando hemos utilizado punteros para alojar los atributos de las diversas clases. Este hecho tiene que ver con el modelo de memoria propio de Java, que, como comentábamos en el Tema 1, está basado en el uso de referencias para alojar los objetos. Por tanto, los atributos de un objeto de la clase “CircunferenciaCentrada” (en este caso, sólo “centro: Punto”) se implementan en memoria por medio de referencias, y dichas referencias deben ser reservadas y construidas adecuadamente.

Como conclusión de lo anterior, hemos podido ver por medio de diversos ejemplos cómo las relaciones de agregación se pueden representar por medio de atributos (por punteros u objetos) en la clase agregadora que representan las partes de dicha agregación.

En realidad, la metodología anterior es válida también para los casos de composición (o agregación fuerte)[1]. Veámoslo sobre el ejemplo que hemos introducido anteriormente en el que implementábamos una clase “Libro” y le añadíamos un parámetro que era una colección de objetos de la clase “Pagina”.

En el desarrollo del mismo hemos decidido que la colección de objetos de tipo “Página” que van a formar un libro se representará por medio de un “array”.

El código de dicho ejemplo en Java sería el siguiente:

//Fichero “Pagina.java”

public class Pagina {

[1] Java ofrece un mecanismo llamado “Clases internas”, o “Clases anidadas”, que posiblemente sea el más adecuado para la implementación de la relaciones de composición, pero por el momento no lo utilizaremos.

```
	//Atributos de instancia
	private String contenido;
	private int numero;

	//Constructores
	public Pagina (String contenido, int numero){
		this.contenido = new String (contenido);
		this.numero = numero;
	}

	//Metodos de instancia
	public String getContenido (){
		return this.contenido;
	}

	public void setContenido (String nuevo_contenido){
		this.contenido = nuevo_contenido;
	}

	public int getNumero (){
		return this.numero;
	}

	public void setNumero (int nuevo_numero){
		this.numero = nuevo_numero;
	}

}

//Fichero Libro.java

public class Libro{

	//Atributos
	private String titulo;
	private long isbn;
	private String autor;
	private int anyoPublicacion;

	//Atributos representando la relación de composición
	private Pagina [] paginas;
	private int numeroPaginas;

	//Constructores

	public Libro(String titulo, long isbn, String autor, int anyoPublicacion){

		this.titulo = titulo;
		this.isbn = isbn;
```

```
        this.autor = autor;
        this.anyoPublicacion = anyoPublicacion;
        //Reservamos espacio en memoria para el objeto “array”:
        this.paginas = new Pagina [999];
        //Reservamos espacio en memoria para las páginas:
        for (int i = 0; i < 999; i++){
                this.paginas [i] = new Pagina ("", 0);
        }
        this.numeroPaginas = 0;
}

//Metodos de instancia

public String getTitulo (){
        return this.titulo;
}

public void setTitulo (String titulo){
        this.titulo = titulo;
}

public long getIsbn (){
        return this.isbn;
}

public void setIsbn (long nuevo_isbn){
        this.isbn = nuevo_isbn;
}

public String getAutor (){
        return this.autor;
}

public void setAutor (String nuevo_autor){
        this.autor = nuevo_autor;
}

public int getAnyoPublicacion (){
        return this.anyoPublicacion;
}

public void setAnyoPublicacion (int nuevo_anyoPublicacion){
        this.anyoPublicacion = nuevo_anyoPublicacion;
}

//Métodos para trabajar con la clase compuesta
public int getNumeroPaginas (){
        return this.numeroPaginas;
}
```

```
	public void anyadirPagina (Pagina nueva_pagina){
		if (this.numeroPaginas < 999){
			this.paginas [this.numeroPaginas] = nueva_pagina;
			this.numeroPaginas++;
		}
	}

	public Pagina getPaginaNumero (int numero_pagina){
		for (int i = 0; i < this.numeroPaginas; i++){
			if (this.paginas[i].getNumero() == numero_pagina){
					return this.paginas[i];
				}
			}
		return null;
	}

}

//Fichero “Principal_EjemploLibroPagina.java”

public class Principal_EjemploLibroPagina{

	public static void main (String [] args){

		//Declaración y definición de objetos
		Libro el_quijote = new Libro ("Don Quijote de la Mancha", 1234567890, "Miguel de Cervantes", 1605);
		Pagina pagina1 = new Pagina ("En un lugar de la Mancha, de cuyo nombre ...", 1);
		Pagina pagina2 = new Pagina ("...no ha mucho tiempo que vivia un hidalgo de los de lanza en astillero, adarga antigua, rocin flaco y galgo corredor...",2);

		//Pasamos al objeto Libro los objetos del tipo Pagina
		el_quijote.anyadirPagina (pagina1);
		el_quijote.anyadirPagina (pagina2);

		for (int i = 1; i <= el_quijote.getNumeroPaginas(); i++){

	System.out.println(el_quijote.getPaginaNumero(i).getContenido());
		}
	}
}
```

Algunos comentarios sobre la implementación anterior:

1. En la clase “Pagina” hemos definido un atributo “numero” para cada objeto de dicha clase. Esto quiere decir que el orden de las páginas no es relativo a su posición en el “array”, sino a su atributo ”numero”.

2. Sobre la clase “Array” en Java. El tipo de dato “array” en Java y C++ difiere en varios aspectos. Para poder trabajar con “arrays” en Java de una forma correcta deberías tener en cuenta lo siguiente:

a) En Java el tipo de dato “array de ... “ se puede escribir de dos formas:

   Con los corchetes detrás del identificador del “array” (al estilo C/C++):

   Pagina paginas [];

   Con los corchetes delante del identificador del “array”:

   Pagina [] paginas;

   A lo largo del curso procuraremos utilizar la segunda de las notaciones, ya que parece más natural hablar del identificador “paginas” que es de tipo “Pagina []”.

b) Una segunda diferencia con C++ que se puede observar también ya en la anterior declaración es que en Java, para declarar un objeto de tipo “array”, no debemos especificar la dimensión del mismo. Sólo en la construcción del mismo deberá especificarse.

c) En tercer lugar, el tipo “array” en Java está representado por la clase “Array”. Puedes encontrar más detalles sobre ella en http://java.sun.com/javase/6/docs/api/java/lang/reflect/Array.html.
   Esto quiere decir que cada vez que se declare un **objeto** de la clase “Array”, también debe construirse. Puedes ver un ejemplo de lo anterior en la línea de código:

   this.paginas = new Pagina [999];

   Lo que hacemos es crear un objeto de la clase Array con espacio para 999 objetos de la misma. Ahora sí que debemos especificar un tamaño para el mismo.

   Recuerda la diferencia entre declarar, donde no especificamos el tamaño del “array”, y construir, donde sí debemos hacerlo.

   Una vez creado el “array”, debemos considerar también los objetos que forman parte del mismo que, de momento, no han sido construidos. Aquí existen diversas opciones. Una de ellas es dejar todos esos elementos con el valor que Java les ha asignado al crear el objeto “array”, es decir, “null”. En el Punto 3 de esta enumeración explicamos en más detalle el significado y comportamiento de “null”. Otra posibilidad, la que hemos seguido en nuestro código, es inicializar todas y cada una de las posiciones del mismo invocando a un constructor. Esto nos asegura que cada componente del “array” contendrá una referencia a un objeto que ha sido alojado en una

región de memoria distinta. Esto lo conseguimos haciendo, por ejemplo:

```
for (int i = 0; i < 999; i++){
        this.paginas [i] = new Pagina ("", 0);
        }
```

Una tercera posibilidad sería hacer que las componentes del mismo contuvieran referencias ya existentes a objetos de la clase. Esta forma es menos segura, ya que al modificar dichas referencias estaremos también modificando las componentes del "array".

d) Al ser "this.paginas" un objeto, deberías prestar atención a los métodos de la clase que puedes utilizar (http://java.sun.com/javase/6/docs/api/java/lang/reflect/Array.html).

e) El acceso y modificación de las componentes de un "array" se hace de modo similar a como se hacía en C++, por medio de las posiciones correspondientes; conviene recordar que los índices de las posiciones de un "array" comienzan en 0.

f) Al ser un "array" en Java un objeto, está implementado en memoria por una referencia. Como consecuencia de esto, en Java podemos copiar un objeto "array" en otro por simple asignación:

```
int array1 [] = new int [50];
int array2 [] = new int [50];
array1 = array2;
```

En el código anterior, la referencia del objeto "array1" estará apuntando a la misma dirección de memoria que la referencia del objeto "array2", es decir, ambos objetos están apuntando a la misma región de memoria (al mismo "array").

A partir de ese momento, todos los cambios que se hagan sobre "array1" sucederán también sobre "array2" (y vivecersa).

3. Si observas el cuerpo del método "getPaginaNumero(): Pagina", verás el uso de la constante "null"

```
public Pagina getPaginaNumero (int numero_pagina){
        for (int i = 0; i < this.numeroPaginas; i++){
                if (this.paginas[i].getNumero() == numero_pagina){
                        return this.paginas[i];
                }
        }
        return null;
}
```

La constante “null” representa el objeto vacío en Java. A cualquier objeto le podemos asignar el valor “null”. Esto permite, por ejemplo, completar la definición del método anterior incluso en los casos que no deberían suceder. Sin embargo, puede ser peligroso el uso excesivo de la misma, ya que si sobre el objeto “null” tratamos de invocar a cualquier método de una clase:

```
null.getContenido();
```

Obtendremos lo que en Java se conoce como una “NullPointerException” que detendrá la ejecución de nuestro programa. En el Tema 5 volveremos a hablar sobre excepciones y cómo gestionarlas.

En el siguiente ejemplo, que presentamos en C++, daremos detalles sobre el trabajo con “arrays” de objetos en C++.

Veamos cómo se pueden implementar las relaciones de asociación, que, además, tienen asociada una multiplicidad distinta de cero en alguno de sus extremos. Implementamos el ejemplo en C++ donde la clase “Pelicula” y la clase “Actor” poseían una relación de asociación.

```
//Fichero Actor.h

#ifndef ACTOR_H
#define ACTOR_H 1

class Actor{

    private:
    char nombre[30];
    int nacimiento;

    public:
    Actor();
    Actor(char [], int);
    char * getNombre();
    void setNombre (char []);
    int getNacimiento();
    void setNacimiento(int);
    void mostrarActor();

};

#endif

//Fichero Actor.cpp

#include <cstring>
#include <iostream>
#include "Actor.h"
```

```
using namespace std;

Actor::Actor(){
    strcpy (this->nombre, “”);
    this->nacimiento = 0;
};

Actor::Actor(char nombre [], int nacimiento){
    strcpy (this->nombre, nombre);
    this->nacimiento = nacimiento;
};

char * Actor::getNombre(){
    return this->nombre;
};

void Actor::setNombre (char nombre []){
    strcpy (this->nombre, nombre);
};

int Actor::getNacimiento(){
   return this->nacimiento;
};

void Actor::setNacimiento(int nacimiento){
    this->nacimiento = nacimiento;
};

void Actor::mostrarActor(){
    cout << "El nombre del Actor es " << this->getNombre() << endl;
    cout << "Su anyo de nacimiento es " << this->getNacimiento() << endl;
    cout << endl;
};

//Fichero Pelicula.h

//En este fichero podemos observar como los actores que tiene asociados
//un objeto de la clase “Pelicula” se representan por medio de un “array” de
//objetos de la clase “Actor”

#ifndef PELICULA_H
#define PELICULA_H 1

#include "Actor.h"

class Pelicula{
    //Atributos de instancia
     private:
     char titulo [30];
     char director [30];
```

```
        int duracion;

       //Atributos que van a representar la relación de asociación
        Actor actores [15];
        int numeroActores;


        public:
        //Consructores
        Pelicula();
        Pelicula(char [], char [], int);
        //Métodos de instancia
        char * getTitulo();
        void setTitulo(char []);
        char * getDirector();
        void setDirector(char []);
        int getDuracion();
        void setDuracion(int);
        void mostrarPelicula();
        bool isActor(Actor);
        void introduceActor(Actor);
};

#endif

//Fichero Pelicula.cpp

#include <cstring>
#include <iostream>
#include "Pelicula.h"

using namespace std;

Pelicula::Pelicula (){
        strcpy (this->titulo, "");
        strcpy (this->director, "");
        this->duracion = 0;
        this->numeroActores = 0;
}

Pelicula::Pelicula (char titulo [], char director [], int duracion){
        strcpy (this->titulo, titulo);
        strcpy (this->director, director);
        this->duracion = duracion;
        this->numeroActores = 0;
}

char * Pelicula::getTitulo (){
         return this->titulo;
}
```

```
void Pelicula::setTitulo (char nuevo_titulo []){
        strcpy (this->titulo, nuevo_titulo);
}

char * Pelicula::getDirector (){
        return this->director;
}

void Pelicula::setDirector (char nuevo_director []){
        strcpy (this->director, nuevo_director);
}

int Pelicula::getDuracion (){
        return this->duracion;
}

void Pelicula::setDuracion (int nueva_duracion){
        this->duracion = nueva_duracion;
}

void Pelicula::mostrarPelicula(){
        cout << "El nombre de la pelicula es " << this->getTitulo() << endl;
        cout << "Su director es " << this->getDirector() << endl;
        cout << "la duracion de la pelicula es " << this->getDuracion() << endl;
        for (int i = 0; i < this->numeroActores; i++){
            cout << "En la peli trabaja: " << endl;
            this->actores[i].mostrarActor ();
        }
}

bool Pelicula::isActor(Actor actor){
    bool encontrado = false;
    for (int i = 0; i < numeroActores; i++){
       if (strcmp (actor.getNombre(), this->actores[i].getNombre()) == 0 &&
actor.getNacimiento() == this->actores[i].getNacimiento())
         {encontrado = true;}
    }
    return encontrado;

};

void Pelicula::introduceActor(Actor actor){
        if (this->numeroActores < 15){
          this->actores[this->numeroActores] = actor;
        this->numeroActores++;
        }
};

//Fichero con la función main
```

```
#include <cstdlib>
#include <iostream>
#include "Actor.h"
#include "Pelicula.h"

using namespace std;

int main (){

   Pelicula la_guerra ("La guerra de las Galaxias", "George Lucas", 121);
   Actor mark ("Mark Hamill", 1951);
   Actor harrison ("Harrison Ford", 1942);
   Actor carrie ("Carrie Fisher", 1956);
   Actor alec ("Alec Guinness", 1914);

   la_guerra.introduceActor (mark);
   la_guerra.introduceActor (harrison);
   la_guerra.introduceActor (carrie);
   la_guerra.introduceActor (alec);

   la_guerra.mostrarPelicula();

   cout << "Alec Guinness trabajo en " << la_guerra.getTitulo() << "??" << endl;
   cout << "La respuesta es " << la_guerra.isActor (alec) << endl;

   system ("PAUSE");
   return 0;
}
```

Algunos detalles sobre la implementación:

1. En primer lugar, debemos mencionar cómo hemos resuelto la implementación de la relación de asociación entre las clases “Pelicula” y “Actor”. Hay diversas soluciones al respecto. En este caso, siguiendo el diagrama UML correspondiente, nos hemos decantado por representar la relación dentro de la clase “Pelicula”, como un atributo de la misma, que contenga un “array” de objetos de la clase “Actor” (por cierto, solución similar a la que hemos aplicado al implementar la relación de composición entre la clase “Pagina” y la clase “Libro”; como ya mencionábamos al comienzo de esta Sección, las relaciones de asociación y las de composición a menudo se implementan en Java y C++ con similares mecanismos).

2. En C++, a diferencia de Java, para poder utilizar una clase por parte de otra, hemos de incluir las sentencias “include” correspondientes. Como norma general, si una clase hace uso de otra (como atributo, por ejemplo), el fichero “.h” de la primera debería llevar una sentencia “#include” de la segunda.

3. Es importante recordar que, en el “array” de objetos de la clase “Actor” que utilizamos, todas y cada una de las componentes del mismo deben ser

construidas apropiadamente en el constructor de la clase “Pelicula”. Recuperamos ahora la definición de los constructores de la clase “Pelicula”:

```
Pelicula::Pelicula (){
        strcpy (this->titulo, "");
        strcpy (this->director, "");
        this->duracion = 0;
        this->numeroActores = 0;
}

Pelicula::Pelicula (char titulo [], char director [], int duracion){
        strcpy (this->titulo, titulo);
        strcpy (this->director, director);
        this->duracion = duracion;
        this->numeroActores = 0;
}
```

Se puede observar que ninguno de ellos ha prestado atención a la inicialización del atributo “this->actores” (en Java debimos realizar la construcción del “array” así como de todas y cada una de sus componentes) ¿Por qué no lo hemos hecho?

En primer lugar, el “array” no requiere de inicialización, porque el tipo “array” no corresponde a una clase en C++ (a diferencia de Java, en este caso). Esto quiere decir que no debemos reservar memoria para él; se supone que el sistema lo hace por nosotros (como con los tipos básicos, “int”, “double”, “bool”,...).

En segundo lugar, C++ realiza la construcción de los objetos del “array” de forma automática. Esto quiere decir que, a pesar de que en el código no podamos apreciarlo, C++ ha invocado internamente, tantas veces como componentes tenga el “array”, al constructor “Actor()”.

Como consecuencia de lo anterior, siempre que queramos trabajar con “arrays” de objetos en C++, debemos declarar un constructor por defecto (sin parámetros) de objetos para esa clase.

Como C++ también permite trabajar con memoria dinámica, también existe la posibilidad de trabajar con “arrays” de punteros a objetos:

```
        private: actores * Actor [15];
```

Un puntero a un objeto no tiene por qué construirse (ni hace falta reservar memoria para el mismo), por lo cual los constructores de la clase Película podrían seguir siendo:

```
Pelicula::Pelicula (){
        strcpy (this->titulo, "");
        strcpy (this->director, "");
        this->duracion = 0;
```

```
        this->numeroActores = 0;
}

Pelicula::Pelicula (char titulo [], char director [], int duracion){
        strcpy (this->titulo, titulo);
        strcpy (this->director, director);
        this->duracion = duracion;
        this->numeroActores = 0;
}
```

En la anterior definición, las componentes del “array” “this->actores” no han sido construidas. Esto se podría evitar con la definición de los constructores como:

```
Pelicula::Pelicula (){
        strcpy (this->titulo, "");
        strcpy (this->director, "");
        this->duracion = 0;
        for (int i = 0; i < 15; i++){
                this->actores [i] = new Actor;
        }
        this->numeroActores = 0;
}

Pelicula::Pelicula (char titulo [], char director [], int duracion){
        strcpy (this->titulo, titulo);
        strcpy (this->director, director);
        this->duracion = duracion;
        for (int i = 0; i < 15; i++){
                this->actores [i] = new Actor;
        }
        this->numeroActores = 0;
}
```

Como conclusión a lo anterior, es importante recordar que cuando se trabaje con “arrays” de objetos debemos definir un constructor sin parámetros para la clase correspondiente. Si trabajamos con “arrays” de punteros a objetos, la construcción y reserva de memoria para las componentes del “array” debe ser realizada por el programador.

Después de hablar de relaciones de agregación (composición) y de asociación entre clases y de haber introducido diversos ejemplos de las mismas, pasamos ahora a hablar de un tipo de relación distinto, las llamadas relaciones de especialización/generalización, o de herencia entre clases, que nos ocuparán el resto del Tema 2.

2.3 RELACIONES DE ESPECIALIZACIÓN/GENERALIZACIÓN

Existe un tipo de relación distinta entre clases que no ha sido contemplada en la Sección anterior. Es la relación de especialización/generalización (o de

herencia) entre dos clases. Esta relación se considera propia de los lenguajes de POO.

Una relación de herencia de la clase B (subclase, o clase hija) con respecto a A (superclase o clase base) nos permite decir que la clase B obtiene todos los métodos y atributos de la clase A, y que luego puede añadir algunas características propias. De este modo podemos crear jerarquías de clases que comparten ciertas propiedades de una forma mas ágil.

En el caso anterior se supone que utilizaremos la relación de herencia para decir que la clase B hereda de la clase A. Por medio de este mecanismo, la clase B comparte todas las características (atributos o estado y métodos o comportamiento) de la clase A. Esto no impide que se le pueda añadir a la clase B características adicionales (de nuevo, tanto atributos como métodos), o incluso, como veremos en la Sección 2.6, se modifique el comportamiento (la definición, no la declaración) de alguno de los métodos heredados.

La creación de jerarquías de clases por medio de relaciones de herencia no sólo tiene consecuencias importantes desde el punto de vista de la reescritura de código. También afecta al sistema de tipos: en el ejemplo sobre las clases A y B, ¿cuál será el tipo ahora de los objetos de la clase B? ¿Existe alguna relación entre su tipo y el tipo de los objetos de la clase A? Los objetos de la clase B son un subtipo de la clase A (comparten, al menos, todos los atributos, y la declaración, quizá no la definición, de todos los métodos de A). Esto quiere decir que, si una función método admite un parámetro de la clase A, también debería admitir un objeto de la clase B como parámetro.

Esto da lugar a la noción de polimorfismo, que introduciremos en el Tema 3, y que es otra de las principales características de la POO.

Las relaciones de herencia son aquéllas que responden al predicado “es - un” (“is-a” en inglés). Diremos que una clase B hereda de otra clase A cuando la clase B “es - una” clase A, y, además puede añadir nuevas propiedades. Esto nos permite distinguir las relaciones de herencia de las relaciones de agregación, que responden al predicado “tiene - un”. Podemos ilustrar el uso de ambos predicados sobre ciertos ejemplos.

Una clase “Estudiante” “tiene - una” clase “Asignatura” como atributo, quizá con multiplicidad mayor que cero (pero un Estudiante no “es - una” Asignatura luego no hay relación de herencia entre ambas clases). En este caso tendríamos una relación de agregación (que también se podría representar como una asociación).

Un objeto de la clase “CircunferenciaCentrada”, en particular, “es – una” “Circunferencia”, y esta relación se representará de forma más adecuada por una relación de herencia (aunque también admite la interpretación de que una “CicunferenciaCentrada” tiene una “Circunferencia”, y por tanto es el resultado de una agregación).

Una “Ventana con campos de texto” en una interfaz gráfica “es - una” “Ventana“ (la relación de herencia se ajusta mejor de nuevo en este caso). Sin embargo, una “Ventana con campos de texto” no “es -un” “Campo de texto”, sino que “tiene - un“ atributo ”Campo de texto”. En este segundo caso la relación de agregación sería la correcta.

Siempre que tengas que definir una relación entre dos clases deberías realizar la anterior comprobación que te permita identificar qué tipo de relación debes definir entre ellas (herencia, agregación, o asociación).

Veamos ahora con un sencillo ejemplo algunas de las ideas que hemos comentado en los párrafos anteriores:

```
//Fichero Principal_AB.java

class A{

	private int unAtributo;
	private String otroAtributo;

	public A (){
		this.unAtributo = 32;
		this.otroAtributo = new String ("estamos en la clase A");
	}

	public int getUnAtributo(){
		return this.unAtributo;
	}

	public String getOtroAtributo(){
		return this.otroAtributo;
	}
}

class B extends A{
	public static final int CONSTANTEB = 25;
}

public class Principal_AB {
	public static void main (String [] args){

	A objetoA = new A();
	B objetoB = new B();
	System.out.println (objetoA.getOtroAtributo());
	System.out.println (objetoB.getOtroAtributo());
	System.out.println ("Constante en B " + objetoB.CONSTANTEB);
	//La siguiente orden provoca un error de compilación
	//CONSTANTEB no es un atributo de la clase A
	//System.out.println ("Constante en A " + objetoA.CONSTANTEB);
	}
```

}

Notas:

1. En primer lugar, se puede observar cómo en un mismo fichero “*.java” hemos podido definir tres clases distintas. Esto es posible en Java siempre y cuando haya una única clase de tipo “public” en el fichero (el fichero deberá tener el mismo nombre que dicha clase). En general, es preferible utilizar ficheros distintos para clases distintas.

2. Podemos observar cómo se define una clase “A” que contiene dos atributos, un constructor, y dos métodos “get” de acceso a los atributos.

3. También podemos observar que se ha definido una nueva clase “B”, cuya definición es:

```
class B extends A{
        public static final int CONSTANTEB  = 25;
}
```

La palabra reservada “extends” es la que nos permite en Java definir relaciones de herencia entre dos clases. Por medio de la misma, decimos que la clase B hereda de la clase A (heredamos la declaración y la definición de todos sus atributos y métodos).

4. También podemos observar que en la clase B sólo hemos añadido una constante de clase (static final) que hemos declarado como “public”, lo cual quiere decir que será visible desde fuera de la clase B (haciendo caso de la declaración de herencia, la clase B está formada ahora por los atributos y métodos propios de la clase A, y por la constante añadida).

5. El ejemplo nos sirve también para observar una propiedad sobre la declaración y definición de constructores en Java (y C++): si de una clase (en este caso, de B) no definimos ningún constructor, Java (y también C++) nos proveerá de uno por defecto (sin parámetros, y, cuyo comportamiento definirán Java o C++).

6. Por último, en el programa principal, podemos ver cómo, al declarar y construir un objeto de la clase B, se han “heredado” los métodos y atributos propios de la clase A de forma directa. Además, en los objetos de la clase B podemos acceder a la constante “CONSTANTEB”. Sin embargo, los objetos de la clase A no disponen de este atributo (la clase B añade funcionalidad a la clase A, en este caso, una constante).

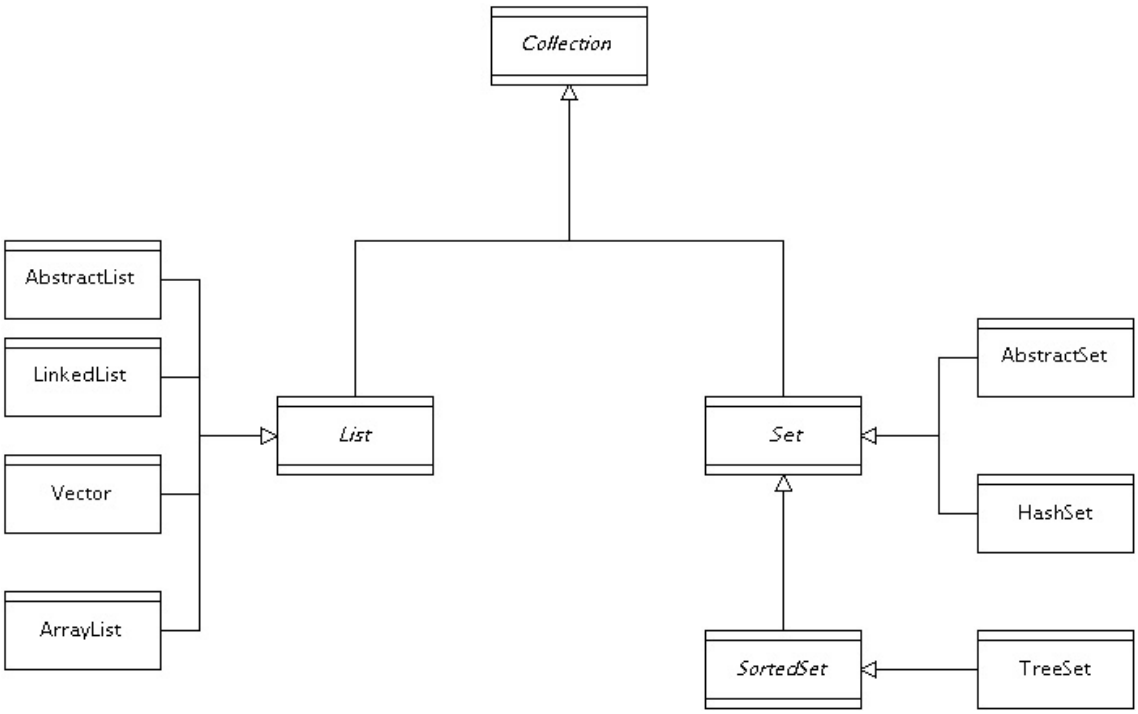
Para terminar esta Sección, vamos a ver un ejemplo más elaborado de un “árbol” de herencias extraído de la librería de Java.

En el mismo aparecen mezcladas clases e interfaces (que no veremos hasta el Tema 4). Simplemente pretendemos ilustrar cómo diversas representaciones

de agrupaciones de objetos pueden ser representadas prestando atención a las relaciones existentes entre las mismas.

En este caso, la noción más general es la de Collection, que es una agrupación de elementos (quizá duplicados). A partir de “Collection” se definen dos especializaciones de la misma, “Set”, una agrupación de elementos que no pueden estar duplicados, y “List”, una agrupación de elementos que posee un orden. Podemos ver cómo la noción de “List” puede luego especializarse a las nociones de “LinkedList”, o lista enlazada, “Vector”, una lista implementada por un “array” cuyo tamaño puede ser aumentado, o “ArrayList”, similar en muchos aspectos a “Vector” (puedes buscar en la API de Java la definición concreta de cada una de estas clases).

Relaciones similares (de especialización) se pueden observar en la parte derecha del “árbol de herencia” de la figura. Vemos como la noción de ”Set” está especializada en la noción de “SortedSet”, así como en las de “HashSet” y “AbstractSet”.



El ejemplo anterior ilustra bastante bien cómo está estructurada la información internamente en la librería de Java. Generalmente, todas las clases guardan relaciones de especialización/generalización con otras clases, dando lugar a árboles de herencia como el que acabamos de mostrar.

2.4 DEFINICIÓN DE LA RELACIÓN DE HERENCIA ENTRE CLASES

En esta Sección presentamos una definición concreta de herencia en los lenguajes de POO. También presentaremos ciertos ejemplos de la misma. Introduciremos la notación UML de relaciones de herencia, así como la sintaxis propia de Java y C++.

Definición: la herencia es una forma de declarar nuevas clases a partir de clases ya existentes. La nueva clase (clase derivada, subclase) hereda todos los atributos y la declaración de los métodos de la clase base (o súperclase). Además, la subclase puede añadir nuevos atributos y métodos. Como forma de identificar las relaciones de herencia, podemos hacer uso del test “es - un” entre las clases que queremos relacionar.

Los principales objetivos de la definición de relaciones de herencia entre clases son la reutilización de código (todos los atributos y métodos de la clase base estarán presentes en las clases derivadas) y la creación de una categorización entre clases que permita agrupar a clases que comparten propiedades.

Cuando una clase B hereda de una clase A se pueden distinguir dos situaciones:

1 Que B sea un “buen hijo” de A. Esto quiere decir que los métodos de A conservan su comportamiento en la clase B (todos los métodos que teníamos en A preservan su comportamiento en la clase B).

2. Que B sea un “mal hijo” de A. Esto quiere decir que la clase B va a modificar el comportamiento de alguno de los métodos que ha heredado de la clase A. En este caso, hablaremos de “redefinición” de métodos. Volveremos a esta noción en la Sección 2.6.

La capacidad para declarar relaciones de especialización/generalización es una de las principales características de los lenguajes orientados a objetos.

Java está diseñado de tal modo que todas las clases en el lenguaje heredan de una clase común, llamada “Object” (cuya especificación puedes encontrar en http://java.sun.com/javase/6/docs/api/java/lang/Object.html). Esto lo puedes comprobar haciendo uso del método “”toString(): String” sobre cualquier objeto de cualquier clase en Java (tanto las definidas por el usuario como las propias de la librería).

En C++, sin embargo, no existe una jerarquía que posea una clase de estas características dentro del lenguaje.

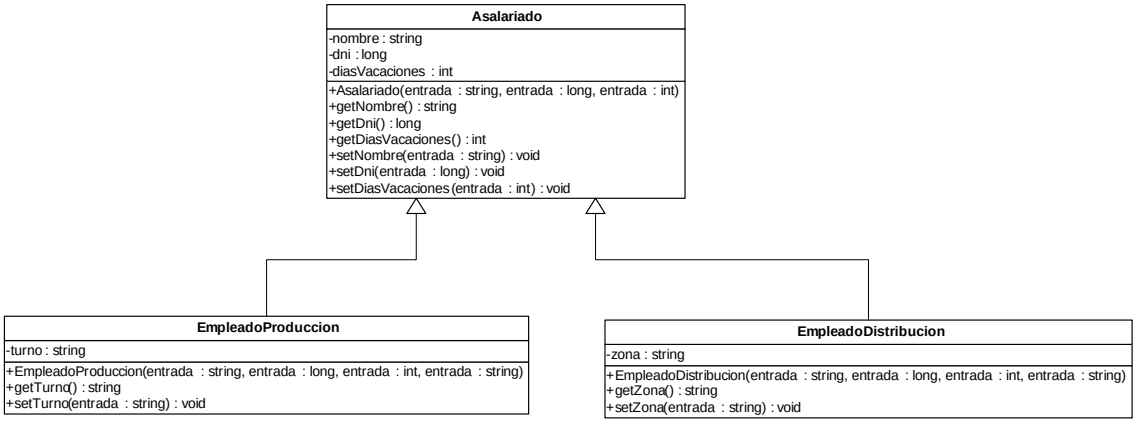
Conviene mencionar también que, si bien en UML podemos representar relaciones de herencia múltiple (donde una clase hereda a la vez de varias clases), esto no está permitido en Java. Sólo se pueden definir herencias simples (donde una clase hereda sólo de otra clase). En C++ sí que se pueden implementar herencias múltiples, pero este caso no lo abordaremos en este curso.

Veamos un sencillo ejemplo de una relación de herencia que nos permita introducir la sintaxis propia de UML, C++ y Java para declarar tales relaciones. Imaginamos el siguiente escenario: queremos representar por medio de clases a los trabajadores de una empresa. Dentro de la misma hay diversas categorías con respecto al puesto que desempeñan en la misma.

2.4.1 REPRESENTACIÓN DE RELACIONES DE HERENCIA EN UML

Definiremos una clase “Asalariado” que nos permita representar a todos los trabajadores de la firma. A partir de esa clase, definiremos dos subclases “EmpleadoProduccion” y “EmpleadoDistribucion” que representan a los trabajadores de dicha firma que trabajan en las divisiones de producción y distribución respectivamente. Los trabajadores de producción trabajan por turnos (“Mañana”, “Tarde”, “Noche”). Los de distribución tienen asignada una zona (o región) donde deben distribuir los productos, pero no trabajan por turnos.

El diagrama de clases en UML que nos permite representar la anterior situación podría ser el siguiente (estamos interesados únicamente en conocer el nombre, DNI, y el número de días de vacaciones que disfrutará cada trabajador):



En primer lugar, podemos verificar que las clases están vinculadas por una relación de herencia. En particular, un objeto de la clase “EmpleadoProduccion” “es - un” “Asalariado”, del mismo modo que un objeto de la clase “EmpleadoDistribucion” “es - un” “Asalariado”.

En el gráfico podemos ver cómo las relaciones de herencia en UML se representan por medio de una línea terminada en una flecha cuyo centro es blanco. Además, todos los atributos que aparecen en la clase base (“Asalariado” en nuestro caso) no deben aparecer en las clases derivadas. De igual manera, los métodos que aparezcan en la clase base y cuyo comportamiento no varíe en las clases derivadas tampoco aparecerán en las mismas. En las clases derivadas sólo debemos incluir los atributos y métodos que no aparecían en la clase base (y aquéllos que redefinamos, como veremos en la Sección 2.6).

Siguiendo el anterior diagrama, disponemos de tres clases, “Asalariado”, “EmpleadoProduccion” y “EmpleadoDistribucion” entre las cuales hemos definido una relación de herencia. Veamos cómo podemos representar dicho diagrama en C++.

2.4.2 DECLARACIÓN Y DEFINICIÓN DE RELACIONES DE HERENCIA EN C++

```
//Fichero “Asalariado.h”

#ifndef ASALARIADO_H
#define ASALARIADO_H 1

class Asalariado{
  //Atributos de instancia
  private:
          char nombre [30];
          long dni;
          int diasVacaciones;
  public:
  //Constructor
  Asalariado(char[], long, int);
  //Métodos de instancia:
  char * getNombre ();
  void setNombre (char[]);
  long getDni ();
  void setDni (long);
  int getDiasVacaciones ();
  void setDiasVacaciones (int);

};

#endif

//Fichero “Asalariado.cpp”

#include <cstring>
#include "Asalariado.h"

using namespace std;

Asalariado::Asalariado(char nombre[], long dni, int diasVacaciones){
   strcpy (this->nombre, nombre);
   this->dni = dni;
   this->diasVacaciones = diasVacaciones;
};

char * Asalariado::getNombre (){
     return this->nombre;
};

void Asalariado::setNombre (char nuevo_nombre[]){
     strcpy (this->nombre, nuevo_nombre);
};
```

```
long Asalariado::getDni (){
    return this->dni;
};

void Asalariado::setDni (long nuevo_dni){
    this->dni = nuevo_dni;
};

int Asalariado::getDiasVacaciones (){
    return this->diasVacaciones;
};

void Asalariado::setDiasVacaciones (int nuevo_diasVacaciones){
    this->diasVacaciones = nuevo_diasVacaciones;
};

//Fichero “EmpleadoProduccion.h”

#ifndef EMPLEADOPRODUCCION_H
#define EMPLEADOPRODUCCION_H 1

#include "Asalariado.h"

class EmpleadoProduccion: public Asalariado{
  //Atributos de instancia
  private:
        char turno [10];
  public:
  //Constructor
  EmpleadoProduccion(char[], long, int, char[]);
  //Métodos de instancia:
  char * getTurno ();
  void setTurno (char[]);
};

#endif

//Fichero “EmpleadoProduccion.cpp”

#include <cstring>
#include "EmpleadoProduccion.h"

EmpleadoProduccion::EmpleadoProduccion(char nombre[], long dni, int
diasVacaciones, char turno[]): Asalariado (nombre, dni, diasVacaciones){
   strcpy (this->turno, turno);
};

char * EmpleadoProduccion::getTurno (){
    return this->turno;
};
```

```
void EmpleadoProduccion::setTurno (char nuevo_turno[]){
    strcpy (this->turno, nuevo_turno);
};

//Fichero “EmpleadoDistribucion.h”

#ifndef EMPLEADODISTRIBUCION_H
#define EMPLEADODISTRIBUCION_H 1

#include "Asalariado.h"

class EmpleadoDistribucion: public Asalariado{
  //Atributos de instancia
  private:
        char region [10];
  public:
  //Constructor
  EmpleadoDistribucion(char[], long, int, char[]);
  //Métodos de instancia:
  char * getRegion ();
  void setRegion (char[]);
};

#endif

//Fichero “EmpleadoDistribucion.cpp”

#include <cstring>
#include "EmpleadoDistribucion.h"

EmpleadoDistribucion::EmpleadoDistribucion(char nombre[], long dni, int
diasVacaciones, char region[]): Asalariado (nombre, dni, diasVacaciones){
   strcpy (this->region, region);
};

char * EmpleadoDistribucion::getRegion (){
    return this->region;
};

void EmpleadoDistribucion::setRegion (char nueva_region[]){
    strcpy (this->region, nueva_region);
};
```

Como se puede observar, en los ficheros anteriores no hemos definido una función “main” que nos permita trabajar con las clases programadas. Comentaremos primero algunas características de los mismos:

1. En primer lugar, podemos observar cómo en la declaración de las clases derivadas o subclases (“EmpleadoProduccion.h” y “EmpleadoDistribucion.h”)

hemos debido incluir el fichero correspondiente a la declaración de la clase base (“Asalariado.h”).

2. También se puede observar cómo la forma de declarar en C++ que una clase hereda de otra es por medio del uso de la orden

```
class EmpleadoProduccion: public Asalariado {
        ...
}
```

En este caso, los dos puntos, “:”, indican que la clase “EmpleadoProduccion” se define por herencia de la clase “Asalariado”. El comando “public” indica que los elementos (atributos y métodos) que sean “public” o “protected” (modificador de acceso que introduciremos en la Sección 2.7) en la clase base, mantendrán ese mismo modificador de acceso en la clase derivada (los elementos que sean “private” en la clase base no serán accesibles fuera de la clase base, por tanto, ni siquiera en la clase derivada).

También podríamos haber declarado la herencia como “protected” (todos los elementos de la clase base que fueran “public” o “protected” habrían pasado a ser “protected” en la clase derivada), o incluso “private”, haciendo que todos los elementos de la clase base pasaran a ser privados en la clase derivada.

En general, salvo que se especifique lo contrario, siempre preferiremos que los elementos heredados conserven su modificador de acceso en la clase derivada, para lo cual definiremos la herencia como “public”.

3. La siguiente peculiaridad que se puede observar en los ficheros de cabeceras “EmpleadoProduccion.h” y “EmpleadoDistribucion.h” es que en los mismos no hemos declarado ninguno de los métodos que encontrábamos en la clase base. Dichos métodos han sido importados a la misma por medio de la declaración de la relación de herencia. Aquí podemos ver ya un primer ejemplo de cómo las relaciones de herencia permiten la reutilización de código: por medio de la declaración de herencia, todos los atributos y métodos de la clase base son incluidos en la clase derivada.

4. Pasemos ahora a los archivos de definición de las clases, “EmpleadoProduccion.cpp” y “EmpleadoDistribucion.cpp”. La principal diferencia que debemos observar en los mismos está en las definiciónes de los constructores. Éstas son:

```
EmpleadoProduccion::EmpleadoProduccion(char nombre[], long dni, int diasVacaciones, char turno[]): Asalariado (nombre, dni, diasVacaciones){
  strcpy (this->turno, turno);
};

EmpleadoDistribucion::EmpleadoDistribucion(char nombre[], long dni, int diasVacaciones, char turno[]): Asalariado (nombre, dni, diasVacaciones){
  strcpy (this->region, region);
};
```

Como se puede observar, tras la cabecera del constructor, el primer comando que encontramos es “: Asalariado (nombre, dni, diasVacaciones)”.

Por medio de dicho comando lo que conseguimos es que el constructor de la clase base, “Asalariado” sea invocado con los parámetros que se especifique a continuación, “nombre”, dni”, “diasVacaciones”. De esta forma, podemos reutilizar el código de dicho constructor, inicializando el valor de los atributos de la clase “Asalariado”. En los constructores de las clases “EmpleadoProduccion” y “EmpleadoDistribucion” sólo queda por inicializar los atributos “turno” y “region” correspondientes.

En C++ (y también en Java) es obligatorio que el constructor de una clase derivada invoque a un constructor de la clase base correspondiente. Si existe un constructor por defecto (sin parámetros), dicha llamada se hará de forma transparente al usuario. En caso contrario, el usuario deberá invocar a alguno de los constructores de la clase base.

5. Un último comentario que podemos añadir es de nuevo que todos los métodos que se encontraban en la clase base no han debido ser definidos ni declarados en la clase derivada, por lo cual hemos reutilizado todo el código de la misma. Las dos herencias que hemos definido son casos de lo que antes definíamos como “hijos buenos”.

Veamos ahora un sencillo ejemplo de programa principal que actúa como cliente de la anterior definición de clases:

```
#include <cstdlib>
#include <iostream>

#include "Asalariado.h"
#include "EmpleadoProduccion.h"
#include "EmpleadoDistribucion.h"

using namespace std;

int main (){

 Asalariado emplead1 ("Manuel Cortina", 12345678, 28);
 EmpleadoProduccion * emplead2 = new EmpleadoProduccion ("Juan Mota", 55333222, 30, "noche");
 EmpleadoDistribucion * emplead3 = new EmpleadoDistribucion ("Antonio Camino", 55333666, 35, "Granada");

 cout << "El nombre del emplead1 es " << emplead1.getNombre() << endl;
 cout << "El nombre del emplead2 es " << emplead2->getNombre() << endl;
 cout << "El turno del emplead2 es " << emplead2->getTurno() << endl;
 cout << "El nombre del emplead3 es " << emplead3->getNombre() << endl;
 cout << "La region del emplead3 es " << emplead3->getRegion() << endl;
```

```
 system ("PAUSE");
 return 0;
}
```

Como podemos observar, en los objetos de la clase “EmpleadoProduccion” podemos hacer uso de los métodos propios de la clase “Asalariado” (como “getNombre()”) con idéntico resultado, así como de sus propios métodos (como “getTurno()”).

Veamos ahora el mismo ejemplo en Java.

2.4.3 DECLARACIÓN Y DEFINICIÓN DE RELACIONES DE HERENCIA EN JAVA

```
//Fichero “Asalariado.java”

public class Asalariado{

	private String nombre;
	private long dni;
	private int diasVacaciones;

	public Asalariado(String nombre, long dni, int diasVacaciones){
		this.nombre = nombre;
		this.dni = dni;
		this.diasVacaciones = diasVacaciones;
	}

	public String getNombre (){
		return this.nombre;
	}

	public void setNombre (String nuevo_nombre){
		this.nombre = nuevo_nombre;
	}

	public long getDni (){
		return this.dni;
	}

	public void setDni (long nuevo_dni){
		this.dni = nuevo_dni;
	}

	public int getDiasVacaciones (){
		return this.diasVacaciones;
	}

	public void setDiasVacaciones (int nuevo_diasVacaciones){
		this.diasVacaciones = nuevo_diasVacaciones;
```

```
        }
}

//Fichero “EmpleadoProduccion.java”

public class EmpleadoProduccion extends Asalariado{

        private String turno;

        public EmpleadoProduccion (String nombre, long dni, int
diasVacaciones, String turno){
                super (nombre, dni, diasVacaciones);
                this.turno = turno;
        }

        public String getTurno (){
                return this.turno;
        }

        public void setTurno (String nuevo_turno){
                this.turno = nuevo_turno;
        }
}

//Fichero “EmpleadoDistribucion.java”

public class EmpleadoDistribucion extends Asalariado{

        private String region;

        public EmpleadoDistribucion (String nombre, long dni, int
diasVacaciones, String region){
                super (nombre, dni, diasVacaciones);
                this.region = region;
        }

        public String getRegion (){
                return this.region;
        }

        public void setRegion (String nueva_region){
                this.region = nueva_region;
        }
}
```

Veamos algunas características de los mismos:

1. En primer lugar, podemos observar cómo en Java podemos evitar los “#include” propios de C++. La visibilidad de las clases depende únicamente de

su modificador de acceso (en este caso “public” las tres) y del “package” en el que se encuentren las mismas (podemos asumir que en el mismo).

2. En segundo lugar, la declaración de relaciones de herencia en Java se hace por medio del comando “extends”. Así, las cabeceras de las clases “EmpleadoProduccion” y “EmpleadoDistribucion” quedan:

```
public class EmpleadoDistribucion extends Asalariado{...}

public class EmpleadoProduccion extends Asalariado{...}
```

Por medio del comando “extends” importamos a nuestra clase todos los atributos y métodos propios de la clase base (“Asalariado” en este caso). A diferencia de C++, la herencia en Java no permite modificar la visibilidad de los atributos de la clase base. Es decir, los elementos que sean “public” en la clase base seguirán siendo tal en la clase derivada, los elementos que sean “protected” en la clase base serán “protected” en la clase derivada (igual con el modificador por defecto “package”), y los elementos que sean “private” en la clase base no serán visibles en la clase derivada.

3. Sobre la definición de métodos en las clases derivadas, lo primero reseñable es que, al igual que en C++, en las clases derivadas no debemos declarar ni definir ninguno de los atributos ni métodos que vayamos a heredar (a no ser que cambiemos su comportamiento). En nuestro ejemplo, todos los atributos y métodos preservan su comportamiento, y no requieren ser declarados ni definidos de nuevo.

4. Con respecto a la definición de los constructores, podemos ver como los mismo hacen uso de un comando que hasta ahora no conocíamos:

```
public EmpleadoProduccion (String nombre, long dni, int diasVacaciones, String turno){
        super (nombre, dni, diasVacaciones);
        this.turno = turno;
    }

public EmpleadoDistribucion (String nombre, long dni, int diasVacaciones, String region){
        super (nombre, dni, diasVacaciones);
        this.region = region;
    }
```

En ambos casos, el método “super(_,_,_)” nos ha permitido llamar al constructor de la clase base (es decir, al método “Asalariado (String, long, int)”) con los parámetros que le debíamos pasar a dicho constructor.

En Java, al igual que en C++, un constructor de una clase derivada siempre debe invocar a un constructor de la clase base. Si existe un constructor de la clase base por defecto (sin parámetros), éste será invocado de forma transparente al usuario, a no ser que el usuario invoque a alguno de los otros

constructores de la clase base. Si no existe constructor por defecto, el programador de la clase derivada debe invocar a alguno de los de la clase base (además, esta invocación debe ser el primer comando del constructor, en caso contrario, se obtendrá un error de compilación).

Este requisito que nos imponen tanto Java como C++ nos permite acceder y poder inicializar los atributos de la clase base que sean privados (private), ya que de no ser así no habría forma de poder acceder a los mismos (al menos directamente).

```
public class Principal_EjemploAsalariado{

        public static void main (String [] args){

                Asalariado emplead1 = new Asalariado ("Manuel Cortina", 12345678, 28);
                EmpleadoProduccion emplead2 = new EmpleadoProduccion ("Juan Mota", 55333222, 30, "noche");
                EmpleadoDistribucion emplead3 = new EmpleadoDistribucion ("Antonio Camino", 55333666, 35, "Granada");

                System.out.println ("El nombre del emplead1 es " + emplead1.getNombre());
                System.out.println ("El nombre del emplead2 es " + emplead2.getNombre());
                System.out.println ("El turno del emplead2 es " + emplead2.getTurno());
                System.out.println ("El nombre del emplead3 es " + emplead3.getNombre());
                System.out.println ("La region del emplead3 es " + emplead3.getRegion());
        }
}
```

Podemos ver cómo el programa principal no tiene ninguna característica particular. Los objetos de las clases derivadas pueden hacer uso de los métodos de la clase base, así como de los suyos propios.

2.5 VENTAJAS DEL USO DE RELACIONES DE HERENCIA: REUTILIZACIÓN DE CÓDIGO Y POLIMORFISMO DE TIPOS DE DATOS

Hay múltiples ventajas en el uso de las relaciones de herencia en lenguajes de POO. Las dos que vamos a destacar en esta Sección son la reutilización de código y la posibilidad de definir estructuras de datos genéricas.

2.5.1 REUTILIZACIÓN DE CÓDIGO

Por medio de la definición de relaciones de herencia entre clases, conseguimos que todos los atributos y métodos de una clase (la clase base) pasen a formar parte de otra clase (la clase derivada). Esto hace que todo el código que

deberíamos haber incluido relativo a esos atributos y métodos no tenga que ser escrito de nuevo.

Aparte de liberarnos de la tarea de reescribir dicho código, el haber definido una relación de herencia entre dos clases nos libera también de la tarea de tener que mantener el código en ambas. Todos los cambios que hagamos en la clase base serán directamente trasladados a la clase derivada por el enlace entre ambas que ha creado la relación de herencia. De este modo, el programador queda liberado de la tarea de mantener simultáneamente el código en ambas clases.

A modo de ejemplo, se puede tratar de comprobar el número de líneas que hemos salvado en los sencillos ejemplos anteriores en Java y C++.

Como norma general tanto para C++ como para Java, en las relaciones de herencia no deben ser redeclarados o redefinidos ni los atributos de la clase base, ni todos aquellos métodos de la misma que no vayan a ser redefinidos (hablaremos de redefinición de métodos en la Sección 2.6).

2.5.2 POSIBILIDAD DE DEFINICIÓN DE ESTRUCTURAS GENÉRICAS

Una segunda ventaja de declarar jerarquías de clases en nuestros programas es que luego podemos hacer uso de ellas para definir estructuras de datos que contengan objetos de distintas clases. La única condición que debemos observar es que la estructura de datos, si es definida de una clase “A”, sólo puede incluir objetos que pertenezcan a clases que hayan sido definidas por especialización de la clase “A” (o subclases de la misma). La estructura genérica podrá contener objetos de la clase “A”, o de cualquiera de sus “subtipos”.

Para ilustrar esta característica presentaremos un sencillo ejemplo que hace uso de las clases presentadas en la Sección anterior.

Supongamos que a partir de las clases anteriores “Asalariado”, “EmpleadoProduccion” y “EmpleadoDistribucion” queremos definir una estructura de datos que nos permita alojar objetos tanto de la clase “EmpleadoProduccion” como de la clase “EmpleadoDistribucion” (por ejemplo, un “array” que contenga a todos los empleados de la fábrica).

Para ello, tanto los objetos de la clase “EmpleadoProduccion” como los de la clase “EmpleadoDistribucion” deberían compartir un (súper)tipo común. En nuestro caso, gracias a haber definido la clase base “Asalariado”, de la que heredan tanto “EmpleadoDistribucion” como “EmpleadoProduccion”, podemos realizar la siguiente operación (también en C++ es posible):

```
public static void main (String [] args){

        Asalariado emplead1 = new Asalariado ("Manuel Cortina", 12345678,
28);
```

```
        EmpleadoProduccion emplead2 = new EmpleadoProduccion ("Juan
Mota", 55333222, 30, "noche");
        EmpleadoDistribucion emplead3 = new EmpleadoDistribucion ("Antonio
Camino", 55333666, 35, "Granada");

        Asalariado [] array_asal = new Asalariado [3];

        for (int i = 0; i < 3; i++){
                array_asal [i]= new Asalariado ("", 0, 0);
        }
        array_asal [0] = emplead1;
        array_asal [1] = emplead2;
        array_asal [2] = emplead3;
}
```

Como se puede observar, los objetos de las clases “EmpleadoProduccion” y “EmpleadoDistribucion”, en particular, son también objetos de la clase “Asalariado” (no en vano, tienen todos los atributos y métodos de la misma). También se puede decir que entre las clases “EmpleadoProduccion” y “EmpleadoDistribucion” y la clase “Asalariado” existe una relación de subtipado. Por medio de esta característica, hemos conseguido construir una estructura genérica (un “array” de objetos de la clase “Asalariado”) que nos ha permitido agrupar en la misma objetos de las tres clases.

La situación anterior se puede hacer todavía más patente por medio de la siguiente declaración de los mismos objetos:

```
public static void main (String [] args){

        Asalariado emplead1 = new Asalariado ("Manuel Cortina", 12345678,
28);
        Asalariado emplead2 = new EmpleadoProduccion ("Juan Mota",
55333222, 30, "noche");
        Asalariado emplead3 = new EmpleadoDistribucion ("Antonio Camino",
55333666, 35, "Granada");

        Asalariado [] array_asal = new Asalariado [3];

        for (int i = 0; i < 3; i++){
                array_asal [i]= new Asalariado ("", 0, 0);
        }
        array_asal [0] = emplead1;
        array_asal [1] = emplead2;
        array_asal [2] = emplead3;
}
```

Hemos declarado los tres objetos de tipo “Asalariado” y los hemos construido, uno de la clase “Asalariado”, y los otros de las clases “EmpleadoProduccion” y “EmpleadoDistribucion”.

Sin embargo, el haber declarado los tres objetos de tipo “Asalariado” provoca que, ahora, no podamos realizar la siguiente operación:

emplead3.getRegion ();

Si realizamos esta operación (tanto en Java como en C++), el compilador nos devolverá un error, avisándonos de que el objeto “emplead3” ha sido declarado de tipo “Asalariado” (aunque lo hayamos construido por medio del constructor de la clase “EmpleadoDistribucion”), y por tanto no dispone de los métodos “setRegion(String): void”, “getRegion (): String”, ni, por supuesto, de los métodos “setTurno(String): void” o getTurno (): String“.

Lo mismo nos pasaría con las componentes (objetos) del “array” “array_asal” que hemos definido en el ejemplo previo. Si bien podemos hacer uso del mismo para alojar objetos de distintos tipos, sólo podremos utilizar los métodos correspondientes a la clase de la cual se ha definido la estructura genérica (en este caso, “Asalariado”). Se puede observar la importancia de distinguir entre la declaración y la definición de un objeto, y la relevancia que puede llegar a tener declarar un determinado objeto de una clase u otra.

En el Tema 3 recuperaremos las estructuras genéricas para ilustrar ejemplos de polimorfismo.

2.6 REDEFINICIÓN DE MÉTODOS EN CLASES HEREDADAS

Hasta ahora, los casos que hemos visto de herencia entre clases han tenido un comportamiento similar. Todos los atributos que están presentes en la clase base lo estaban también en la clase derivada, y los métodos declarados en la clase base se transferían a la clase derivada conservando tanto su declaración (la cabecera) como la definición (o lo que llamaríamos el cuerpo del método). Esta situación es lo que hemos definido como “hijos buenos” (siempre mantienen el comportamiento de su clase base).

Sin embargo, es fácil encontrar ejemplos en los que lo anterior no es cierto, y una clase que mantiene los atributos y la declaración de los métodos de una clase base debe modificar el comportamiento de alguno de los métodos de la misma. Esto es lo que llamaremos un “mal hijo” (es decir, el que se “comporta” de forma distinta a como lo hace su clase base). La mayor parte de los lenguajes de POO admiten esta situación (en particular, C++ y Java lo hacen).

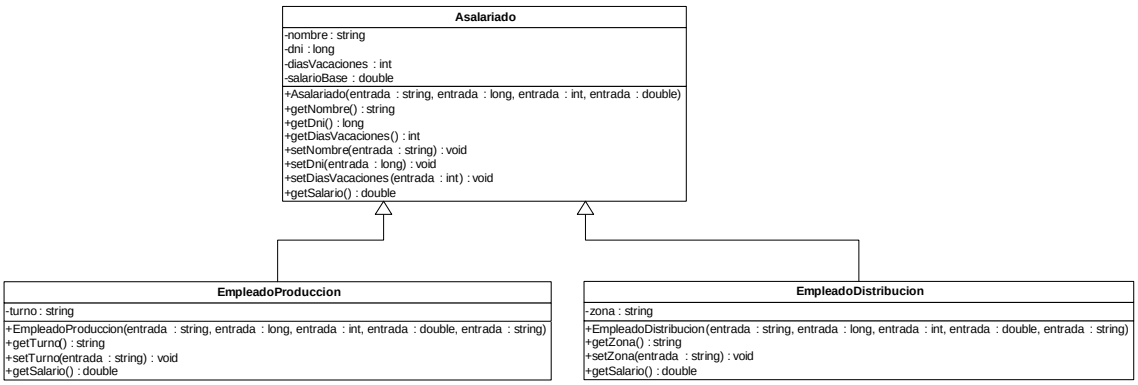
Si tenemos dos clases entre las cuales hay una relación de herencia, de tal forma que la clase derivada modifica el comportamiento (o la definición) de alguno de los métodos de la clase base, diremos que dicho método está *redefinido* en la clase derivada.

La redefinición de métodos tiene importantes consecuencias desde el punto de vista del sistema de tipos de los lenguajes de programación, ya que el compilador debe resolver en tiempo de ejecución (y no de compilación, caso que se conoce como enlazado estático) la definición concreta de los métodos

que debe utilizar. Esto es lo que se conoce como enlazado dinámico, y permite implementar el *polimorfismo*, que veremos en el Tema 3.

Veamos ahora algún ejemplo de redefinición de métodos. Imaginemos que, sobre el ejemplo anterior con la clase “Asalariado”, debemos introducir un nuevo método que permita calcular la nómina de un empleado. Todos los asalariados de la firma tienen un atributo representando su salario base. Los empleados de producción reciben sobre esta cantidad un incremento del 15%. Los empleados de distribución, sin embargo, reciben un aumento de sólo el 10%.

Veamos con la anterior especificación cómo quedaría el diagrama de clases en UML.



En primer lugar, observa que aquellos métodos que se redefinen (en este caso, “getSalario (): double”) sí deben aparecer en las clases derivadas. Es la forma que tiene UML de distinguir entre métodos que van a ser redefinidos y aquéllos que no.

Veamos ahora cómo podemos programar tal método (“getSalario (): double”) en las distintas clases del diagrama. Observa, en primer lugar, que por ser “salarioBase:double” privada (o “private”) en la clase “Asalariado” su valor no va a ser visible en las clases “EmpleadoProduccion” y “EmpleadoDistribucion”. Esto quiere decir que deberemos acceder a este valor por medio del método “getSalario (): double” de la clase “Asalariado”.

Observamos primero la solución en C++:

```
//Fichero “Asalariado.h”

#ifndef ASALARIADO_H
#define ASALARIADO_H 1

class Asalariado{
  //Atributos de instancia
  private:
        char nombre [30];
        long dni;
        int diasVacaciones;
```

```
        double salarioBase;
  public:
  //Constructor
  Asalariado(char[], long, int, double);
  //Métodos de instancia:
  char * getNombre ();
  void setNombre (char[]);
  long getDni ();
  void setDni (long);
  int getDiasVacaciones ();
  void setDiasVacaciones (int);
  double getSalario ();

};

#endif

//Fichero “Asalariado.cpp”

#include <cstring>
#include "Asalariado.h"

using namespace std;

Asalariado::Asalariado(char nombre[], long dni, int diasVacaciones, double
salarioBase){
   strcpy (this->nombre, nombre);
   this->dni = dni;
   this->diasVacaciones = diasVacaciones;
   this->salarioBase = salarioBase;
};

char * Asalariado::getNombre (){
    return this->nombre;
};

void Asalariado::setNombre (char nuevo_nombre[]){
    strcpy (this->nombre, nuevo_nombre);
};

long Asalariado::getDni (){
    return this->dni;
};

void Asalariado::setDni (long nuevo_dni){
    this->dni = nuevo_dni;
};

int Asalariado::getDiasVacaciones (){
    return this->diasVacaciones;
```

```
};

void Asalariado::setDiasVacaciones (int nuevo_diasVacaciones){
    this->diasVacaciones = nuevo_diasVacaciones;
};

double Asalariado::getSalario (){
      return this->salarioBase;
};

//Fichero “EmpleadoProduccion.h”

#ifndef EMPLEADOPRODUCCION_H
#define EMPLEADOPRODUCCION_H 1

#include "Asalariado.h"

class EmpleadoProduccion: public Asalariado{
     //Atributos de instancia
     private:
            char turno [10];
     public:
     //Constructor
     EmpleadoProduccion(char[], long, int, double, char[]);
     //Métodos de instancia:
     char * getTurno ();
     void setTurno (char[]);
     double getSalario ();
};

#endif

//Fichero “EmpleadoProduccion.cpp”

#include <cstring>
#include "EmpleadoProduccion.h"

EmpleadoProduccion::EmpleadoProduccion(char nombre[], long dni, int
diasVacaciones, double salarioBase, char turno[]):Asalariado(nombre, dni,
diasVacaciones, salarioBase){
    strcpy (this->turno, turno);
};

char * EmpleadoProduccion::getTurno (){
    return this->turno;
};

void EmpleadoProduccion::setTurno (char nuevo_turno[]){
    strcpy (this->turno, nuevo_turno);
};
```

```
double EmpleadoProduccion::getSalario (){
      return Asalariado::getSalario() * (1 + 0.15);
};

//Fichero “EmpleadoDistribucion.h”

#ifndef EMPLEADODISTRIBUCION_H
#define EMPLEADODISTRIBUCION_H 1

#include "Asalariado.h"

class EmpleadoDistribucion: public Asalariado{
  //Atributos de instancia
  private:
         char region [10];
  public:
  //Constructor
  EmpleadoDistribucion(char[], long, int, double, char[]);
  //Métodos de instancia:
  char * getRegion ();
  void setRegion (char[]);
  double getSalario ();
};

#endif

//Fichero “EmpleadoDistribucion.cpp”

#include <cstring>
#include "EmpleadoDistribucion.h"

EmpleadoDistribucion::EmpleadoDistribucion(char nombre[], long dni, int
diasVacaciones, double salarioBase, char region[]): Asalariado (nombre, dni,
diasVacaciones, salarioBase){
   strcpy (this->region, region);
};

char * EmpleadoDistribucion::getRegion (){
     return this->region;
};

void EmpleadoDistribucion::setRegion (char nueva_region[]){
     strcpy (this->region, nueva_region);
};

double EmpleadoDistribucion::getSalario (){
      return Asalariado::getSalario() * (1 + 0.10);
};
```

Notas:

1. Observa cómo, al igual que en el diagrama UML, hemos tenido que declarar el método “getSalario(): double” en todas y cada una de las clases que debemos redefinirlo (en este caso, en “EmpleadoDistribucion.h” y “EmpleadoProduccion.h”). Es la manera de avisar al compilador de que ese método va a ser redefinido en las clases “EmpleadoDistribucion” y “EmpleadoProduccion”.

2. Con respecto a la definición del método en las clases “EmpleadoDistribucion” y “EmpleadoProduccion”, es importante observar que hemos tenido que acceder al método “Asalariado::getSalario (): double”, propio de la clase “Asalariado” (ya que “salarioBase: double” es de tipo privado) y cómo hemos tenido que acceder al mismo:

```
double EmpleadoProduccion::getSalario (){
      return Asalariado::getSalario() * (1 + 0.15);
};

double EmpleadoDistribucion::getSalario (){
      return Asalariado::getSalario() * (1 + 0.10);
};
```

La forma de acceder a un método de la clase base en C++ (y, en general, de cualquier clase externa) es por medio de la sintaxis:

NombreClase::nombreMetodo ()

A partir del valor devuelto por la invocación al método “Asalariado::getSalario()” y de la correspondiente operación (multiplicar por “1 + 0.15” o por “1 + 0.10”) somos capaces de definir el nuevo comportamiento que debe tener el método “getSalario(): double” para las clases “EmpleadoDistribucion” y “EmpleadoProduccion”.

Veamos ahora si la anterior redefinición de métodos ha surtido efecto con un programa cliente que haga uso de dicho método:

```
#include <cstdlib>
#include <iostream>

#include "Asalariado.h"
#include "EmpleadoProduccion.h"
#include "EmpleadoDistribucion.h"

using namespace std;

int main (){

 Asalariado emplead1 ("Manuel Cortina", 12345678, 28, 1200);
```

```
 EmpleadoProduccion * emplead2 = new EmpleadoProduccion ("Juan Mota", 55333222, 30, 1200, "noche");
 EmpleadoDistribucion * emplead3 = new EmpleadoDistribucion ("Antonio Camino", 55333666, 35, 1200, "Granada");

 cout << "El nombre del emplead1 es " << emplead1.getNombre() << endl;
 cout << "Su salario es " << emplead1.getSalario() << endl;
 cout << "El nombre del emplead2 es " << emplead2->getNombre() << endl;
 cout << "El turno del emplead2 es " << emplead2->getTurno() << endl;
 cout << "Su salario es " << emplead2->getSalario() << endl;
 cout << "El nombre del emplead3 es " << emplead3->getNombre() << endl;
 cout << "La region del emplead3 es " << emplead3->getRegion() << endl;
 cout << "Su salario es " << emplead3->getSalario() << endl;

 system ("PAUSE");
 return 0;
}
```

El resultado de ejecutar el código anterior es:

```
El nombre del emplead1 es Manuel Cortina
Su salario es 1200
El nombre del emplead2 es Juan Mota
El turno del emplead2 es noche
Su salario es 1380
El nombre del emplead3 es Antonio Camino
La region del emplead3 es Granada
Su salario es 1320
```

Como podemos ver, cada objeto ha sido capaz de hacer uso de la definición del método “getSalario(): double” correspondiente a la clase de la que había sido declarado.

Realizamos ahora una pequeña modificación sobre el programa “main” anterior, y declaramos todos los objetos como de la clase “Asalariado”:

```
#include <cstdlib>
#include <iostream>

#include "Asalariado.h"
#include "EmpleadoProduccion.h"
#include "EmpleadoDistribucion.h"

using namespace std;

int main (){

 Asalariado emplead1 ("Manuel Cortina", 12345678, 28, 1200);
 Asalariado * emplead2 = new EmpleadoProduccion ("Juan Mota", 55333222, 30, "noche", 1200);
```

```
 Asalariado * emplead3 = new EmpleadoDistribucion ("Antonio Camino", 55333666, 35, "Granada", 1200);

 cout << "El nombre del emplead1 es " << emplead1.getNombre() << endl;
 cout << "Su salario es " << emplead1.getSalario() << endl;
 cout << "El nombre del emplead2 es " << emplead2->getNombre() << endl;
 cout << "Su salario es " << emplead2->getSalario() << endl;
 cout << "El nombre del emplead3 es " << emplead3->getNombre() << endl;
 cout << "Su salario es " << emplead3->getSalario() << endl;

 system ("PAUSE");
 return 0;
}
```

Nos podemos formular ahora las siguientes preguntas: ¿Qué método “getSalario(): double” será invocado en cada caso? ¿Cuál debería haber sido invocado?

El resultado de la ejecución del anterior fragmento de código es:

```
El nombre del emplead1 es Manuel Cortina
Su salario es 1200
El nombre del emplead2 es Juan Mota
Su salario es 1200
El nombre del emplead3 es Antonio Camino
Su salario es 1200
```

Vemos cómo ahora el programa le ha asignado a los tres objetos el método “getSalario (): double” propio de la clase “Asalariado”, que es la clase de la que hemos declarado los tres objetos. Aquí puedes observar cómo el enlazado ha sido estático. Cuando hemos declarado los objetos (cuando el programa ha sido compilado), los distintos métodos han recibido la definición propia de la clase “Asalariado”, y posteriormente han mantenido tal comportamiento.

Veremos una forma de resolver el problema anterior en el Tema 3.

Veamos ahora cómo podemos implementar la redefinición de métodos en Java:

```
//Fichero “Asalariado.java”

public class Asalariado{

        private String nombre;
        private long dni;
        private int diasVacaciones;
        private double salarioBase;

        public Asalariado(String nombre, long dni, int diasVacaciones, double salarioBase){
                this.nombre = nombre;
```

```
		this.dni = dni;
		this.diasVacaciones = diasVacaciones;
		this.salarioBase = salarioBase;
	}

	public String getNombre (){
		return this.nombre;
	}

	public void setNombre (String nuevo_nombre){
		this.nombre = nuevo_nombre;
	}

	public long getDni (){
		return this.dni;
	}

	public void setDni (long nuevo_dni){
		this.dni = nuevo_dni;
	}

	public int getDiasVacaciones (){
		 return this.diasVacaciones;
	}

	public void setDiasVacaciones (int nuevo_diasVacaciones){
		this.diasVacaciones = nuevo_diasVacaciones;
	}

	public double getSalario (){
		return this.salarioBase;
	}
}

//Fichero EmpleadoProduccion.java

public class EmpleadoProduccion extends Asalariado{

	private String turno;

	public EmpleadoProduccion (String nombre, long dni, int
diasVacaciones, double salarioBase, String turno){
		super (nombre, dni, diasVacaciones, salarioBase);
		this.turno = turno;
	}

	public String getTurno (){
		return this.turno;
	}
```

```
	public void setTurno (String nuevo_turno){
		this.turno = nuevo_turno;
	}

	public double getSalario (){
		return super.getSalario () * (1 + 0.15);
	}
}

//Fichero EmpleadoDistribucion.java

public class EmpleadoDistribucion extends Asalariado{

	private String region;

	public EmpleadoDistribucion (String nombre, long dni, int
diasVacaciones, double salarioBase, String region){
		super (nombre, dni, diasVacaciones, salarioBase);
		this.region = region;
	}

	public String getRegion (){
		return this.region;
	}

	public void setRegion (String nueva_region){
		this.region = nueva_region;
	}

	public double getSalario (){
		return super.getSalario () * (1 + 0.10);
	}
}
```

Antes de pasar a presentar el programa cliente de la anterior implementación de las clases dadas, veamos algunas de sus propiedades:

1. Al igual que en C++ (y en UML), los métodos que van a ser redefinidos (en este caso, “getSalario (): double”) tienen que ser declarados en todas y cada una de las clases en las que se van a redefinir. Por ese motivo podemos encontrar la declaración del método “getSalario (): double” tanto en la clase “EmpleadoDistribucion” como en la clase “EmpleadoProduccion”.

2. Como además, en Java, definición y declaración de métodos van unidas, podemos observar también cómo hemos redefinido los mismos en las clases derivadas:

Clase EmpleadoProduccion:

```
	public double getSalario (){
```

```
        return super.getSalario () * (1 + 0.15);
    }
```

Clase EmpleadoDistribucion:

```
    public double getSalario (){
        return super.getSalario () * (1 + 0.10);
    }
```

Lo más importante que se puede observar en la definición anterior es cómo hemos tenido que acceder al método “getSalario(): double” de la clase base. El acceso al mismo se hace por medio de la palabra reservada “super”, al igual que en el acceso al constructor de clases base, seguida del método que invocamos (a diferencia de C++, donde el acceso se hacía por medio de la sintaxis “NombreClase::nombreMetodo();”):

```
        super.nombreMetodo()
```

El resto de la definición no presenta ninguna particularidad.

```
//Fichero “Principal_EjemploAsalariado.java”

public class Principal_EjemploAsalariado{

    public static void main (String [] args){

        Asalariado emplead1 = new Asalariado ("Manuel Cortina", 12345678, 28, 1200);
        EmpleadoProduccion emplead2 = new EmpleadoProduccion ("Juan Mota", 55333222, 30, 1200, "noche");
        EmpleadoDistribucion emplead3 = new EmpleadoDistribucion ("Antonio Camino", 55333666, 35, 1200, "Granada");

        System.out.println ("El nombre del emplead1 es " + emplead1.getNombre());
        System.out.println ("El sueldo del emplead1 es " + emplead1.getSalario());
        System.out.println ("El nombre del emplead2 es " + emplead2.getNombre());
        System.out.println ("El turno del emplead2 es " + emplead2.getTurno());
        System.out.println ("El sueldo del emplead2 es " + emplead2.getSalario());
        System.out.println ("El nombre del emplead3 es " + emplead3.getNombre());
        System.out.println ("La region del emplead3 es " + emplead3.getRegion());
        System.out.println ("El sueldo del emplead3 es " + emplead3.getSalario());
```

```
    }
}
```

El resultado de la ejecución del código anterior sería:

```
El nombre del emplead1 es Manuel Cortina
El sueldo del emplead1 es 1200.0
El nombre del emplead2 es Juan Mota
El turno del emplead2 es noche
El sueldo del emplead2 es 1380.0
El nombre del emplead3 es Antonio Camino
La region del emplead3 es Granada
El sueldo del emplead3 es 1320.0
```

Vemos que en cada uno de los tres casos se ha invocado al método “getSalario(): double” de la clase correspondiente.

Si hacemos una segunda prueba con el programa principal, similar a la que hemos realizado antes en C++:

```
public class Principal_EjemploAsalariado{

        public static void main (String [] args){

                Asalariado emplead1 = new Asalariado ("Manuel Cortina", 12345678, 28, 1200);
                Asalariado emplead2 = new EmpleadoProduccion ("Juan Mota", 55333222, 30, 1200, "noche");
                Asalariado emplead3 = new EmpleadoDistribucion ("Antonio Camino", 55333666, 35, 1200, "Granada");

                System.out.println ("El nombre del emplead1 es " + emplead1.getNombre());
                System.out.println ("El sueldo del emplead1 es " + emplead1.getSalario());
                System.out.println ("El nombre del emplead2 es " + emplead2.getNombre());
                System.out.println ("El sueldo del emplead2 es " + emplead2.getSalario());
                System.out.println ("El nombre del emplead3 es " + emplead3.getNombre());
                System.out.println ("El sueldo del emplead3 es " +
        emplead3.getSalario());
        }
}
```

El resultado de ejecutar dicho programa sería:

```
El nombre del emplead1 es Manuel Cortina
El sueldo del emplead1 es 1200.0
```

El nombre del emplead2 es Juan Mota
El sueldo del emplead2 es 1380.0
El nombre del emplead3 es Antonio Camino
El sueldo del emplead3 es 1320.0
El nombre del emplead1 es Manuel Cortina
El nombre del emplead2 es Juan Mota
El nombre del emplead3 es Antonio Camino

Si comparas el resultado con el obtenido al hacer la misma prueba en C++, verás que éste es distinto.

En C++, debido al enlazado estático, los tres objetos han sido declarados de la clase “Asalariado”, y todos ellos acceden a la definición del método “getSalario(): double” de dicha clase. En Java, sin embargo, gracias al enlazado dinámico, aunque los objetos han sido declarados como del tipo “Asalariado”, en tiempo de ejecución se comprueba a qué definición del método “getSalario(): double” de todas los disponibles tienen que acceder, invocando de esta forma al que corresponde.

A modo de conclusión del ejemplo anterior, hemos podido observar que en Java el enlazado es siempre dinámico, y esto quiere decir que en tiempo de ejecución los tipos de los objetos serán comprobados y las definiciones correspondientes de los métodos buscadas. Sin embargo, en C++, existe la posibilidad de trabajar con enlazado estático (como en el ejemplo que hemos presentado en esta Sección) o de trabajar con enlazado dinámico (utilizando las técnicas que veremos en el Tema 3). En general, se considera que para poder aplicar las técnicas propias de la POO, siempre debe haber enlazado dinámico.

2.7 MODIFICADOR DE USO “PROTEGIDO”: POSIBILIDADES DE USO

A lo largo de este Tema ya hemos mencionado la existencia de un modificador de acceso propio de las relaciones de herencia, llamado “protegido” (o “protected” en C++ y Java, que se representa por medio de “#” en UML).

El significado concreto de este modificador de acceso varía ligeramente entre Java y C++:

En C++, un atributo que sea declarado como “protected” en una clase será visible únicamente en dicha clase y en las subclases de la misma.

En Java, un atributo que sea declarado como “protected” será visible en dicha clase, en el “package” que se encuentre la misma, y en las subclases de la misma.

Como se puede observar, este modificador es más restrictivo que “public” pero menos que “private” (y en Java, menos que “package”).

Con “protected” hemos introducido ya todos los modificadores de acceso disponibles en Java y C++, por lo que podemos realizar las siguientes Tablas

que nos permiten observar mejor los ámbitos de visibilidad de métodos y atributos en ambos lenguajes:

C++:

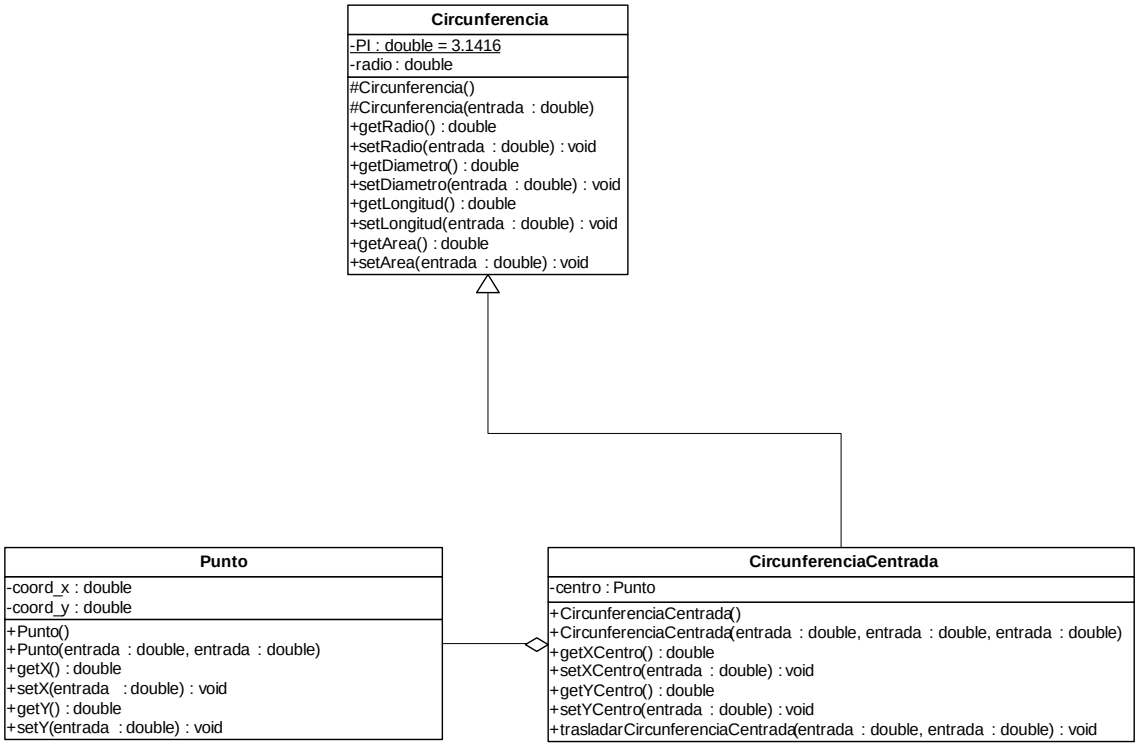
	private	**protected**	**public**
Misma clase	Sí	Sí	Sí
Subclases	No	Sí	Sí
Cualquier clase	No	No	Sí

Java:

	private	package	**Protected**	**public**
Misma clase	Sí	Sí	Sí	Sí
Package	No	Sí	Sí	Sí
Package y subclases	No	No	Sí	Sí
Cualquier clase	No	No	No	Sí

Para ilustrar algunas de las posibilidades de uso del modificador de acceso “protected” recuperaremos algunos de los ejemplos presentados en la Práctica 5 y a lo largo de este Tema.

Comenzamos en primer lugar por el ejemplo sobre la relación de herencia entre las clases “CircunferenciaCentrada” y “Circunferencia”. Lo modificamos de la siguiente forma:

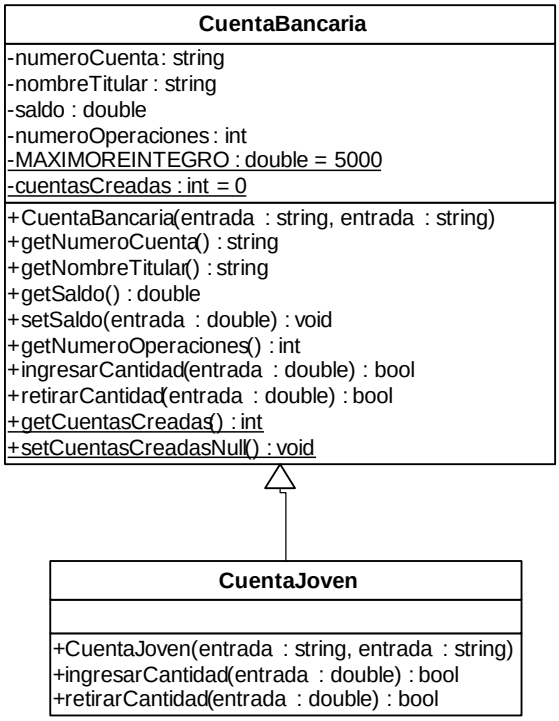


Como se puede observar, hemos declarado los dos constructores de la clase “Circunferencia” como “protected”.

Esto quiere decir que sólo serán accesibles desde las clases que hereden de “Circunferencia” (por ejemplo, “CircunferenciaCentrada”), y, en Java, desde las clases presentes en el mismo “package”. Sin embargo, no serán accesibles desde ninguna otra clase.

De este modo, sólo declarando como “protected” los constructores de la clase “Circunferencia”, ésta se ha convertido en una clase auxiliar que sólo será útil para definir clases por especialización a partir de ella.

Veamos un segundo ejemplo sobre la clase “CuentaBancaria” tal y como la presentamos en la Práctica 5:



Tal y como está definida en el diagrama de clases anterior, la constante de clase “MAXIMOREINTEGRO: double” es de tipo “private” y no es accesible desde fuera de la misma. Esto quiere decir que si pretendemos acceder al valor de MAXIMOREINTEGRO desde la clase “CuentaJoven” obtendremos un error de compilación.

Se pueden proponer varias soluciones a este problema:

1. Una primera solución podría consistir en declarar la constante MAXIMOREINTEGRO como “public”. Esta solución tiene una contrapartida importante, y es que entonces cualquier cliente de la clase tendría visibilidad sobre la misma (aunque no pudiese modificarla ya que la hemos declarado como constante). Además, tampoco presta atención a la noción de encapsulación de la información que enunciamos en el Tema 1 (los atributos de

una clase deben resultar, en general, no visibles a otras clases, para que si tuviéramos que modificarlos dichas clases no se vieran afectadas). Es probable que el programador de la clase no esté satisfecho con dicha solución.

2. Una segunda solución menos agresiva sería declarar dicho atributo como "protected". Esto lo convertiría en visible desde la clase "CuentaJoven" y todas las clases que hereden de la misma, pero no desde los clientes o clases externas a la misma. Esta segunda solución resulta más satisfactoria que la primera, pero va en contra también del principio de encapsulación de la información que enunciamos en el Tema 1.

3. Una tercera solución, que puede solventar los dos problemas que presentaban las anteriores, consistiría en declarar un método de acceso a la constante "MAXIMOREINTEGRO: double", es decir "getMAXIMOREINTEGRO(): double" que fuese de tipo "protected", y mantener "MAXIMOREINTEGRO: double" como "private". De este modo conseguimos que las subclases de "CuentaBancaria" tengan acceso a dicho método, y, además, preservamos el principio de ocultación de la información, haciendo que la constante de clase "MAXIMOREINTEGRO: double" sea sólo visible dentro de la clase en la que ha sido definida.

Con estos dos ejemplos hemos pretendido presentar ciertas posibilidades de uso del modificador "protected". El mismo nos ha servido para declarar "clases auxiliares", que sólo puedan ser útiles en las clases que heredan de la misma, y también para facilitar el acceso a atributos declarados como "private".

En general, se puede decir que el uso del mismo debe estar basado siempre en razones de diseño de nuestro sistema de información, que deberán corresponder con alguno de los principios que presentamos en el Tema 1 (como ocultación de la información, necesidad de acceso y/o modificación de ciertos atributos, ...).

2.8 REPRESENTACIÓN DE RELACIONES DE HERENCIA EN DIAGRAMAS UML

A lo largo del Tema 2 y de la Práctica 5 hemos introducido ya la sintaxis propia de UML para relaciones de herencia. Una referencia donde puedes encontrar más información referente a dicho tema es el libro "El lenguaje unificado de modelado UML", de Grady Booch, James Rumbaugh, Ivar Jacobson, en Addison Wesley (2006).

2.9 PROGRAMACIÓN EN JAVA Y C++ DE RELACIONES DE HERENCIA

Al igual que en la Sección anterior, la sintaxis propia de Java y C++ ha sido introducida a lo largo del Tema 2 en detalle, además con ejemplos que ilustraban su uso. Puedes encontrar también referencias útiles sobre las mismas en los libros "Thinking in C++", de Bruce Eckel y Chuck Allison, en Prentice-Hall (2004) y "Thinking in Java" de Bruce Eckel, en Prentice-Hall (2006).