TEMA 3. DEFINICIÓN Y USO DE MÉTODOS POLIMORFOS

Introducción:

El uso de relaciones de herencia entre clases (Tema 2) junto con la idea de la redefinición de métodos (Sección 2.6) puede tener consecuencias a veces un tanto inesperadas en nuestros programas, como el hecho de que, un mismo método, contenga varias definiciones distintas.

Si a esto le unimos la idea de subtipado junto a la posibilidad de declarar objetos de una clase y construirlos como de cualquiera de sus subclases, la de crear estructuras de datos genéricas (como presentamos en la Sección 2.5.2) que contengan objetos de una clase base y de todas sus subclases, o la de definir métodos y funciones que acepten objetos de una clase y cualquiera de sus subtipos podemos comprender la importancia que puede tener la cuestión anterior.

Las pregunta que trataremos de resolver en este Tema es, cuando invocamos a un método sobre un objeto, ¿qué método está siendo invocado (el de dicha clase o el de alguna de sus superclases)?, ¿cuál debería ser llamado?, ¿podemos influir en cuál ha de ser llamado?.

Las preguntas anteriores están directamente vinculadas a la noción de enlazado estático (“static binding”) o dinámico (“dynamic binding”) en los lenguajes de programación. El “enlazado” es el proceso por el cual la declaración de un método apunta a la definición del mismo. En C++ cada método es “enlazado” con su definición en tiempo de compilación. Por tanto, la declaración del objeto va a determinar la lista de métodos que le van a ser asignados. Este “enlazado estático” permite mejorar el rendimiento de los programas (a todos los objetos se les asignan sus métodos en tiempo de compilación, y nunca van a ser modificados; por tanto, desde el punto de vista de velocidad de ejecución del código resulta óptimo).

Sin embargo, la posibilidad de redefinir métodos junto con el subtipado hace que esta opción no siempre sea la correcta. Si sobre un objeto que hemos declarado de una superclase queremos alojar un objeto de uno de sus subtipos en el cual hemos redefinido un método, este método redefinido debería ser usado. Esto no es posible si el objeto, en tiempo de compilación, ya asignó (de forma inmutable) el método de la superclase.

De ahí la necesidad de un nuevo tipo de enlazado, llamado “enlazado dinámico”, que nos permita enlazar una declaración de un método con la definición del mismo en tiempo de ejecución (de forma dinámica), y decidir si dicha definición corresponde a la superclase o a cualquiera de sus subclases. Este comportamiento se puede obtener en C++, y es el comportamiento por

defecto en Java, y en general en los lenguajes basados en el paradigma de POO.

El Tema estará dividido en las siguientes Secciones. En la Sección 3.1 daremos una definición del polimorfismo y de las ventajas y conveniencia de uso del mismo. En la Sección 3.2 mostraremos diversos ejemplos de uso del polimorfismo en algunos de los ejemplos ya construidos a lo largo del curso. La Sección 3.3 mostrará cómo conseguir polimorfismo en C++, donde deberemos prestar especial atención al uso del modificador "virtual" y de memoria dinámica, y por último en la Sección 3.4 ilustraremos cómo el polimorfismo en Java no requiere de ninguna modificación en nuestros programas por medio de algunos ejemplos.

3.1 DEFINICIÓN DE POLIMORFISMO Y VENTAJAS DE USO

Definición: polimorfismo (en POO) es la capacidad que tienen ciertos lenguajes para hacer que, al enviar el mismo mensaje (o, en otras palabras, invocar al mismo método) desde distintos objetos, cada uno de esos objetos pueda responder a ese mensaje (o a esa invocación) de forma distinta.

El anterior tipo de polimorfismo es conocido como polimorfismo de métodos. Un mismo método puede dar distintas respuestas cuando se le llama a través de distintos objetos. Algunos autores definen otro tipo de polimorfismo, el polimorfismo de objetos, que es el que en la Sección 2.5.2 nos permitía definir estructuras genéricas (que está basado en la idea de que un objeto de una subclase, pertenece, aparte de al tipo correspondiente a dicha clase, al tipo de todas sus clases base).

A lo largo de esta Sección hablaremos de polimorfismo de métodos y nos referiremos a él como polimorfismo, salvo que digamos lo contrario.

Como comentábamos en la introducción del Tema, hay dos ingredientes básicos necesarios para que tenga sentido hablar de polimorfismo de métodos.

1. Que tengamos definidas ciertas relaciones de herencia entre clases
2. Que exista redefinición de algún método

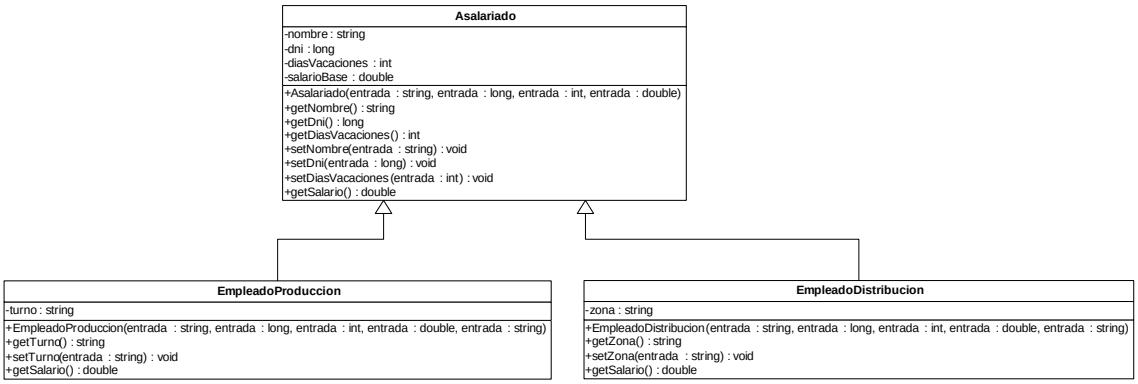
Si se verifican las dos condiciones anteriores, podemos plantear el problema del polimorfismo en los siguientes términos:

Si construimos un objeto como perteneciente a una subclase en la que se redefine un método, y posteriormente tratamos de invocar a dicho método sobre el objeto, ¿cuál de las distintas definiciones del mismo debería ser invocada?, ¿alguna de las de las clases bases del objeto, o la de la clase derivada?

La respuesta natural es que el método invocado debe ser el de la clase sobre la que hemos construido el objeto (independientemente de cómo éste haya sido declarado). Pero este comportamiento, que hemos definido como natural y

que es el que deberíamos esperar en nuestros programas, requiere ciertos características por parte del compilador de nuestros programas.

En particular, observemos el siguiente fragmento de código (suponer que tenemos definidas clases “Asalariado”, “EmpleadoDistribucion” y “EmpleadoProduccion” tal y como las enunciamos en la Sección 2.6 de acuerdo con el siguiente diagrama UML):



Lo más relevante del anterior diagrama UML es que se ha redefinido el método “getSalario(): double” para que se comporte de forma distinta en las clases “Asalariado”, “EmpleadoDistribucion” y “EmpleadoProduccion”.

Si ahora tratamos de ejecutar el siguiente fragmento de código (en Java o en C++):

EmpleadoDistribucion empl1 = new EmpleadoDistribucion (“Antonio”, 11222333, 28, 1400);
Asalariado empl2;

empl2 = empl1;
empl2.getSalario();

La pregunta que nos deberíamos plantear es, ¿qué definición del método “getSalario(): double” le ha sido asignado al objeto “empl2”?

Una primera respuesta que podríamos formular a dicha pregunta es que, como “empl2” ha sido declarado como un objeto de la clase “Asalariado”, entonces utilizará la definición del método “getSalario(): double” de la misma. Pero al cambiar el valor de “empl2” a “empl1”, lo que esperaríamos es que “empl2” pasara a utilizar la definición de “getSalario(): double” de “EmpleadoDistribucion”.

Lo que pretendemos ilustrar es que si, al compilar el programa, a “empl2” ya le asignamos la definición de “getSalario(): double” que debe utilizar (la de la clase “Asalariado”), nuestro programa se estará comportando de forma errónea (o al menos no la deseada por nosotros).

Por tanto, el compilador de nuestro lenguaje de programación debe ser capaz, en tiempo de ejecución, de decidir qué definición del método “getSalario():

double” ha de utilizar, ya que ésta depende del tipo concreto al que pertenezca “empl2” en cada momento del programa.

Esto es lo que se conoce en los lenguajes de programación como “enlazado dinámico” (o “dynamic binding”). Un método es “enlazado” con su definición correspondiente en tiempo de ejecución (de forma dinámica). Otra opción posible es lo que se conoce como “enlazado temprano” (o “early binding”), en la cual un método es enlazado con su definición en tiempo de compilación, y siempre hará uso de la misma definición.

La ventaja de utilizar “enlazado dinámico” es que nos va a permitir hacer uso del polimorfismo de métodos. Un mismo método, en tiempo de ejecución, puede invocar a cualquiera de las definiciones que se han dado del mismo (de ahí el concepto de “polimorfismo de métodos”).

Los compiladores que ofrecen “enlazado temprano” poseen también una ventaja, y es que la velocidad de ejecución de los programas puede ser mayor que realizando “enlazado dinámico”, ya que los métodos (y los tipos de los objetos) se fijan en tiempo de compilación y ya no serán comprobados durante la ejecución del mismo.

Al realizar POO nos decantaremos siempre por el uso de “enlazado dinámico”, ya que el polimorfismo es uno de los requisitos básicos que se le exige a un programa para ser “orientado a objetos”.

Si nuestro compilador dispone de “enlazado dinámico”, la forma en que produce la invocación de un método se puede explicar, de forma intuitiva, como sigue (seguimos ilustrándolo con el ejemplo sobre la clase “Asalariado)”:

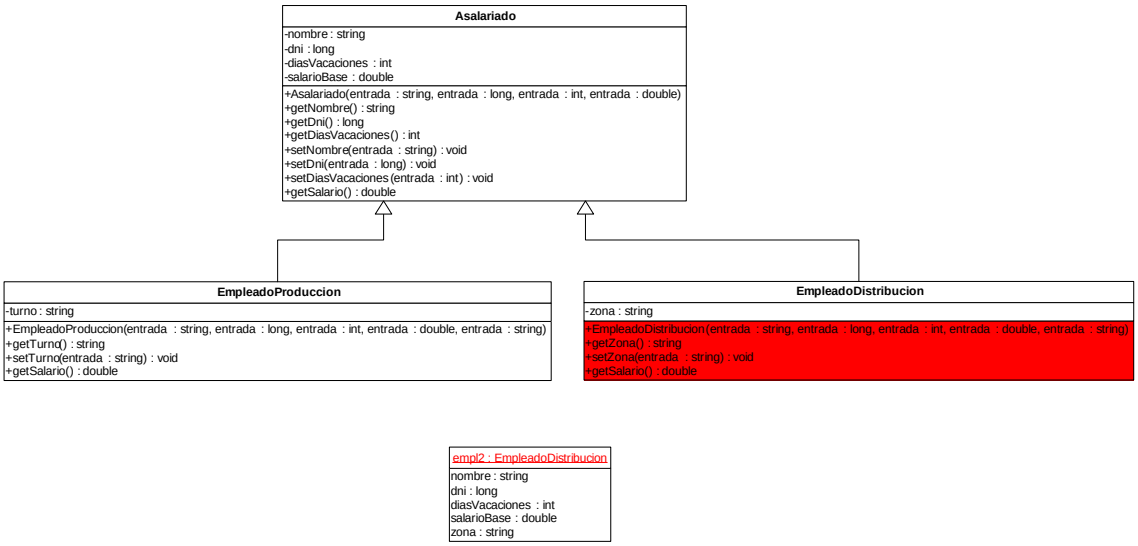
```
EmpleadoDistribucion empl1 = new EmpleadoDistribucion (“Antonio”, 11222333, 28, 1400);
Asalariado empl2;

empl2 = empl1;
empl2.getSalario();
```

Cuando sobre el objeto “empl2” invocamos al método “getSalario(): double”, el compilador comprueba el tipo del objeto “empl2”. En el caso anterior, determinaría que el mismo es de tipo “EmpleadoDistribucion”.

Entonces se dirige a la clase “EmpleadoDistribucion” y verifica si la misma posee una definición propia de dicho método. Si es así, “enlaza” la declaración del método con dicha definición, y lo invoca. Si no es así, sube “un nivel” (a la superclase “Asalariado”) y comprueba si allí está definido dicho método. Este proceso se repite hasta que una definición del método “getSalario(): double” es encontrada.

En este caso, la búsqueda (que va siempre de abajo hacia arriba), termina en la propia clase “EmpleadoDistribucion”, y el compilador “enlaza” “empl2” con la versión en dicha clase de “getSalario(): double”.

El proceso anterior se podría entender más fácilmente sobre el propio diagrama UML:



Cuando invocamos al método “getSalario(): double” sobre el objeto “empl2”, en un compilador de “enlazado dinámico”, en tiempo de ejecución, “empl2” comprueba su tipo (a qué clase pertenece), y el compilador se dirige a dicha clase (“EmpleadoDistribucion”) a buscar una definición del método. Si encuentra dicha definición, como es el caso, la “enlaza” con “empl2” y hace uso de ella.

El compilador así ha evitado hacer uso del método “getSalario(): double” tal y como éste está definido en la clase “Asalariado”.

Si invocamos a un método que no está redefinido en la clase “EmpleadoDistribucion” sobre el objeto “empl2” el proceso será el siguiente: El compilador busca una definición del método en la clase “EmpleadoDistribucion”. Como éste no existe, sube un nivel, a la clase “Asalariado”, y repite la búsqueda. En este caso la respuesta es afirmativa, y utiliza esa definición del método.

La pregunta que nos planteamos ahora es si los compiladores de Java y C++ ofrecen “enlazado dinámico” o “enlazado temprano”. En C++, por defecto, y al ser un lenguaje heredado de C donde no existía la noción de herencia (y por tanto tampoco la de polimorfismo), el enlazado es temprano (y no habrá polimorfismo). Sin embargo, existen herramientas en C++ que permiten obtener polimorfismo y “enlazado dinámico”. En Java, al ser un lenguaje diseñado siguiendo el paradigma de POO, el compilador ofrece “enlazado dinámico” para todos aquellos métodos que lo requieran y el programador no deberá preocuparse de lo mismo.

3.2 OBTENCIÓN DE POLIMORFISMO EN C++: UTILIZACIÓN DE MEMORIA DINÁMICA Y MÉTODOS VIRTUAL

En esta Sección presentaremos los requisitos necesarios en C++ para que un método se comporte de modo polimorfo. Estos requisitos son presentados en la

Sección 3.2.1, con un caso de uso sobre objetos en C++. Después presentaremos otros dos casos de uso de métodos polimorfos, que nos servirán de nuevo para ilustrar la necesidad de usar los requisitos presentados en la Sección 3.2.1.

En la Sección 3.2.2 presentaremos un caso de uso de polimorfismo de métodos trabajando con estructuras de datos genéricas. Los requisitos para que el método se comporte de modo polimorfo son los mismo presentados en la Sección 3.2.1. En este caso, nos restringiremos a la estructura de datos "array".

En la Sección 3.2.3 presentaremos un nuevo caso de uso que nos permita ilustrar cómo, los mecanismos presentados en la Sección 3.2.1 para la obtención de polimorfismo, nos permiten también conseguirlo para funciones auxiliares definidas por el usuario.

Finalmente, en la Sección 3.2.4 presentaremos las conclusiones de los anteriores casos de uso.

Tanto en la Sección 3.2.1 como en 3.2.2 y en 3.2.3 exploraremos primero el comportamiento por defecto en C++, lo que nos permitirá observar mejor las herramientas necesarias para conseguir polimorfismo.

Como acabamos de mencionar, los mecanismos que nos permitirán obtener polimorfismo en las Secciones 3.2.1, 3.2.2 y 3.2.3 son los mismos. El único motivo por el que los separamos es el de presentar distintas situaciones donde el polimorfismo puede ser aplicado.

3.2.1 POLIMORFISMO DE MÉTODOS TRABAJANDO CON OBJETOS

Antes de mostrar cómo se puede obtener polimorfismo en C++ lo que haremos será ilustrar el comportamiento por defecto en el mismo lenguaje. Para ello, de nuevo recuperamos el anterior ejemplo sobre las clases "Asalariado", "EmpleadoProduccion" y "EmpleadoDistribucion", cuyo código fuente (disponible también en la Sección 2.6) era el siguiente:

```
//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[]);
    //Los métodos redefinidos deben ser declarados en la clase correspondiente
     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[]);
  //Los métodos redefinidos deben ser declarados en la clase correspondiente
  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);
};
```

Realizamos ahora un sencillo programa cliente para la anterior aplicación:

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

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

using namespace std;

int main (){

 Asalariado empl1 ("Manuel Cortina", 12345678, 28, 1200);
 EmpleadoProduccion empl2 ("Juan Mota", 55333222, 30, 1200, "noche");

 empl1 = empl2;

 cout << "El nombre del empleado 1 es " << empl1.getNombre() << endl;
 cout << "Su salario es " << empl1.getSalario() << endl;
 //La siguiente invocación produciría un error de compilación:
 //cout << "El turno del empleado 1 es " << empl1.getTurno() << endl;
 cout << "El nombre del empleado 2 es " << empl2.getNombre() << endl;
 cout << "El turno del empleado 2 es " << empl2.getTurno() << endl;
 cout << "Su salario es " << empl2.getSalario() << endl;

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

El resultado de ejecutar el anterior fragmento de código sería:

```
El nombre del empleado 1 es Juan Mota
Su salario es 1200
El nombre del empleado 2 es Juan Mota
El turno del empleado 2 es noche
Su salario es 1380
```

Como podemos observar, hemos asignado al objeto “empl1” el objeto “empl2”. Tras realizar tal asignación, hemos invocado al método “getSalario(): double” sobre “empl1” y “empl2”.

El comportamiento que deberíamos esperar del anterior fragmento de código es que, si estamos invocando al método “getSalario(): double” sobre dos objetos que previamente hemos asignado (y, por tanto, deberían ser iguales), el comportamiento de ambas invocaciones debería ser también igual.

Sin embargo, “empl1.getSalario()” ha devuelto como valor “1200”, y “empl2.getSalario()” ha devuelto “1380” ¿Cómo podemos explicar dicho comportamiento?

La explicación está relacionada con el “enlazado” de métodos. El comportamiento del método “getSalario(): double” para el objeto “empl1” es el propio de la clase “Asalariado”. Al declarar “empl1” como objeto de la clase “Asalariado”, en tiempo de compilación se ha “enlazado” el método “getSalario(): double” con la definición del mismo existente en la clase “Asalariado”. Como el enlazado en C++, por defecto, es temprano, y se realiza en tiempo de compilación, dicho enlazado pasa a ser definitivo, y el método “getSalario(): double” sobre “empl1” tendrá siempre el comportamiento propio de la clase “Asalariado” (el método no será polimorfo).

Aprovechamos el ejemplo anterior para incidir una vez más en la relevancia de declarar un objeto como de una determinada clase. El haber declarado “empl1” objeto de la clase “Asalariado”, quiere decir que sobre este objeto sólo podemos invocar a los métodos propios de dicha clase (y no a los exclusivos de las clases derivadas). Ese es el motivo por el cual la invocación:

```
//La siguiente invocación produciría un error de compilación:
 //cout << "El turno del emplead1 es " << empl1.getTurno() << endl;
```

Produciría un error de compilación en C++ (también en Java, como veremos más adelante). El error, una vez más, es debido a que el tipo del que declaramos el objeto determina la lista de métodos (la interfaz) a la cual podemos tener acceso a través de ese objeto.

Veamos ahora cómo podemos conseguir que los métodos del objeto “empl1” se comporten de modo polimorfo. Existen dos requisitos para lo mismo en C++:

1. El primero es que, todos aquellos métodos que queramos que se porten de modo polimorfo (es decir, aquellos que han sido redefinidos y queremos que se enlacen en tiempo de ejecución) deben ser declarados con el modificador “virtual” en el correspondiente fichero de cabeceras.

En nuestro ejemplo, únicamente el método “getSalario(): double” ha sido redefinido, y por tanto puede comportarse de formas distintas. Por tanto, su declaración ahora en el fichero “Asalariado.h” pasaría a ser:

```
#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);
        virtual double getSalario ();

};

#endif
```

Como podemos observar, la única diferencia con la anterior declaración de “Asalariado.h” consiste en que el modificador “virtual” ha sido añadido al método “getSalario (): double”. No es necesario modificarlo ni en “EmpleadoProduccion.h” ni en “EmpleadoDistribucion.h”.

Ésta es una de las particularidades del modificador “virtual”. Una vez hayamos declarado un método con este modificador en una superclase, dicho método será siempre comprobado en tiempo de ejecución para cualquiera de las subclases (el haber declarado “getSalario(): double” como “virtual” en los ficheros “EmpleadoProduccion.h” y “EmpleadoDistribucion.h” no produciría ningún error de compilación, simplemente, sería redundante).

Si ahora tratamos de ejecutar la función “main” anterior veremos que el resultado de dicha acción es:

```
El nombre del emplead1 es Juan Mota
Su salario es 1200
El nombre del emplead2 es Juan Mota
El turno del emplead2 es noche
Su salario es 1380
```

De nuevo obtenemos el mismo resultado que anteriormente. Por el momento no hemos conseguido que el método “getSalario(): double” se comporte de modo polimorfo. Debemos realizar una segunda modificación en nuestro código.

2. El segundo requisito en C++ para obtener comportamiento polimorfo de métodos es que utilicemos memoria dinámica, es decir, que los objetos sean gestionados por medio de punteros o referencias. De este modo, C++, en tiempo de ejecución, es capaz de decidir a qué clase concreta pertenece el objeto al que apunta una referencia, y de ese modo “enlazar” el método redefinido con la definición del mismo correspondiente a dicha clase.

Veámoslo de nuevo con el anterior ejemplo, donde los objetos pasan ahora a estar alojados en memoria por medio de referencias:

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

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

using namespace std;

int main (){

 Asalariado * punt_empl1;
 EmpleadoProduccion * punt_empl2 = new EmpleadoProduccion ("Juan Mota",
55333222, 30, 1200, "noche");

 punt_empl1 = punt_empl2;

 cout << "El nombre del empleado 1 es " << punt_empl1->getNombre() << endl;
 cout << "Su salario es " << punt_empl1->getSalario() << endl;
 //La siguiente invocación produciría un error de compilación:
 //cout << "El turno del empleado 1 es " << punt_empl1->getTurno() << endl;
 cout << "El nombre del empleado 2 es " << punt_empl2->getNombre() << endl;
 cout << "El turno del empleado 2 es " << punt_empl2->getTurno() << endl;
 cout << "Su salario es " << punt_empl2->getSalario() << endl;

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

Como se puede observar, los objetos (tanto “punt_empl1” como “punt_empl2”) han pasado ahora a estar alojados en memoria por medio de referencias o punteros, aunque sólo con haberlo hecho para “punt_empl1” hubiese sido suficiente). El resultado de ejecutar el anterior fragmento de código ahora es:

```
El nombre del empleado 1 es Juan Mota
Su salario es 1380
El nombre del empleado 2 es Juan Mota
El turno del empleado 2 es noche
Su salario es 1380
```

Podemos ver cómo ahora, el compilador, sí ha sido capaz de, en tiempo de ejecución, “enlazar” el método “getSalario(): double” con la definición de dicho método que podemos encontrar en la clase “EmpleadoProduccion”.

Por lo tanto, podemos resumir lo anterior en las dos siguientes condiciones, que permiten conseguir comportamiento de métodos polimorfo en C++:

1. Los métodos redefinidos deben ser declarados como “virtual” en el correspondiente fichero de cabeceras.

2. Los objetos desde los que se invoca al método redefinido deben ser alojados en memoria por medio de referencias o punteros.

Recuperamos la siguiente línea del código anterior para incidir una vez más en la importancia de la diferencia entre declaración y definición de un objeto. Si bien el puntero “punt_empl1” ahora apunta a “punt_empl2” y además utiliza la definición de “getSalario(): double” propia de la clase “EmpleadoProduccion”, sigue poseyendo únicamente los métodos propios de la clase “Asalariado” (y no es capaz de acceder, por ejemplo, al método “getTurno(): string” de dicha clase).

3.2.2 POLIMORFISMO DE MÉTODOS TRABAJANDO SOBRE ESTUCTURAS DE DATOS

Antes de empezar a ilustrar este segundo caso de uso del polimorfismo en C++, debe quedar claro que los requisitos para que un método se pueda comportar de modo polimorfo son que dicho método sea declarado como “virtual”, y que los objetos sean alojados en memoria por medio de referencias o punteros.

Veamos ahora cómo sería el comportamiento de un “array” de objetos con respecto al polimorfismo. Partimos del ejemplo tal y como lo dejamos al final de la sección anterior (es decir, con “punt_empl1” y “punt_empl2”). Ahora, dentro del propio “main”, definimos un “array” de objetos (no de punteros a objetos) y le incluimos los dos objetos apuntados por “punt_empl1” y “punt_empl2”:

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

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

using namespace std;

int main (){

 Asalariado * punt_empl1 = new Asalariado ("Manuel Cortina", 12345678, 28, 1200);
 EmpleadoProduccion * punt_empl2 = new EmpleadoProduccion ("Juan Mota", 55333222, 30, 1200, "noche");

 punt_empl1 = punt_empl2;

 cout << "El nombre del empleado 1 es " << punt_empl1->getNombre() << endl;
 cout << "Su salario es " << punt_empl2->getSalario() << endl;
 cout << "El nombre del emplead2 es " << punt_empl2->getNombre() << endl;
 cout << "Su salario es " << punt_empl2->getSalario() << endl;

 Asalariado array_empl [5];
 array_empl [0] = (* punt_empl1);
```

```
 array_empl [1] = (* punt_empl2);

 for (int i = 0; i <= 1; i++){
     cout << "El salario del empleado " << i << " es " << array_empl[i].getSalario()
<< endl;
 }

 return 0;
}
```

Vemos que la primera modificación a incluir en nuestro código es que, para poder definir un “array” de objetos, debemos añadir un constructor sin parámetros para la clase “Asalariado”. Suponemos que hemos declarado (en “Asalariado.h”) y definido (en “Asalariado.cpp”) uno con valores por defecto para los distintos atributos.

El resultado de ejecutar ahora el anterior fragmento de código es:

```
El nombre del empleado 1 es Juan Mota
Su salario es 1380
El nombre del empleado 2 es Juan Mota
Su salario es 1380
El salario del empleado 0 es 1200
El salario del empleado 1 es 1200
```

Como podemos observar del resultado de ejecutar el programa, el bucle ha producido la siguiente salida:

```
El salario del empleado 0 es 1200
El salario del empleado 1 es 1200
```

Los dos objetos que hemos introducido en el bucle han utilizado el método “getSalario(): double” tal y como estaba definido en la clase “Asalariado”. Sin embargo, ambos objetos habían sido construidos con el constructor de la clase “EmpleadoProduccion”, y ése es el comportamiento que deberíamos esperar de ellos (es decir, que hubiesen utilizado el método “getSalario(): double” tal y como está definido en la clase “EmpleadoProduccion”).

La explicación a dicho comportamiento es la siguiente. Nuestro “array” ha sido declarado sobre el tipo “Asalariado”. Como no hemos utilizado memoria dinámica para alojar los objetos (en esta caso las componentes del “array”), los métodos de los mismos han sido “enlazados” de forma temprana con las definiciones de los métodos en la clase “Asalariado”. Por tanto, cualquier objeto que introduzcamos en el “array”, tanto si es de la clase “Asalariado” o de cualquiera de sus subtipos, utilizará las definiciones de métodos propias de la clase “Asalariado”. Se puede decir que hemos perdido el “enlazado dinámico” de métodos, y por tanto, también el comportamiento polimorfo de métodos.

Nota: el anterior ejemplo nos sirve también para ilustrar que no es suficiente con declarar como “virtual” los métodos que vayan a ser redefinidos, sino que también se requiere el uso de memoria dinámica sobre los objetos.

La solución al problema anterior sería, al igual que en la Sección 3.2.1, alojar los distintos objetos por medio de memoria dinámica (es decir, referencias o punteros). Veamos entonces el comportamiento del siguiente programa:

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

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

using namespace std;

int main (){

 Asalariado * punt_empl1 = new Asalariado ("Manuel Cortina", 12345678, 28, 1200);
 EmpleadoProduccion * punt_empl2 = new EmpleadoProduccion ("Juan Mota", 55333222, 30, 1200, "noche");

 punt_empl1 = punt_empl2;

 cout << "El nombre del empleado 1 es " << punt_empl1->getNombre() << endl;
 cout << "Su salario es " << punt_empl1->getSalario() << endl;
 cout << "El nombre del empleado 2 es " << punt_empl2->getNombre() << endl;
 cout << "Su salario es " << punt_empl2->getSalario() << endl;

 Asalariado * array_empl [5];
 array_empl [0] = punt_empl1;
 array_empl [1] = punt_empl2;

 for (int i = 0; i <= 1; i++){
    cout << "El salario del empleado " << i << " es " << array_empl[i]->getSalario() << endl;
 }

 return 0;
}
```

El resultado de ejecutar dicho código sería:

```
El nombre del empleado 1 es Juan Mota
Su salario es 1380
El nombre del empleado 2 es Juan Mota
Su salario es 1380
El salario del empleado 0 es 1380
```

El salario del empleado 1 es 1380

Como podemos observar, el método “getSalario(): double” que ha sido invocado sobre los (punteros a) objetos en el “array” “array_empl” es el propio de la clase “EmpleadoProduccion”, y no el de la clase “Asalariado” al haber utilizado referencias para alojar los objetos en memoria, en tiempo de ejecución se ha “enlazado” el método “getSalario(): double” con la definición dada en la clase “EmpleadoProduccion”. Del mismo modo, si añadimos a nuestro ejemplo anterior la siguiente acción:

```
array_empl[2] = new EmpleadoDistribucion ("Antonio Colorado", 11999666, 32, 1200, "Zamora");
cout << "El salario del empleado 2 es " << array_empl[2]->getSalario() << endl;
```

Observaremos que el resultado de ejecutarlo es:

El salario del empleado 2 es 1320

Al incluir en nuestro “array” un (puntero a) objeto de la clase “EmpleadoDistribucion”, observamos que el método “getSalario(): double” que se pasa a utilizar no es el propio de la clase “Asalariado” sino que, de nuevo, en tiempo de ejecución el método “getSalario(): double” ha sido “enlazado” con la definición del mismo en “EmpleadoDistribucion” y ésta es la definición que ha sido usada.

Como conclusión al ejemplo anterior, podemos decir que para obtener comportamiento polimorfo de métodos con estructuras genéricas en C++, al igual que hicimos en la Sección 3.2.1, debemos declarar los métodos correspondientes como “virtual” y hacer que los objetos sean alojados por medio de memoria dinámica (punteros o referencias).

3.2.3 POLIMORFISMO DE MÉTODOS TRABAJANDO SOBRE FUNCIONES AUXILIARES

Veremos ahora una tercera situación en la que nos gustaría contar con comportamiento polimorfo de métodos y en la cual, por defecto, no es obtenido. En primer lugar, debe quedar claro que la forma de conseguir comportamiento polimorfo va a ser idéntica a la empleada en las Secciones 3.2.1 y 3.2.2, es decir, declarando los métodos correspondientes como “virtual” y utilizando memoria dinámica para gestionar los objetos.

El ejemplo que presentamos a continuación será como sigue. Definimos una función auxiliar, “mostrarSalario(): void”, en nuestro código que va a realizar la siguiente operación: va a tomar como dato un objeto de la clase “Asalariado” (o, por tanto, de cualquiera de sus subtipos) y va a mostrar por pantalla el salario del mismo y los descuentos del salario correspondientes a impuestos (suponemos que esto asciende, en todos los casos, a un 15% del salario). El código para esta función auxiliar sería:

Declaración de la función:

```
void mostrarSalario (Asalariado);
```

Definición de la misma:

```
void mostrarSalario(Asalariado empl){
    cout << "El sueldo del trabajador es " << empl.getSalario() << endl;
    cout << "La parte descontada por impuestos es " << (0.15) * empl.getSalario() << endl;
}
```

Veamos ahora un cliente “main” que haga uso de dicha función (mostramos también la declaración y definición de “mostrarSalario(Asalariado): void”):

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

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

using namespace std;

void mostrarSalario (Asalariado);

int main (){

 Asalariado * punt_empl1 = new Asalariado ("Manuel Cortina", 12345678, 28, 1200);
 EmpleadoProduccion * punt_empl2 = new EmpleadoProduccion ("Juan Mota", 55333222, 30, 1200, "noche");
 EmpleadoDistribucion * punt_empl3 = new EmpleadoDistribucion ("Antonio Colorado", 11999666, 32, 1200, "Zamora");

 cout << "El nombre del empleado 1 es " << punt_empl1->getNombre() << endl;
 cout << "Su salario es " << punt_empl1->getSalario() << endl;
 cout << "El nombre del empleado 2 es " << punt_empl2->getNombre() << endl;
 cout << "Su salario es " << punt_empl2->getSalario() << endl;
 cout << "El nombre del empleado 3 es " << punt_empl3->getNombre() << endl;
 cout << "Su salario es " << punt_empl3->getSalario() << endl;
 mostrarSalario (* punt_empl1);
 mostrarSalario (* punt_empl2);
 mostrarSalario (* punt_empl3);

 system ("PAUSE");
 return 0;
}

void mostrarSalario(Asalariado empl){
```

```
    cout << "El sueldo del trabajador es " << empl.getSalario() << endl;
    cout << "La parte descontada por impuestos es " << (0.15) *
empl.getSalario() << endl;
}
```

Se supone que estamos haciendo uso de las clases “Asalariado”, “EmpleadoDistribucion” y “EmpleadoProduccion” que hacen uso del método “getSalario(): double” con el modificador “virtual” ya incluido. El resultado de ejecutar dicho código es:

```
El nombre del empleado 1 es Manuel Cortina
Su salario es 1200
El nombre del empleado 2 es Juan Mota
Su salario es 1380
El nombre del empleado 3 es Juan Mota
Su salario es 1320
El sueldo del trabajador es 1200
La parte descontada por impuestos es 180
El sueldo del trabajador es 1200
La parte descontada por impuestos es 180
El sueldo del trabajador es 1200
La parte descontada por impuestos es 180
```

Como podemos observar, hemos obtenido comportamiento polimorfo del método “getSalario(): double” en la parte correspondiente al “main”, ya que utilizábamos memoria dinámica para alojar los objetos y el método había sido declarada como “virtual”.

Sin embargo, la función auxiliar “mostrarSalario(Asalariado): void” ha invocado, para los tres objetos que le hemos dado como parámetros, al método “getSalario(): double” tal y como está definido en la clase “Asalariado”.

El problema es el mismo que el que hemos citado en las Secciones 3.2.1 y 3.2.2. Al declarar el parámetro de la función como “Asalariado”, en tiempo de compilación, la definición del método “getSalario(): double” que se ha enlazado (de forma “temprana”) con el parámetro de la función “mostrarSalario(Asalariado): void” es la propia de la clase “Asalariado”. Por tanto, independientemente de que pasemos como parámetro un objeto de la clase “Asalariado” o de cualquiera de sus subclases en las que dicho método ha sido redefinido, el compilador hará uso siempre de la propia de la clase “Asalariado”. De nuevo, podemos afirmar que hemos perdido el comportamiento polimorfo (al haber perdido el “enlazado dinámico”).

La solución a este problema, igual que en las Secciones 3.2.1 y 3.2.2, pasa por utilizar memoria dinámica para alojar los objetos en los cuales ha habido alguna redefinición de métodos. En este caso, dicho objeto es el parámetro de la función “mostrarSalario(Asalariado): void”, y por lo tanto pasaremos a declararla y definirla como “mostrarSalario(Asalariado *): void”.

Realizando los cambios necesarios el cliente “main” queda como:

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

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

using namespace std;

void mostrarSalario (Asalariado *);

int main (){

 Asalariado * punt_empl1 = new Asalariado ("Manuel Cortina", 12345678, 28, 1200);
 EmpleadoProduccion * punt_empl2 = new EmpleadoProduccion ("Juan Mota", 55333222, 30, 1200, "noche");
 EmpleadoDistribucion * punt_empl3 = new EmpleadoDistribucion ("Antonio Colorado", 11999666, 32, 1200, "Zamora");

 cout << "El nombre del empleado 1 es " << punt_empl1->getNombre() << endl;
 cout << "Su salario es " << punt_empl1->getSalario() << endl;
 cout << "El nombre del empleado 2 es " << punt_empl2->getNombre() << endl;
 cout << "Su salario es " << punt_empl2->getSalario() << endl;
 cout << "El nombre del empleado 3 es " << punt_empl2->getNombre() << endl;
 cout << "Su salario es " << punt_empl3->getSalario() << endl;
 mostrarSalario (punt_empl1);
 mostrarSalario (punt_empl2);
 mostrarSalario (punt_empl3);

 system ("PAUSE");
 return 0;
}

void mostrarSalario(Asalariado * empl){
    cout << "El sueldo del trabajador es " << empl->getSalario() << endl;
    cout << "La parte descontada por impuestos es "
                << (0.15)*empl->getSalario() << endl;
}
```

Y el resultado de ejecutarlo es:

```
El nombre del empleado 1 es Manuel Cortina
Su salario es 1200
El nombre del empleado 2 es Juan Mota
Su salario es 1380
El nombre del empleado 3 es Juan Mota
Su salario es 1320
El sueldo del trabajador es 1200
```

La parte descontada por impuestos es 180
El sueldo del trabajador es 1380
La parte descontada por impuestos es 207
El sueldo del trabajador es 1320
La parte descontada por impuestos es 198

Podemos observar cómo ahora la función auxiliar “mostrarSalario(Asalariado *): void” sí que ha invocado, sobre cada uno de los tres (punteros a) objetos, a la definición del método “getSalario (): double” propio de su clase (hemos recuperado el “enlazado dinámico” de métodos), y, de nuevo, hemos obtenido el comportamiento polimorfo del método.

Una vez más, los requisitos para obtener comportamiento polimorfo de un método han sido que dicho método sea declarado como “virtual” en la clase correspondiente y que utilicemos memoria dinámica para alojar los objetos que deben acceder a dicho método (en este caso, el parámetro de la función “mostrarSalario(Asalariado *): void”).

Por cierto, este último caso podía haber sido resuelto también por medio del uso de referencias:

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

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

using namespace std;

void mostrarSalario (Asalariado &);

int main (){

 Asalariado * punt_empl1 = new Asalariado ("Manuel Cortina", 12345678, 28, 1200);
 EmpleadoProduccion * punt_empl2 = new EmpleadoProduccion ("Juan Mota", 55333222, 30, 1200, "noche");
 EmpleadoDistribucion * punt_empl3 = new EmpleadoDistribucion ("Antonio Colorado", 11999666, 32, 1200, "Zamora");

 cout << "El nombre del empleado 1 es " << punt_empl1->getNombre() << endl;
 cout << "Su salario es " << punt_empl1->getSalario() << endl;
 cout << "El nombre del empleado 2 es " << punt_empl2->getNombre() << endl;
 cout << "Su salario es " << punt_empl2->getSalario() << endl;
 cout << "El nombre del empleado 3 es " << punt_empl3->getNombre() << endl;
 cout << "Su salario es " << punt_empl3->getSalario() << endl;
 mostrarSalario (* punt_empl1);
 mostrarSalario (* punt_empl2);
 mostrarSalario (* punt_empl3);
```

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

void mostrarSalario(Asalariado & empl){
    cout << "El sueldo del trabajador es " << empl.getSalario() << endl;
    cout << "La parte descontada por impuestos es " << (0.15) *
empl.getSalario() << endl;
}
```

Como conclusión al ejemplo de las funciones auxiliares, podemos señalar que para conseguir funciones auxiliares o métodos auxiliares en los cuales los métodos se comporten de modo polimorfo es imprescindible que el paso de los parámetros (en este caso los objetos) sobre los cuales queremos obtener polimorfismo se haga por medio de punteros o referencias.

3.2.4 CONCLUSIONES

La conclusión a la que llegamos tras los distintos ejemplos que hemos introducido en las páginas anteriores es la siguiente:

Para conseguir que la declaración de un método se “enlace dinámicamente” con su definición correspondiente (y por tanto el método se comporte de modo polimorfo) es necesario declarar el método como “virtual” y hacer que, en el contexto en que ese método vaya a ser invocado, los objetos estén alojados de forma dinámica (por contexto entendemos un “array”, una función auxiliar, o, en general, cualquier fragmento de código).

3.3 POLIMORFISMO EN JAVA

Como ya hemos dicho en la Sección 3.1, en Java el enlazado de los métodos con sus correspondientes definiciones se realiza siempre en tiempo de ejecución, de forma dinámica, lo cual quiere decir que los métodos se comportarán siempre de modo polimorfo.

Lo que haremos será ilustrar distintos ejemplos en los cuales los métodos se comportan de modo polimorfo en Java, siguiendo los ejemplos que hemos introducido en C++. En la Sección 3.3.1 veremos un ejemplo que ilustra el polimorfismo de métodos sobre objetos. En la Sección 3.3.2 un segundo caso de uso del polimorfismo de métodos, en este caso sobre estructuras genéricas. En la Sección 3.3.3 presentaremos un tercer caso de uso del polimorfismo de métodos con el uso de funciones auxiliares.

Antes de pasar a hablar del polimorfismo, cabe mencionar que en Java existe una forma de evitar que los métodos que definimos en nuestras clases puedan ser redefinidos (y, por tanto, deje de tener sentido hablar de comportamiento polimorfo y de enlazado dinámico). Esto se consigue por medio del uso del modificador “final” sobre métodos en Java. Ya vimos como el modificador “final” para atributos de clase hacía que dichos atributos no pudieran ser modificados,

una vez habían recibido un valor inicial. Cuando adjuntamos el modificador “final” a un método en una clase, este método no podrá ser redefinido por ninguna de las clases derivadas. Por tanto, el comportamiento del método pasa a ser inmutable para cualquier clase derivada. De este modo, el método puede ser enlazado de forma estática y el compilador puede optimizar el acceso al mismo.

3.3.1 POLIMORFISMO DE MÉTODOS TRABAJANDO CON OBJETOS

Recuperamos el ejemplo que introdujimos en C++ en la Sección 3.2.1. Para ello, debemos hacer uso de la definición de las clases “Asalariado”, “EmpleadoProduccion” y “EmpleadoDistribucion” tal y como la dimos en la Sección 2.6.

```
//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);
	}
}
```

En los ficheros anteriores podemos observar que no hemos introducido ninguna modificación. Definimos ahora el siguiente cliente de las clases anteriores.

```
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");

		emplead1 = emplead2;

		System.out.println ("El nombre del empleado 1 es " +
emplead1.getNombre());
		System.out.println ("El sueldo del empleado 1 es " +
emplead1.getSalario());
		System.out.println ("El nombre del empleado 2 es " +
emplead2.getNombre());
		System.out.println ("El sueldo del empleado 2 es " +
emplead2.getSalario());
	}
}
```

En el programa anterior hemos declarado y construido dos objetos “emplead1” y “emplead2”, y los hemos construido uno de la clase “Asalariado” y otro de la clase “EmpleadoProduccion”. Después, al objeto “emplead1” le hemos asignado el objeto “emplead2” (recordamos ahora que en Java, por su modelo de memoria, lo que pasa es que la referencia “emplead1” pasa a apuntar a la dirección de memoria a la que apuntaba la referencia “emplead2”).

Al invocar entonces al método “getSalario(): double”, en tiempo de ejecución, es capaz de identificar que el objeto al que está apuntando “emplead1” pertenece a la clase “EmpleadoProduccion”, y que por tanto debe utilizar la definición de “getSalario(): double” propia de dicha clase (y no la de “Asalariado”, que es la clase de la que declaramos “emplead1”).

Por tanto, el resultado de la ejecución de dicho código es:

El nombre del empleado 1 es Juan Mota
El sueldo del empleado 1 es 1380.0
El nombre del empleado 2 es Juan Mota
El sueldo del empleado 2 es 1380.0

Como se puede observar, el método “getSalario(): double” invocado ha sido el esperado (sin necesidad de haber modificado nada en nuestro código).

3.3.2 POLIMORFISMO DE MÉTODOS TRABAJANDO SOBRE ESTUCTURAS DE DATOS

Veamos ahora un segundo caso en donde también debe intervenir el polimorfismo de métodos para que nuestro programa tenga el comportamiento deseado. Declaramos ahora un “array” de objetos de la clase “Asalariado” y utilizamos el mismo para alojar objetos de dicha clase y también de sus subclases (aprovechamos la ocasión para recordar que esto es posible ya que entre una clase y sus subclases siempre hay una relación de subtipado).

```
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());

                Asalariado [] array_asal = new Asalariado [3];
```

```
                array_asal [0] = emplead1;
                array_asal [1] = emplead2;
                array_asal [2] = emplead3;

                for (int i = 0; i < 3; i ++){
                        System.out.println ("El sueldo del trabajador " + i + " es " +
array_asal[i].getSalario());
                }
                //Aprovechamos la ocasión para ilustrar de nuevo la diferencia
entre declaración y definición
                //System.out.println ("La region del emplead3 es " +
emplead3.getRegion());
                //System.out.println ("La region del emplead3 es " +
array_asal[2].getRegion());
        }
}
```

El resultado de ejecutar el código anterior es:

```
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
El sueldo del trabajador 0 es 1200.0
El sueldo del trabajador 1 es 1380.0
El sueldo del trabajador 2 es 1320.0
```

Como podemos observar, cada invocación del método “getSalario(): double”, a pesar de ser realizada desde un “array” de objetos de la clase “Asalariado”, ha sido capaz de utilizar la definición del método “getSalario(): double” correspondiente a la clase de la que había sido construido.

Aprovechamos el ejemplo anterior de nuevo para volver a ilustrar las consecuencias de declarar un objeto como perteneciente a una clase. Si bien la llamada “array_asal[2].getSalario()” ha sido capaz de llamar a la definición del método “getSalario(): double” propia de la clase “EmpleadoDistribucion”, si intentamos invocar a “array_asal[2].getRegion()” podemos observar que se producirá un error de compilación, ya que el objeto “array_asal[2]” ha sido declarado como de la clase “EmpleadoDistribucion” y no dispone de método “getRegion (): string”.

3.3.3 POLIMORFISMO DE MÉTODOS TRABAJANDO SOBRE FUNCIONES AUXILIARES

Presentamos ahora un tercer marco en el que el polimorfismo de métodos de nuevo aparece. Se trata del uso de funciones auxiliares. Imitando el ejemplo presentado en la Sección 3.2.3, vamos a definir una función auxiliar “mostrarSalario(Asalariado): void” que muestre por pantalla el salario de un objeto de la clase “Asalariado” y el resultado de la retención de impuestos sobre el mismo (calcularemos un 15%), y vamos a llamar a la misma con objetos pertenecientes a la clases “Asalariado” y a sus subtipos “EmpleadoProduccion” y “EmpleadoDistribucion”.

```
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");

		mostrarSalario (emplead1);
		mostrarSalario (emplead2);
		mostrarSalario (emplead3);
	}

	public static void mostrarSalario(Asalariado asl){
		System.out.println ("El salario del trabajador es " + asl.getSalario());
		System.out.println ("El resultado de aplicarle la retencion es " + asl.getSalario() * 0.15);

	}

}
```

El resultado de ejecutar el anterior código es:

```
El salario del trabajador es 1200.0
El resultado de aplicarle la retencion es 180.0
El salario del trabajador es 1380.0
El resultado de aplicarle la retencion es 207.0
El salario del trabajador es 1320.0
El resultado de aplicarle la retencion es 198.0
```

Una vez más, a pesar de que la función auxiliar “mostrarSalario(Asalariado): void” ha definido su parámetro como de tipo “Asalariado”, en tiempo de ejecución se ha comprobado la clase a la que pertenecían cada uno de sus parámetros y se ha utilizado la definición de ”getSalario(): double” propia de dicha clase (por tanto, hemos tenido enlazado dinámico de métodos, y comportamiento polimorfo).

3.3.4 CONCLUSIONES

Como conclusión a los ejemplos anteriores sólo nos cabe señalar que en Java siempre hay comportamiento polimorfo de métodos, y que tal comportamiento no requiere ninguna modificación por parte del programador.

3.4 UTILIZACIÓN DE MÉTODOS POLIMORFOS SOBRE EJEMPLOS YA CONSTRUIDOS

En esta Sección nos detendremos un poco más detalladamente en el uso del polimorfismo con algunos ejemplos más elaborados que los vistos en las Secciones anteriores. Esto nos permitirá introducir también algunas ideas nuevas sobre el polimorfismo.

Lo que haremos será extraer algunos ejemplos de la librería estándar de Java (de la API, http://java.sun.com/javase/6/docs/api/) que nos permitan ilustrar el uso de la misma y que además nos ayuden a comprender mejor el polimorfismo.

Para empezar, visitaremos la especificación de la clase “Object” en Java (http://java.sun.com/javase/6/docs/api/java/lang/Object.html). Como ya comentamos en el Tema 2 al hablar de la herencia, todas las clases que definimos en Java heredan de la clase “Object”. Esto lo hace el compilador de forma transparente al usuario, pero conviene que seamos conscientes de ello. Si observamos la especificación de dicha clase, observaremos que dispone de un número considerable de métodos. Algunos son utilizados con frecuencia.

Veamos el uso de algunos de ellos (para ello recuperamos el caso de uso de las clases “Asalariado”, “EmpleadoDistribucion” y “EmpleadoProduccion”):

```
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");

                //Uso de algunos de los métodos heredados de la clase “Object”:

                System.out.println (“El objeto emplead1 es igual que el objeto emplead1: ” + emplead1.equals(emplead1));

                System.out.println (“El objeto emplead1 es igual que el objeto emplead2: ” + emplead1.equals(emplead2));

                //Mostramos la clase a la que pertenece cada objeto:
```

```
                System.out.println (“La clase de emplead1 ” +
emplead1.getClass().getName());
                System.out.println (“La clase de emplead2 ” +
emplead2.getClass().getName());

                //Finalmente mostraremos por pantalla los objetos:

                System.out.println (“El objeto emplead1 ” + emplead1.toString());
                System.out.println (“El objeto emplead1 ” + emplead2.toString());
        }
}
```

El resultado de ejecutar el código anterior sería:

```
El objeto emplead1 es igual que el objeto emplead1: true
El objeto emplead1 es igual que el objeto emplead2: false
La clase de emplead1 Asalariado
La clase de emplead2 EmpleadoProduccion
El objeto emplead1 Asalariado@addbf1
El objeto emplead2 EmpleadoProduccion@42e816
```

Podemos observar el comportamiento de alguno de los métodos propios de la clase “Object”.

Por ejemplo, el método “equals (Object): boolean” nos permite comparar dos objetos (incluso si éstos tienen distintos tipos, puesto que ambos heredan de la clase “Object”). Su comportamiento exacto lo puedes encontrar en http://java.sun.com/javase/6/docs/api/java/lang/Object.html#equals(java.lang.Object).

El método “getClass(): Class” nos devuelve un objeto de tipo “Class”, sobre el que podemos invocar a “getName(): String” para conocer el nombre de la clase a la que pertenece un objeto. Su comportamiento exacto lo puedes encontrar en http://java.sun.com/javase/6/docs/api/java/lang/Object.html#getClass().

El método “toString(): String” nos ha servido para conseguir convertir un objeto de una clase cualquiera a una cadena de caracteres. La cadena obtenida sólo nos ha aportado información útil sobre la clase a la que pertenecía el objeto y la dirección de memoria que está ocupando, pero no así sobre el valor de sus atributos. Su comportamiento exacto está detallado en http://java.sun.com/javase/6/docs/api/java/lang/Object.html#toString().

Estos métodos (“equals (Object): boolean” o “toString(): String”) están definidos de una forma tan genérica (están definidos para cualquier clase que herede en Java de “Object”, es decir, para cualquier clase que se defina) que su comportamiento muchas veces puede resultar de poca utilidad.

Por ejemplo, observemos el siguiente ejemplo de uso del método “equals(Object): boolean”:

```
public class Principal_EjemploAsalariado{

	public static void main (String [] args){

		Asalariado emplead1 = new Asalariado ("Manuel Cortina", 12345678, 28, 1200);
		Asalariado emplead3 = new Asalariado ("Manuel Cortina", 12345678, 28, 1200);

		System.out.println ("El objeto emplead1 es igual que el objeto emplead3: " + emplead1.equals(emplead3));

	}
}
```

El resultado de ejecutar el anterior fragmento de código en Java sería:

El objeto emplead1 es igual que el objeto emplead3: false

Dos objetos que tienen todos sus atributos iguales, y, sin embargo, el método “equals (Object): boolean” ha devuelto “false” al ser interrogado por su igualdad. Esto se debe a que, básicamente, el método “equals (Object): boolean”, tal y como está definido en la librería de Java, compara las referencias a las que apuntan “emplead1” y “emplead3” y, si ambos objetos apuntan a la misma referencia (es decir, a la misma zona de memoria), devuelve “true”.

Es muy probable que en nuestros programas necesitemos una versión del método “equals(Object): boolean” que sea un poco menos restrictiva. Por ejemplo, puede que nos interese tener un método para determinar si dos objetos son iguales simplemente comparando todos y cada uno de sus atributos. Ello exigiría por parte del programador la tarea de redefinir el método de forma conveniente.

Veremos un ejemplo más desarrollado de lo mismo para el método propio de la librería de Java “toString(): String”. Si observamos con más atención la especificación del método “toString(): String” dada en la librería en Java nos encontraremos con el siguiente consejo (http://java.sun.com/javase/6/docs/api/java/lang/Object.html#toString()):

“It is recommended that all subclasses override this method.” (“Se recomienda que todas las subclases redefinan este método”.)

Además, como hemos podido observar anteriormente, su comportamiento era bastante poco informativo sobre el valor real que tienen los atributos de un objeto:

El objeto emplead1 Asalariado@addbf1
El objeto emplead2 EmpleadoProduccion@42e816

Lo que vamos a hacer ahora es proponer una redefinición del mismo para las clases “Asalariado”, “EmpleadoProduccion” y “EmpleadoDistribucion” que nos permita mostrar los diversos atributos de los mismos:

```
//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;
	}
```

```
	public String toString(){
		return ("La clase a la que pertenece el objeto es "
					+ this.getClass().getName() + "\n" +
				"El nombre del asalariado es "
					+ this.getNombre() + "\n" +
				"El dni del asalariado es "
					+ this.getDni() + "\n" +
				"Los dias de vacaciones del asalariado son "
					+ this.getDiasVacaciones() + "\n" +
				"El salario base del asalariado es "
					+ this.getSalario() + "\n");
	}
}

//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);
	}

	public String toString(){
		return (super.toString ()
				+ "El turno del empleado es "
				+ this.getTurno() + "\n");
	}
}

//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);
	}

	public String toString(){
		return (super.toString () +
				"La region del empleado es "
				+ this.getRegion() + "\n");
	}
}
```

Definimos ahora un cliente sencillo del anterior diagrama de clases en Java:

```
//Fichero Principal_EjemploAsalariado.java


public class Principal_EjemploAsalariado{

	public static void main (String [] args){

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

		System.out.println (emplead1.toString());
		System.out.println (emplead2.toString());
		System.out.println (emplead3.toString());
	}
}
```

El resultado de ejecutar el cliente anterior de la aplicación sería:

La clase a la que pertenece el objeto es Asalariado
El nombre del asalariado es Manuel Cortina
El dni del asalariado es 12345678
Los dias de vacaciones del asalariado son 28
El salario base del asalariado es 1200.0

La clase a la que pertenece el objeto es EmpleadoProduccion
El nombre del asalariado es Juan Mota
El dni del asalariado es 55333222
Los dias de vacaciones del asalariado son 30
El salario base del asalariado es 1380.0
El turno del empleado es noche

La clase a la que pertenece el objeto es EmpleadoDistribucion
El nombre del asalariado es Antonio Camino
El dni del asalariado es 55333666
Los dias de vacaciones del asalariado son 35
El salario base del asalariado es 1320.0
La region del empleado es Granada

Hay varias cosas interesantes que se podrían resaltar sobre el código anterior:

1. En primer lugar, como hemos observado en la API de Java, la cabecera del método “toString(): String” tiene la siguiente especificación:

public String toString()

Por tanto, ésa es la definición que le debemos dar en las clases en las que queramos redefinir dicho método.

2. En segundo lugar, con respecto a la definición de dicho método en las clases “Asalariado”, “EmpleadoProduccion” y “EmpleadoDistribucion”, podemos observar lo siguiente (tomamos como ejemplo la definición en “Asalariado.java”):

```
public String toString(){
        return ("La clase a la que pertenece el objeto es "
                        + this.getClass().getName() + "\n" +
                "El nombre del asalariado es "
                        + this.getNombre() + "\n" +
                "El dni del asalariado es "
                        + this.getDni() + "\n" +
                "Los dias de vacaciones del asalariado son "
                        + this.getDiasVacaciones() + "\n" +
                "El salario base del asalariado es "
                        + this.getSalario() + "\n");
}
```

Podemos observar que en la definición del método hemos hecho uso de métodos “getNombre(): String”, “getDni(): String”, “getDiasVacaciones(): int” y

“getSalario(): double” que no son propios de la clase “Object” (donde se definía el método “toString(): String”) sino de la clase “Asalariado”.

3. Un tercer dato interesante sobre el anterior fragmento de código es que, para poder invocar al método “toString(): String” desde un objeto, basta con que éste esté declarado como de la clase “Object”:

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

System.out.println (emplead1.toString());
System.out.println (emplead2.toString());
System.out.println (emplead3.toString());
```

El método “toString(): String” es propio de la clase “Object”, y por tanto resulta suficiente con declarar un objeto de dicha clase para poder hacer uso del mismo. Gracias al enlazado dinámico, en tiempo de ejecución, ese objeto es enlazado con la definición propia de la clase a la que pertenece dicho objeto en tiempo de ejecución (“Asalariado”, “EmpleadoProduccion”, ...). Como hemos mencionado en el punto 2, la definición de “toString(): String” en dichas clases puede hacer uso de métodos que no sean propios de la clase “Object”.

4. Por último, conviene también destacar como, la invocación al método “getSalario(): double” que se realiza desde “toString(): String” en “Asalariado”, se ha comportado también de manera polimorfa, llamando a la definición de “getSalario(): double” de la clase “Asalariado”, “EmpleadoDistribucion” o “EmpleadoProduccion” dependiendo del objeto desde el que se invocara a dicho método.

Podemos ver ahora como, desde la definición de “toString(): String” en la clase “Asalariado”

```
public String toString(){
        return ("La clase a la que pertenece el objeto es "
                                        + this.getClass().getName() + "\n" +
                "El nombre del asalariado es "
                                        + this.getNombre() + "\n" +
                "El dni del asalariado es "
                                        + this.getDni() + "\n" +
                "Los dias de vacaciones del asalariado son "
                                        + this.getDiasVacaciones() + "\n" +
                "El salario base del asalariado es "
                                        + this.getSalario() + "\n");
}
```

obtenemos los siguientes comportamientos del método “getSalario(): double” (es decir, los propios de la clase “Asalariado”, “EmpleadoProduccion” y “EmpleadoDistribucion”):

La clase a la que pertenece el objeto es Asalariado
El nombre del asalariado es Manuel Cortina
El dni del asalariado es 12345678
Los dias de vacaciones del asalariado son 28
El salario base del asalariado es 1200.0

La clase a la que pertenece el objeto es EmpleadoProduccion
El nombre del asalariado es Juan Mota
El dni del asalariado es 55333222
Los dias de vacaciones del asalariado son 30
El salario base del asalariado es 1380.0
El turno del empleado es noche

La clase a la que pertenece el objeto es EmpleadoDistribucion
El nombre del asalariado es Antonio Camino
El dni del asalariado es 55333666
Los dias de vacaciones del asalariado son 35
El salario base del asalariado es 1320.0
La region del empleado es Granada

Los ejemplos anteriores nos han servido para mostrar algunas situaciones nuevas en las que el polimorfismo puede resultar de utilidad. Situaciones similares se podrían también implementar en C++, aunque carezca de una súperclase “Object” de la cual hereden todas las clases definidas en el lenguaje.

La librería de Java (accesible a través de la API) es una buena fuente para encontrar otros muchos casos de redefinición de métodos.