TEMA 4. CLASES ABSTRACTAS E INTERFACES

Introducción:

La posibilidad de definir relaciones de herencia entre clases, dando lugar a relaciones de subtipado entre las mismas, nos ha permitido en el Tema 3 definir el polimorfismo de métodos (es decir, que un mismo método tuviese distintas definiciones, y además los objetos fuesen capaces de acceder a la definición del método adecuada en tiempo de ejecución).

Estas mismas relaciones de subtipado entre clases daban lugar a una segunda situación menos deseable. Siempre que declaramos un objeto como perteneciente a un tipo (por ejemplo, al declarar una estructura genérica o al definir funciones auxiliares), restringimos la lista de métodos (o la interfaz) que podemos utilizar de dicho objeto a los propios de la clase declarada.

En este caso estábamos perdiendo información (en la forma de métodos a los que poder acceder) que podrían sernos de utilidad. Una posible solución a este problema la ofrecen los métodos abstractos. Un método abstracto nos da la posibilidad de introducir la declaración de un método (y no su definición) en una clase, e implementar dicho método en alguna de las subclases de la clase en que nos encontramos.

De este modo, la declaración del método estará disponible en la clase, y lo podremos utilizar para, por ejemplo, definir otros métodos, y además no nos vemos en la obligación de definirlo, ya que su comportamiento puede que sea todavía desconocido.

Una consecuencia de definir un método abstracto es que la clase correspondiente ha de ser también abstracta, o, lo que es lo mismo, no se podrán crear objetos de la misma (¿cómo sería el comportamiento de los objetos de la misma al invocar a los métodos abstractos?), pero su utilidad se observará al construir objetos de las clases derivadas. Además, la posibilidad de declarar métodos abstractos enriquecerá las jerarquías de clases, ya que más clases podrán compartir la declaración de un método (aunque no compartan su definición).

La idea de un método abstracto puede ser fácilmente generalizada a clases que sólo contengan métodos abstractos, y nos servirán para relacionar clases (por relaciones de subtipado) que tengan una interfaz (o una serie de métodos) común (aunque las definiciones de los mismos sean diferentes en cada una de las subclases).

El presente Tema comenzará con la Sección 4.1 (“Definición de métodos abstractos en POO. Algunos ejemplos de uso”) donde presentaremos más concretamente la noción de método abstracto e ilustraremos algunas situaciones en las que los mismos pueden ser de utilidad; la propia noción de método abstracto nos llevará a la noción de clase abstracta que introduciremos en la misma Sección. La Sección 4.2 nos servirá para repasar la idea de

polimorfismo y para mostrar la dependencia de los métodos abstractos en la presencia del mismo; aprovecharemos también para introducir las sintaxis propias de Java y C++ de métodos y clases abstractas. En la Sección 4.3 mostraremos cómo se puede generalizar la noción de método abstracto para llegar a la de clase completamente abstracta. También presentaremos la noción de "interface" en Java, partiendo de la idea de que una "interface" está basada en la presencia de métodos abstractos, pero poniendo énfasis también en el hecho de que en Java las "interfaces" permiten implementar ciertas situaciones que a través de clases abstractas no serían posibles. La Sección 4.4 nos servirá para repasar la notación propia de UML para los conceptos introducidos en el Tema (aunque ya la habremos introducido antes), del mismo modo que la Sección 4.5 y la Sección 4.6 las utilizaremos para recuperar las sintaxis propias de dichas nociones en C++ y Java.

4.1 DEFINICIÓN DE MÉTODOS ABSTRACTOS EN POO. ALGUNOS EJEMPLOS DE USO

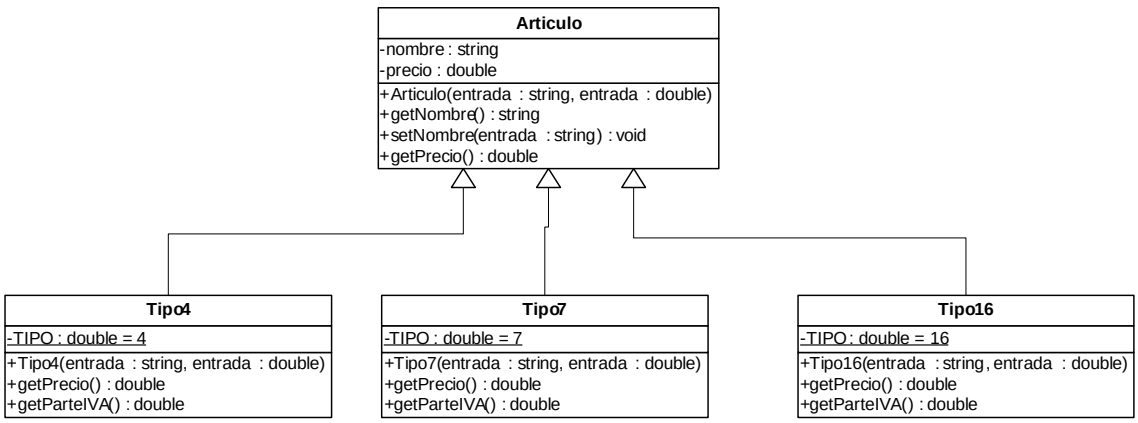
4.1.1 MÉTODOS ABSTRACTOS: DEFINICIÓN Y NOTACIÓN UML

Definición: un método abstracto es un método de una clase (o también de una "interface" en Java) que no tiene implementación o definición (es decir, sólo tiene declaración).

Sus principales usos son, en primer lugar, como un "parámetro indefinido" en expresiones que contienen objetos de dicha clase, que debe ser *redefinido* en alguna de las subclases que heredan de dicha clase (o que implementan la "interface"). En segundo lugar, sirve para definir "interfaces abstractas" de clases (entendido como partes públicas de las mismas) que deberán ser definidas por las subclases de las mismas.

Por tanto, lo primero que debemos observar es que al hablar de métodos abstractos hemos tenido que recuperar las nociones de herencia y subclases (Secciones 2.3, y ss.), así como de redefinición de métodos (Sección 2.6), e, implícitamente, de polimorfismo (Tema 3).

Presentamos ahora un ejemplo sencillo de método abstracto que nos permita ilustrar mejor su utilidad. En la Práctica 8 introducíamos el siguiente ejemplo sobre una posible implementación de artículos y su IVA correspondiente:

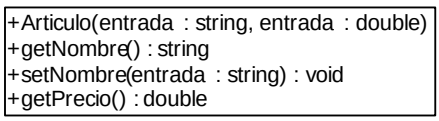


Como comentábamos en la Práctica 8, sobre el diagrama de clases anterior, no es posible utilizar el método “getParteIVA(): double” sobre los objetos que hayan sido declarados como propios de la clase “Articulo”.

Incluso lo especificábamos por medio del siguiente error:

```
Articulo art1:
art1 = new Tipo7 (“Gafas”, 160);
//La siguiente invocación produce un error de compilación:
//art1.getParteIVA();
```

El objeto “art1”, declarado como de la clase “Articulo”, sólo puede acceder, directamente, a los métodos especificados en el diagrama UML para dicha clase (como regla general se puede enunciar que un objeto sólo puede acceder, directamente, a los métodos de la clase de la que ha sido declarado, no construido):

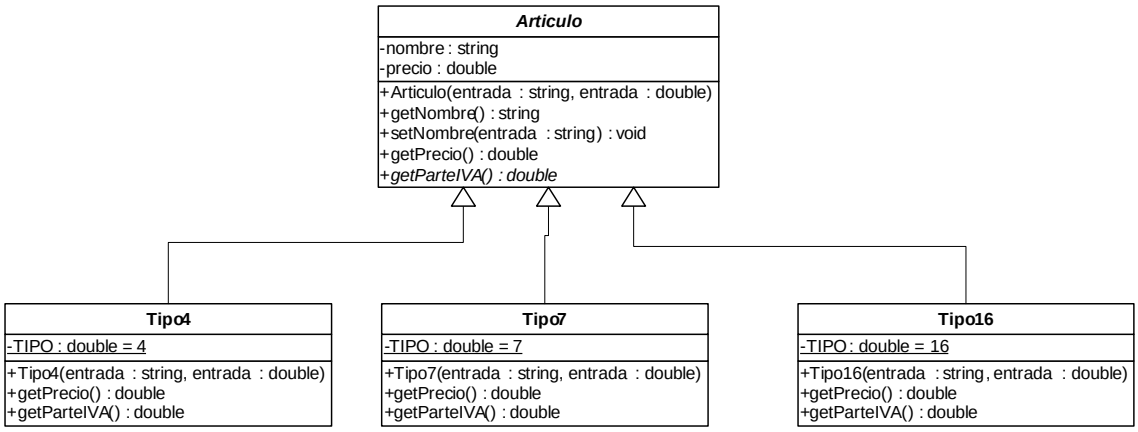


Si miramos detenidamente el diagrama anterior, podemos observar cómo todas las clases que heredan de la clase “Articulo” (en este caso “Tipo4”, “Tipo7” y “Tipo16”) contienen el método “getParteIVA(): double” junto con una definición del mismo. Sin embargo, en la clase “Articulo” no podemos especificar un comportamiento para el mismo, ya que desconocemos el valor de la constante “TIPO: double”, que ha sido definida únicamente en las subclases.

Por lo tanto, hay un buen motivo para declarar el método “getParteIVA(): double” en la clase “Articulo”, ya que es común a todas sus subclases, y todos los artículos deben tener un “IVA” asignado; por el contrario, en la clase “Articulo” todavía no podemos dar una definición adecuada del mismo.

Para resolver esta situación (en la cual queremos *declarar* un método en una clase para enriquecer su parte pública y poder accederlo directamente desde los objetos declarados de la misma, pero no lo podemos *definir* hasta sus subclases) es para lo que nos van a ayudar los métodos abstractos.

Gracias a la inclusión de métodos abstractos en nuestro ejemplo, nos encontramos con el siguiente diagrama de clases en UML (veremos que luego incluso lo podemos simplificar más):



Lo primero que se debe observar en el mismo es que el método "*getParteIVA(): double*" ha pasado a formar parte de la clase "Articulo". Sin embargo, este método tiene una peculiaridad, y es que su definición de momento no es posible (¿qué valor tomaría la constante "TIPO: double" para un artículo genérico?). Sólo podemos declararlo (es decir, especificar su cabecera, pero no su comportamiento).

Por tanto, en el diagrama UML aparece en *letra cursiva*, lo cual quiere decir que es abstracto (veremos luego la sintaxis propia de métodos abstractos para C++ y Java).

Nos detenemos ahora en las consecuencias de que el método "*getParteIVA(): double*" haya sido declarado como abstracto en la clase "Articulo". Imaginemos que sobre el diagrama de clases anterior tratamos de ejecutar el siguiente fragmento de código:

```
Articulo art1:
art1 = new Articulo ("Periodico", 1.10);
//La siguiente invocación produce un error de compilación:
//art1.getParteIVA();
```

¿Qué comportamiento debería tener? En realidad, el método "*getParteIVA(): double*" en la clase "*Articulo*" posee una declaración, pero no una definición, por lo que el comportamiento de la llamada anterior no está definido.

Para resolver esta situación surge la noción de *clase abstracta*.

4.1.2 CLASES ABSTRACTAS: DEFINICIÓN Y VENTAJAS DE USO

Definición: una clase abstracta es una clase de la cual no se pueden definir instancias (u objetos).

Por tanto, las clases abstractas tendrán dos utilidades principales:

1. En primer lugar, evitan que los usuarios de la clase puedan crear objetos de la misma, como dice la definición de clase abstracta. De este modo, en nuestro ejemplo anterior, no se podrán crear instancias de la clase "*Articulo*". Éste es

un comportamiento deseado, ya que si bien "*Articulo*" nos permite crear una jerarquía sobre las clases "Tipo4", "Tipo7" y "Tipo16", un objeto de la clase "*Articulo*" como tal no va a aparecer en nuestro desarrollo (todos los artículos tendrán siempre un IVA asignado). Sin embargo, es importante notar que sí se pueden *declarar* objetos de la clase "*Articulo*" (que luego deberán se construidos como de las clases "Tipo4", "Tipo7" ó "Tipo16").

2. En segundo lugar, permiten crear interfaces que luego deben ser implementados por las clases que hereden de la clase abstracta. Es evidente que una clase abstracta, al no poder ser instanciada, no tiene sentido hasta que una serie de clases que heredan de ella la implementan completamente y le dan un significado a todos sus métodos. A estas clases, que son las que hemos utilizado a lo largo de todo el curso, las podemos nombrar *clases concretas* para diferenciarlas de las clases abstractas.

De la propia definición de clase abstracta no se sigue que una clase abstracta deba contener algún método abstracto, aunque generalmente será así. En realidad, el hecho de definir una clase cualquiera como abstracta se puede entender como un forma de evitar que los usuarios finales de la misma puedan crear objetos de ella; es como una medida de protección que el programador de una clase pone sobre la misma.

Sin embargo, lo contrario sí es cierto siempre: si una clase contiene un método abstracto, dicha clase debe ser declarada como abstracta. Si hemos declarado un método abstracto en una clase, no podremos construir objetos de dicha clase (ya que, ¿cuál sería el comportamiento de dicho método al ser invocado desde un objeto de la clase? Estaría sin especificar, o sin definir).

Por lo tanto, como resumen a los dos anteriores párrafos, si bien que un método esté declarado como abstracto implica que la clase en la que se encuentra debe ser declarada como abstracta (en caso contrario obtendremos un error de compilación), que una clase sea abstracta no implica que alguno de los métodos que contiene haya de serlo (únicamente implica que no se pueden crear objetos de la misma).

Por este motivo, en el ejemplo anterior, el hecho de declarar el método "*getParteIVA(): double*" como abstracto tiene como consecuencia que la clase "*Articulo*" deba ser definida como abstracta.

La notación UML para clases abstractas consiste en escribir en letra cursiva el nombre de dicha clase, como se puede observar en el diagrama anterior (en nuestro ejemplo, "*Articulo*").

Por lo tanto, ya no podremos crear objetos de la clase "*Articulo*" en nuestra aplicación. Sin embargo, aunque una clase sea abstracta, podemos observar cómo puede contener atributos ("nombre: string" y "precio: double"), constructores ("Articulo(string, double)") o métodos no abstractos ("getNombre(): string", ...).

Una vez más, insistimos en que la única consecuencia de declarar una clase como abstracta es que evitamos que los usuarios de la clase puedan definir instancias de la misma (aunque sí pueden declararlas). Salvo esto, es una clase que puede contener atributos, constantes, constructores y métodos (tanto abstractos como no abstractos).

La siguiente pregunta podría ser formulada: ¿Qué utilidad tiene un constructor de una clase abstracta, si no se pueden crear objetos de la misma?

La respuesta a dicha pregunta es doble:

1. En primer lugar, para inicializar los atributos que pueda contener la clase abstracta (en nuestro ejemplo anterior, el “nombre: string”, o el “precio: double”).

2. En segundo lugar, y volviendo a uno de los aspectos que enfatizamos al introducir las relaciones de herencia (“La primera orden que debe contener un constructor de una clase derivada es una llamada al constructor de la clase base”), el constructor de una clase abstracta será de utilidad para que lo invoquen todos los constructores de la las clases derivadas. En realidad, éstas deberían ser las únicas invocaciones a los mismos que contuviera nuestro sistema, ya que no se pueden crear objetos de las clases abstractas.

4.1.3 AUMENTANDO LA REUTILIZACIÓN DE CÓDIGO GRACIAS A LOS MÉTODOS ABSTRACTOS

Como ventajas de uso de las clases abstractas hemos señalado ya que permiten al programador decidir qué clases van a poder ser instanciables (se van a poder crear objetos de ellas) y cuáles no (es decir, van a servir sólo para hacer de soporte para programar nuevas clases por herencia).

También hemos señalado que los métodos abstractos nos permiten declarar métodos sin tener que definirlos, y de este modo “enriquecer” la parte visible (“public”, “protected” o “package”) de una clase, dotándola de más métodos (que no es necesario definir hasta más adelante).

Estos métodos declarados como abstractos pueden ser también utilizados para definir los métodos restantes de la clase abstracta, permitiéndonos así reutilizar código para diversas clases. Veámoslo con un ejemplo.

Gracias a la declaración del método “*getParteIVA(): double*” (como método abstracto) dentro de la clase “*Articulo*”, hemos enriquecido la lista de métodos disponibles en dicha clase.

Esto nos permite dar una nueva definición ahora para alguno de los métodos de la clase “*Articulo*” que haga uso de los métodos abstractos añadidos (“*getParteIVA(): double*”). En nuestro caso concreto, vamos a empezar por observar la definición que habíamos dado del método “getPrecio(): double” en las clases “Articulo”, “Tipo4”, “Tipo7” y “Tipo16” antes de declarar la clase

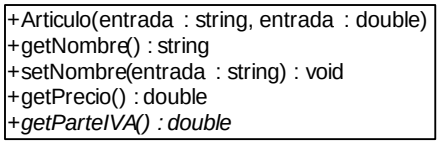
“Articulo” como abstracta (mostramos la definición de los mismos en Java, que no difiere de la que se podría dar en C++ salvo los detalles propios de la sintaxis de cada lenguaje):

```
//Clase Articulo

public double getPrecio (){
        return this.precio;
}

//Clase Tipo4

public double getPrecio (){
        return (super.getPrecio() + this.getParteIVA());
}

//Clase Tipo7

public double getPrecio (){
        return (super.getPrecio() + this.getParteIVA());
}

//Clase Tipo16

public double getPrecio (){
        return (super.getPrecio() + this.getParteIVA());
}
```

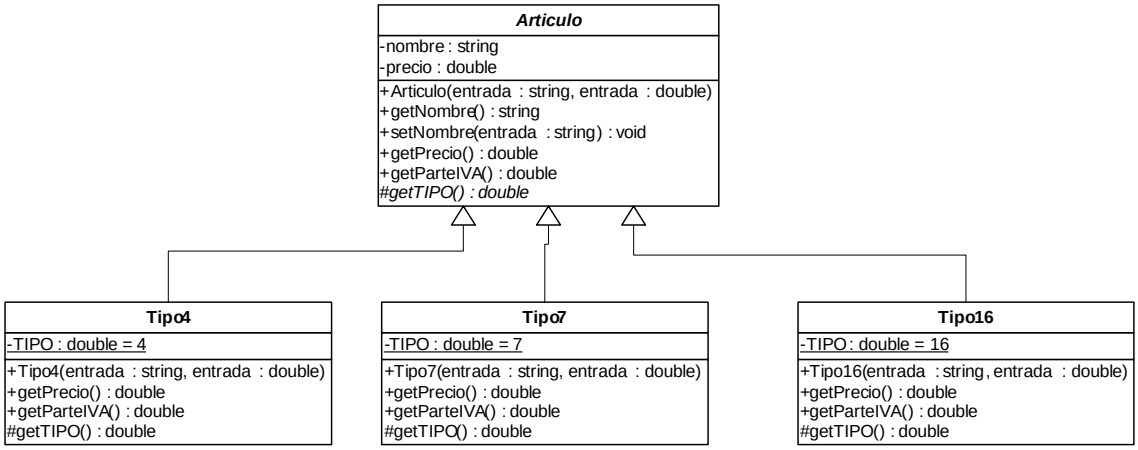
Los siguientes comentarios surgen al observar las anteriores definiciones:

1. La definición del método “getPrecio(): double” en la clase “Articulo” no resulta de especial utilidad, ya que cualquier objeto que utilicemos en nuestra aplicación pertenecerá a una de las clases “Tipo4”, “Tipo7” ó “Tipo16”. Esto resultará más obvio cuando declaremos la clase “*Articulo*” como abstracta y el método “getPrecio(): double” propio de la misma sólo pueda ser accedido desde las subclases (en nuestra aplicación no aparecerán objetos propios de la clase “*Articulo*”)

2. El segundo hecho que podemos resaltar es que el método “getPrecio(): double” ha sido definido en las clases “Tipo4”, “Tipo7” y “Tipo16” del mismo modo, es decir, accediendo al valor del atributo “precio: double” de la clase “Articulo” (a través del método de acceso “getPrecio(): double” de la clase “Articulo”) y sumándole a dicha cantidad el resultado de llamar al método “getParteIVA(): double”. En nuestro nuevo diagrama de clases, al declarar “*getParteIVA(): double*” como método abstracto, este método aparece en a clase (abstracta) “*Articulo*”:



Por tanto, dicho método va a ser visible para los métodos restantes de la clase "*Articulo*", y lo pueden utilizar en sus definiciones. Realizamos ahora una nueva modificación sobre la misma, añadiendo un método abstracto "*getTIPO(): double*", que será definido en "Tipo4", "Tipo7" y "Tipo16" como un método de acceso a la constante de clase "TIPO: double". Este método no tiene por qué ser visible para los usuarios externos de la clase, por lo cual le añadimos el modificador de acceso "protected" (es suficiente con que sea visible en la clase y subclases). Obtenemos el siguiente diagrama de clases UML para nuestra aplicación:



Pero ahora, sobre el diagrama anterior, podemos observar que el método "getPrecio(): double" admite la siguiente definición en la clase abstracta "*Articulo*":

```
//Clase Articulo

public double getPrecio (){
        return (this.precio + this.getParteIVA());
}
```

La anterior definición es válida para las tres clases "Tipo4", "Tipo7" y "Tipo16", ya que tiene en cuenta el precio base de un artículo así como la parte correspondiente a su IVA.

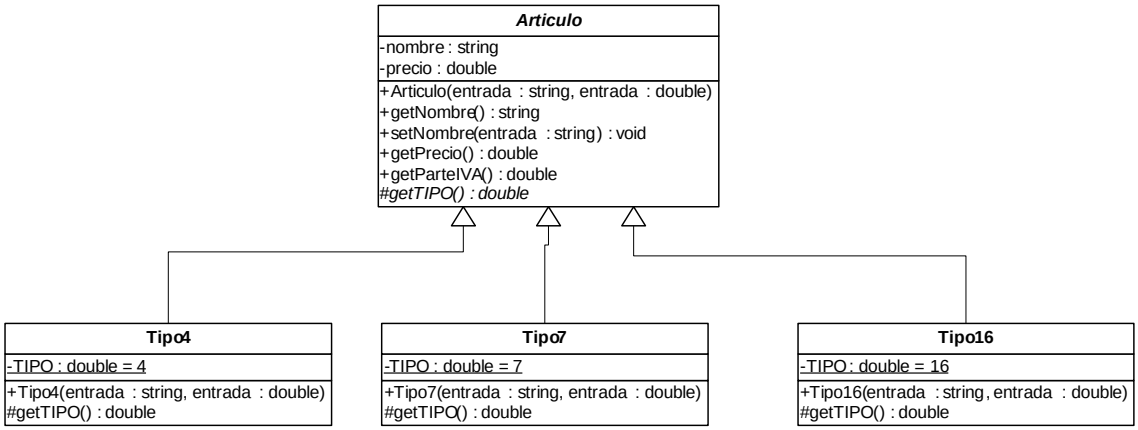
De modo similar, podemos proponer ahora una definición unificada para el método "getParteIVA(): double" (por lo cual deja de ser abstracto) en la propia clase "*Articulo*" que sea válida para las clases "Tipo4", "Tipo7" y "Tipo16":

```
//Clase Articulo

public double getParteIVA (){
        return (this.precio * this->getTIPO() / 100);
```

}

Esta definición del método hace uso de un método abstracto (“getTIPO(): double”) que en la clase “*Articulo*” no ha sido definido, sólo declarado. Esta situación no es “peligrosa” (desde el punto de vista del compilador) siempre y cuando no se puedan construir objetos de la clase “*Articulo*”, ya que para esos objetos no habría un método definido “getTIPO(): double”, pero como ya hemos comentado, el compilador nos asegura que no se puedan construir objetos de la clase “*Articulo*” (sólo de las clases “Tipo4”, “Tipo7” y “Tipo16”).

Por tanto, haciendo uso de las definiciones anteriores de “getParteIVA(): double”, de “getPrecio(): double” y del método “getTIPO(): double”, el nuevo diagrama de clases en UML se podría simplificar al siguiente:



Podemos hacer dos comentarios breves sobre el diagrama anterior:

1. En primer lugar, podemos observar como el número de métodos que aparecen en el mismo (excluyendo los constructores, tenemos 8 métodos, uno de ellos abstracto) es menor que el que aparecía en nuestra versión original del mismo sin hacer uso de clases y método abstractos (en aquel aparecían 9 métodos excluyendo los constructores). Esta diferencia sería aún mayor si tenemos en cuenta que las clases “Tipo4”, “Tipo7” y “Tipo16” anteriormente contenían 2 métodos (“getPrecio(): double” y “getParteIVA(): double”) aparte de los constructores, mientras que ahora sólo aparece uno (“getTIPO(): double”). Cuanto mayor sea el número de clases que hereden de “*Articulo*”, mayor será el número de métodos que nos evitemos de redefinir. Por tanto, hemos simplificado nuestra aplicación, y además hemos conseguido reutilizar código, ya que métodos que antes poseían igual definición (“getPrecio(): double” en “Tipo4”, “Tipo7” y “Tipo16” y “getParteIVA(): double” en “Tipo4”, “Tipo7” y “Tipo16”) ahora han pasado a estar definidos una sola vez. Desde el punto de vista de mantenimiento del código, hemos simplificado también nuestra aplicación.

2. En segundo lugar, el hecho de utilizar métodos y clases abstractas nos ha permitido mejorar el diseño de nuestro sistema de información. Primero, hemos podido hacer que la clase “*Articulo*” fuese definida como abstracta, impidiendo así que se creen objetos de la misma. En segundo lugar, el uso de métodos abstractos (“*getTIPO(): double*”) nos ha permitido cambiar la definición de

algunos otros métodos (finalmente de “getPrecio(): double” y “getParteIVA(): double”), reduciendo el número de métodos en nuestro sistema de información y simplificando el diseño del mismo.

En la Sección siguiente, a la vez que explicamos la necesidad del polimorfismo para poder hacer uso de métodos abstractos en nuestras aplicaciones, aprovecharemos para introducir la notación propia de C++ y Java con respecto al mismo.

4.2 RELACIÓN ENTRE POLIMORFISMO Y MÉTODOS ABSTRACTOS

4.2.1 NECESIDAD DEL POLIMORFISMO PARA EL USO DE MÉTODOS ABSTRACTOS

De las explicaciones y diagramas de clases anteriores en UML se puede extraer una conclusión inmediata. Para poder hacer uso de métodos abstractos, es estrictamente necesaria la presencia de polimorfismo de métodos, o, lo que es lo mismo, de enlazado dinámico de los mismos. Imaginemos que no disponemos de enlazado dinámico (es decir, en enlazado en nuestro compilador es estático, y en tiempo de compilación a cada objeto se le asignan las definiciones de los métodos de los que hará uso). Supongamos que, haciendo uso del último diagrama de clases que hemos presentado en la Sección 4.1.3, realizamos la siguiente declaración (en Java, en C++ debería ser un puntero):

Articulo art1;

La anterior declaración puede ser hecha ya que, aun siendo “*Articulo*” una clase abstracta, se pueden declarar (no construir) objetos de dicha clase.

En condiciones de enlazado estático (de falta de polimorfismo), el compilador le habría asignado al objeto “art1”, independientemente de cómo se construya el mismo, las definiciones de los métodos que se pueden encontrar en la clase “*Articulo*”. Es decir, en el caso del método “*getTIPO(): double*”, se le habría asignado al objeto “art1” un método abstracto, sin definición.

Por tanto, si queremos que nuestra aplicación se comporte de una forma coherente, todos los métodos abstractos deben ser polimorfos (y realizar enlazado dinámico). Conviene recordar ahora que, si bien en Java (como introdujimos en la Sección 3.3), el compilador siempre realiza enlazado dinámico de métodos, y todos los métodos se comportan de modo polimorfo, en C++ (Sección 3.2), para que un método se comporte de manera polimorfa, éste debe ser declarado (en el archivo de cabeceras correspondiente) como “virtual”, y el objeto desde el que se invoque al método debe estar alojado en memoria dinámica (es decir por medio de un puntero o referencia).

Así que todos los métodos que sean declarados en Java y en C++ como abstractos, deberán satisfacer, al menos, los requisitos de los métodos polimorfos. En lo que queda de esta Sección veremos qué más requisitos deben cumplir.

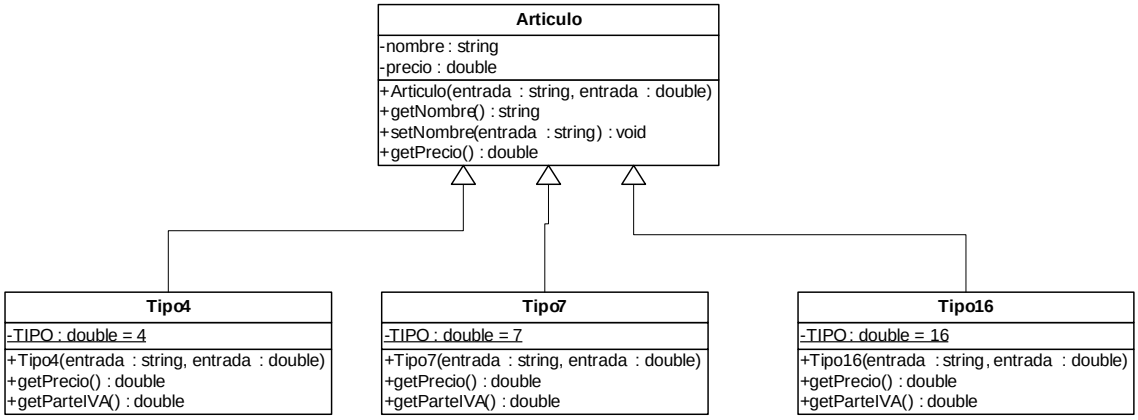
Antes de pasar a ver la sintaxis propia de Java y C++ para definir clases y métodos abstractos, conviene remarcar una diferencia sustancial entre ambos a la hora de considerar una clase como abstracta:

En C++, una clase es abstracta si (y sólo si) contiene al menos un método abstracto. Por ser abstracta, no podremos declarar construir objetos de la misma.

En Java, una clase es abstracta si (y sólo si) contiene en su cabecera el modificador “abstract”. Si contiene algún método abstracto, deberemos declarar también la clase con el modificador “abstract”. Pero, a diferencia de C++, existe la posibilidad de que una clase no contenga ningún método abstracto, y sin embargo sea declarada como abstracta (haciendo uso del modificador “abstract”). De igual modo que en C++, el hecho de ser abstracta implica que no podremos crear objetos de la misma.

4.2.2 SINTAXIS DE MÉTODOS Y CLASES ABSTRACTAS EN C++

Pasemos a ilustrar la sintaxis propia de métodos y clases abstractas en C++. Para ello de nuevo retomamos el ejemplo de la clase “Articulo” y las clases “Tipo4”, “Tipo7” y “Tipo16”. En primer lugar, veamos la codificación del mismo con respecto al siguiente diagrama UML (es decir, sin hacer uso de métodos ni clases abstractas):



El código en C++ correspondiente al diagrama de clases sería el siguiente (omitimos el programa principal “main” por el momento):

```
//Fichero Articulo.h

#ifndef ARTICULO_H
#define ARTICULO_H 1

class Articulo{
    private:
        char nombre [30];
        double precio;
    public:
```

```
            Articulo(char [], double);
            char * getNombre();
            void setNombre(char []);
            virtual double getPrecio();
};

#endif

//Fichero Articulo.cpp

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

using namespace std;

Articulo::Articulo(char nombre [], double precio){
        strcpy (this->nombre, nombre);
        this->precio = precio;
};

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

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

double Articulo::getPrecio(){
      return this->precio;
};

//Fichero Tipo4.h

#ifndef TIPO4_H
#define TIPO4_H

#include "Articulo.h"

class Tipo4: public Articulo{
     private:
            const static double TIPO = 4.0;
     public:
            Tipo4 (char [], double);
            double getPrecio();
            double getParteIVA();
};

#endif
```

```
//Fichero Tipo4.cpp

#include "Tipo4.h"

Tipo4::Tipo4 (char nombre [], double precio): Articulo(nombre, precio){
};

double Tipo4::getPrecio(){
      return (Articulo::getPrecio() + this->getParteIVA());
};

double Tipo4::getParteIVA(){
      return (Articulo::getPrecio() * TIPO / 100);
};

//Fichero Tipo7.h

#ifndef TIPO7_H
#define TIPO7_H

#include "Articulo.h"

class Tipo7: public Articulo{
     private:
            const static double TIPO = 7.0;
     public:
           Tipo7 (char [], double);
           double getPrecio();
           double getParteIVA();
};

#endif

//Fichero Tipo7.cpp

#include "Tipo7.h"

Tipo7::Tipo7 (char nombre [], double precio): Articulo(nombre, precio){
};

double Tipo7::getPrecio(){
      return (Articulo::getPrecio() + this->getParteIVA());
};

double Tipo7::getParteIVA(){
      return (Articulo::getPrecio() * TIPO / 100);
};

//Fichero Tipo16.h
```

```
#ifndef TIPO16_H
#define TIPO16_H

#include "Articulo.h"

class Tipo16: public Articulo{
    private:
        const static double TIPO = 16.0;
    public:
        Tipo16 (char [], double);
        double getPrecio();
        double getParteIVA();
};

#endif

//Fichero Tipo16.cpp

#include "Tipo16.h"

Tipo16::Tipo16 (char nombre [], double precio): Articulo(nombre, precio){
};

double Tipo16::getPrecio(){
    return (Articulo::getPrecio() + this->getParteIVA());
};

double Tipo16::getParteIVA(){
    return (Articulo::getPrecio() * TIPO / 100);
};
```

Las principales peculiaridades que se pueden observar en los ficheros anteriores han sido la necesidad de declarar el método “getPrecio(): double” como “virtual” en el fichero de cabeceras “Articulo.h”, y las llamadas desde unas clases a los métodos de otras (en particular, las llamadas al método “Articulo::getPrecio()” y las llamadas al constructor “Articulo(char [], double)” desde cada uno de los constructores de “Tipo4”, “Tipo7” y “Tipo16”).

La primera modificación que propusimos en la Sección 4.1.1 consistía en definir el método “*getParteIVA(): double*” como abstracto e introducirlo en la clase “*Articulo*” (con lo cual, esta clase pasaría a ser también abstracta). Veamos cómo quedaría tras esas modificaciones el fichero de cabeceras “Articulo.h”:

```
#ifndef ARTICULO_H
#define ARTICULO_H 1

class Articulo{
    private:
        char nombre [30];
        double precio;
```

```
    public:
        Articulo(char [], double);
        char * getNombre();
        void setNombre(char []);
        virtual double getPrecio();
        virtual double getParteIVA()=0;
};

#endif
```

Pasamos a realizar algunos comentarios sobre el archivo de cabeceras "Articulo.h":

1. En primer lugar, debemos destacar que el único archivo que ha sufrido modificación al declarar "*getParteIVA(): double*" como método virtual ha sido el fichero de cabeceras "Articulo.h". Es más, la única modificación que ha sufrido este archivo es que ahora incluye la declaración

"virtual double getParteIVA()=0;"

Esto quiere decir que la clase "*Articulo*" no ha recibido ningún modificador adicional (veremos que esto no es así en Java), y que las clases restantes tampoco. Simplemente debemos observar que las clases que redefinan "getParteIVA(): double" deben incluir en su archivo de cabeceras la declaración del mismo.

2. En segundo lugar, convendría observar más detenidamente la declaración del método:

"virtual double getParteIVA()=0;"

Podemos observar como el mismo incluye el modificador "virtual" que avisa al compilador de que dicho método será redefinido (y que ya introdujimos en la Sección 3.2). Recordamos que la declaración del método deberá ser incluida en todas las clases que lo redefinan (igual que sucede en los diagramas UML y en Java). En segundo lugar, conviene observar la "definición" del método. Como el método es abstracto, y no va a ser definido en esta clase (en el fichero "Articulo.cpp"), se lo advertimos al compilador asignándole la "definición" "double getParteIVA()=0;". El compilador así entiende que este método es abstracto, que por tanto su definición no va a aparecer en el fichero "Articulo.cpp", y que serán las clases derivadas las que se encarguen de definirlo. Veremos cómo en Java la notación es distinta.

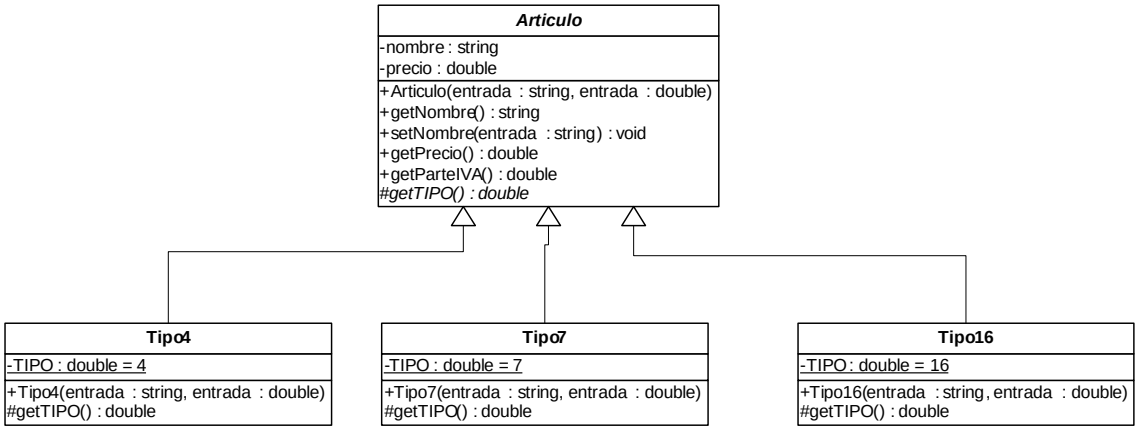
Para concluir, veamos ahora qué sucede cuando intentamos construir un objeto de la clase "*Articulo*":

```
#include <iostream>

#include "Articulo.h"
#include "Tipo4.h"
```

```
#include "Tipo7.h"
#include "Tipo16.h"

using namespace std;

int main (){

   Articulo arti ("La historia interminable", 9);
   arti.getParteIVA();

   Articulo * art1;
   art1 = new Articulo ("La historia Interminable", 9);

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

El anterior fragmento de código produce dos errores de compilación, como ya mencionamos en la Sección 4.1.2, advirtiéndonos de que no podemos construir un objeto de la clase "*Articulo*" (ni tampoco a través de punteros) ya que la misma contiene métodos que son abstractos (en este caso, "*getParteIVA(): double*"), y por tanto es abstracta. El constructor de la clase "*Articulo*" pasa a tener utilidad únicamente para ser invocado desde los constructores de "Tipo4", "Tipo7" y "Tipo16", que siguen haciendo uso del mismo.

Veamos ahora cómo quedaría la implementación en C++ del diagrama de clases en el que introdujimos un nuevo método abstracto "*getTIPO(): double*", y los métodos "getPrecio(): double" y "getParteIVA(): double" pasaron a estar definidos en la clase abstracta "*Articulo*":




```
//Fichero Articulo.h

#ifndef ARTICULO_H
#define ARTICULO_H 1

class Articulo{
    private:
        char nombre [30];
```

```
            double precio;
    public:
            Articulo(char [], double);
            char * getNombre();
            void setNombre(char []);
            double getPrecio();
            double getParteIVA();
    protected:
            virtual double getTIPO() = 0;
};

#endif

//Fichero Articulo.cpp

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

using namespace std;

Articulo::Articulo(char nombre [], double precio){
        strcpy (this->nombre, nombre);
        this->precio = precio;
};

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

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

double Articulo::getPrecio(){
      return this->precio + this->getParteIVA();
};

double Articulo::getParteIVA(){
      return this->precio * this->getTIPO()/100;

};

//Fichero Tipo4.h

#ifndef TIPO4_H
#define TIPO4_H

#include "Articulo.h"

class Tipo4: public Articulo{
```

```
      private:
            const static double TIPO = 4.0;
      public:
            Tipo4 (char [], double);
      protected:
            double getTIPO();
};

#endif

//Fichero Tipo4.cpp

#include "Tipo4.h"

Tipo4::Tipo4 (char nombre [], double precio): Articulo(nombre, precio){
};

double Tipo4::getTIPO(){
      return (this->TIPO);
};

//Fichero Tipo7.h

#ifndef TIPO7_H
#define TIPO7_H

#include "Articulo.h"

class Tipo7: public Articulo{
      private:
            const static double TIPO = 7.0;
      public:
            Tipo7 (char [], double);
    protected:
            double getTIPO();
};

#endif

//Fichero Tipo7.cpp

#include "Tipo7.h"

Tipo7::Tipo7 (char nombre [], double precio): Articulo(nombre, precio){
};

double Tipo7::getTIPO(){
      return (this->TIPO);
};
```

```
//Fichero Tipo16.h

#ifndef TIPO16_H
#define TIPO16_H

#include "Articulo.h"

class Tipo16: public Articulo{
    private:
        const static double TIPO = 16.0;
    public:
        Tipo16 (char [], double);
    protected:
        double getTIPO();
};

#endif

//Fichero Tipo16.cpp

#include "Tipo16.h"

Tipo16::Tipo16 (char nombre [], double precio): Articulo(nombre, precio){
};

double Tipo16::getTIPO(){
    return (this->TIPO);
};
```

Algunos comentarios sobre el código anterior:

1. Conviene resaltar de nuevo la notación específica de C++ para declarar métodos abstractos (en este caso "*getTIPO(): double*"):

"virtual double getTIPO() = 0;"

Un método abstracto debe incluir el modificador "virtual" que nos permitirá acceder a él de modo polimorfo, así como la declaración "double getTIPO() = 0;" advirtiendo de que el mismo es abstracto.

2. En segundo lugar, destacar la definición que de los métodos "getPrecio(): double" y "getParteIVA(): double" hemos podido realizar en el nuevo entorno creado:

```
double Articulo::getPrecio(){
    return this->precio + this->getParteIVA();
};

double Articulo::getParteIVA(){
    return this->precio * this->getTIPO()/100;
```

```
};
```

Vemos que cualquiera de los dos (“getParteIVA(): double” directamente y “getPrecio(): double” indirectamente a través del primero) acceden al método (abstracto) “*getTIPO(): double*”, cuya definición no ha sido dada todavía (y lo será en alguna de las clases derivadas).

Aquí es donde el compilador debe desarrollar la tarea de asegurar que no podamos construir objetos de la clase abstracta “*Articulo*”, ya que para los mismos no habría una definición de “*getTIPO(): double*”, y comprobar que en las subclases que definamos de “*Articulo*” y de las cuales queramos construir objetos, el método “getTIPO(): double” sea definido.

Un programa “main” cliente del anterior sistema de clases podría ser el siguiente:

```
#include <iostream>

#include "Articulo.h"
#include "Tipo4.h"
#include "Tipo7.h"
#include "Tipo16.h"

using namespace std;

int main (){

   Articulo * art1;
   art1 = new Tipo4 ("La historia Interminable", 9);
   Tipo7 * art2;
   art2 = new Tipo7 ("Gafas", 160);
   Articulo * art3;
   art3 = new Tipo16 ("Bicicleta", 550);

   cout << "El precio del primer articulo es " << art1->getPrecio() << endl;
   cout << "El precio del segundo articulo es " << art2->getPrecio() << endl;
   cout << "El precio del tercer articulo es " << art3->getPrecio() << endl;

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

En el mismo podemos observar cómo, si bien tiene sentido declarar objetos (o punteros a objetos) de la clase abstracta “*Articulo*”, los mismos siempre son construidos por medio del constructor de alguna de las clases derivadas de “*Articulo*” en las que todos los métodos son definidos. También es posible observar que, si bien el método “getPrecio(): double” no está redefinido, internamente invoca al método “getTipo(): double” que sí lo está, y por tanto es necesario que haya comportamiento polimorfo del mismo (de ahí la necesidad de declararlo como “virtual” y de invocarlo desde memoria dinámica).

4.2.3 SINTAXIS DE MÉTODOS Y CLASES ABSTRACTAS EN JAVA

Pasamos ahora a mostrar la sintaxis propia de Java para la definición de métodos abstractos y de clases abstractas. Para lo mismo, recuperamos el mismo ejemplo que en la Sección anterior.

Tomamos como punto de partida la definición de las clases “Articulo”, “Tipo4”, “Tipo7” y “Tipo16” antes de incluir en los mismos métodos y clases abstractas, para poder luego observar mejor las diferencias:

```
//Fichero Articulo.java

public class Articulo{

      private String nombre;
      private double precio;

      public Articulo(String nombre, double precio){
                        this.nombre = nombre;
                        this.precio = precio;
       }

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

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

        public double getPrecio (){
                        return this.precio;
         }
}

//Fichero Tipo4.java

public class Tipo4 extends Articulo{

    private static final double TIPO = 4.0;

     public Tipo4(String nombre, double precio){
               super (nombre, precio);
     }

      public double getPrecio (){
               return (super.getPrecio() + this.getParteIVA());
      }

      public double getParteIVA (){
```

```
                return (super.getPrecio() * TIPO / 100);
        }
 }

//Fichero Tipo7.java

public class Tipo7 extends Articulo{

      private static final double TIPO = 7.0;

       public Tipo7(String nombre, double precio){
                        super (nombre, precio);
       }

       public double getPrecio (){
                        return (super.getPrecio() + this.getParteIVA());
       }

       public double getParteIVA (){
                       return (super.getPrecio() * TIPO / 100);
        }
}

//Fichero Tipo16.java

public class Tipo16 extends Articulo{

      private static final double TIPO = 16.0;

       public Tipo16(String nombre, double precio){
                        super (nombre, precio);
       }

       public double getPrecio (){
                        return (super.getPrecio() + this.getParteIVA());
       }

       public double getParteIVA (){
                       return (super.getPrecio() * TIPO / 100);
        }
}
```

Veamos ahora cómo podemos conseguir que el método "*getParteIVA(): double*" pase a estar declarado, como método abstracto, en la clase "*Articulo*", que a consecuencia de este cambio también pasará a ser abstracta:

```
//Fichero Articulo.java

public abstract class Articulo{
```

```
        private String nombre;
        private double precio;

        public Articulo(String nombre, double precio){
                this.nombre = nombre;
                this.precio = precio;
        }

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

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

        public double getPrecio (){
                return this.precio;
        }

        public abstract double getParteIVA();
}
```

Veamos en primer lugar las diferencias con respecto a la definición de la clase "Articulo" que teníamos antes:

1. En primer lugar, el método "getParteIVA(): double" ha sido declarado de la forma:

"public abstract double getParteIVA();"

Podemos observar cómo hemos añadido el modificador "abstract" que nos ha permitido no tener que definir el mismo; sólo hemos declarado su cabecera, sin especificar su comportamiento.

2. En segundo lugar, la clase, al contener un método abstracto, debe ser declarada también como abstracta, por medio del modificador "abstract":

"public abstract class Articulo{...}"

No incluir el modificador "abstract" antes del nombre de la clase habría producido un error de compilación. Conviene recordar que, al incluirlo, estamos previniendo que se construyan objetos de la misma.

Conviene ilustrar también las diferencias de la declaración de métodos y clases abstractas entre Java y C++:

1. Sobre la declaración de métodos abstractos:

Sintaxis propia de Java:

"public abstract double getParteIVA();"

Sintaxis propia de C++:

"virtual double getParteIVA() = 0;"

2. Sobre la declaración de clases abstractas:

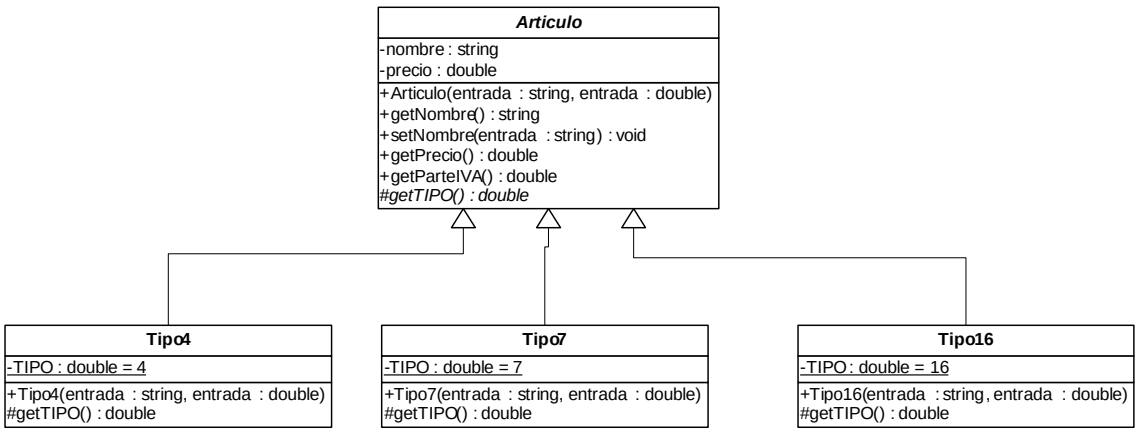
Sintaxis propia de Java:

"public abstract class Articulo{...}"

Sintaxis propia de C++:

"class Articulo{...}"

Veamos ahora cómo quedaría en Java una posible implementación del diagrama de clases que introdujimos en la Sección 4.1.3:



//Fichero Articulo.java

```
public abstract class Articulo{

        private String nombre;
        private double precio;

        public Articulo(String nombre, double precio){
                this.nombre = nombre;
                this.precio = precio;
        }

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

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

```
	public double getPrecio (){
		return this.precio + this.getParteIVA();
	}

	public double getParteIVA(){
		return this.precio * this.getTIPO() / 100;
	}

	protected abstract double getTIPO();
}

//Fichero Tipo4.java

public class Tipo4 extends Articulo{

	private static final double TIPO = 4.0;

	public Tipo4(String nombre, double precio){
		super (nombre, precio);
	}

	protected double getTIPO(){
		return this.TIPO;
	}
}

//Fichero Tipo7.java

public class Tipo7 extends Articulo{

	private static final double TIPO = 7.0;

	public Tipo7(String nombre, double precio){
		super (nombre, precio);
	}

	protected double getTIPO(){
		return this.TIPO;
	}
}

//Fichero Tipo16.java

public class Tipo16 extends Articulo{

	private static final double TIPO = 16.0;

	public Tipo16(String nombre, double precio){
		super (nombre, precio);
```

```
        }

        protected double getTIPO(){
                return this.TIPO;
        }
}
```

Se pueden observar las siguientes características de la implementación anterior:

1. En primer lugar, el método “getTIPO(): double” ha sido declarado como “abstract” en la clase “*Articulo*” por medio de la declaración:

“protected abstract double getTIPO();”

A consecuencia de esto, la clase “*Articulo*” también ha de ser declarada como “abstract”:

“public abstract class Articulo{...}”

2. En segundo lugar, conviene apuntar una vez más que, desde los métodos “getParteIVA(): double” y “getPrecio(): double” propios de la clase abstracta “*Articulo*”, hemos podido hacer uso del método abstracto “*getTIPO(): double*”. Ambos métodos sólo podrán ser usados desde objetos que pertenezcan a clases en las cuales el método abstracto “*getTIPO(): double*” haya sido definido y que no hayan sido declaradas como abstractas.

Conviene recordar que, en Java, el modificador “abstract” puede ser añadido a clases en las que no haya ningún método abstracto, como simple medida de seguridad para prevenir nuestra clase de que sea instanciada (o de que se construyan objetos de la misma) por los usuarios finales. Por ejemplo, si recuperamos la clase “Articulo” tal y como la definimos al principio de esta Sección (es decir, sin contener ningún método abstracto), y le añadimos el modificador “abstract” en la cabecera de la misma:

```
public abstract class Articulo{

        private String nombre;
        private double precio;

        public Articulo(String nombre, double precio){
                this.nombre = nombre;
                this.precio = precio;
        }

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

        public void setNombre (String nuevo_nombre){
```

```
            this.nombre = nuevo_nombre;
        }

        public double getPrecio (){
            return this.precio;
        }
}
```

Se puede comprobar cómo no es posible crear objetos de la misma (obtendremos un error de compilación), a pesar de que todos sus métodos hayan sido declarados y definidos.

4.3 DEFINICIÓN Y VENTAJAS DE USO DE CLASES COMPLETAMENTE ABSTRACTAS O INTERFACES

4.3.1 INTRODUCCIÓN A INTERFACES (EN JAVA) Y CLASES COMPLETAMENTE ABSTRACTAS (EN C++)

Hasta ahora hemos visto ejemplos sencillos es los que un único método en una clase se definía como abstracto y las clases que heredaban de la misma lo definían, pasando así a ser clases concretas. De este modo conseguíamos "enriquecer" la parte visible de una clase e incluso aumentar la reutilización de código, haciendo uso de métodos que no habían sido definidos pero sí declarados.

El concepto anterior se puede extender un poco más allá, dando lugar a "clases completamente abstractas", en las cuales todos los métodos están sin especificar, y que no poseen ni atributos ni constructores. Estas "clases" en realidad no aportan estado ni comportamiento, simplemente declaraciones de métodos o cabeceras, con lo cual su utilidad no va especialmente dedicada a reutilizar código.

En general, este tipo de "clases" se suelen entender más bien como una especie de declaración de un tipo de dato. La "clase" contiene una serie de cabeceras de métodos, y cualquier clase que herede de ella debe especificar la forma concreta de todos y cada uno de los métodos que contenga la clase (no sus declaraciones).

Se podría entender también que estas "clases" constituyen un contrato entre el programador de una clase y su cliente. El cliente le da al programador una "clase" que sólo especifica una serie de métodos, y el programador de la clase debe ser capaz de implementarlos de forma correcta.

Lo que hemos dado en llamar "clases" (o "clases completamente abstractas") en los párrafos anteriores, toma dos formas distintas en Java y en C++. En Java da lugar a la noción de "interface", mientras que en C++ se puede lograr por medio del uso de "clases completamente abstractas".

Definición: una “interface” en Java es un conjunto de métodos relacionados que no tienen definición (es decir, métodos abstractos).

Una “interface” puede contener también declaraciones de constantes, aunque es una práctica que suele ser desaconsejada.

Definición: una “clase completamente abstracta” en C++ es una clase que contiene sólo declaraciones de métodos (es decir, métodos abstractos).

De la definición de “clase completamente abstracta” se puede deducir que una “clase completamente abstracta” sólo poseerá archivo de cabeceras (es decir, un fichero “*.h”).

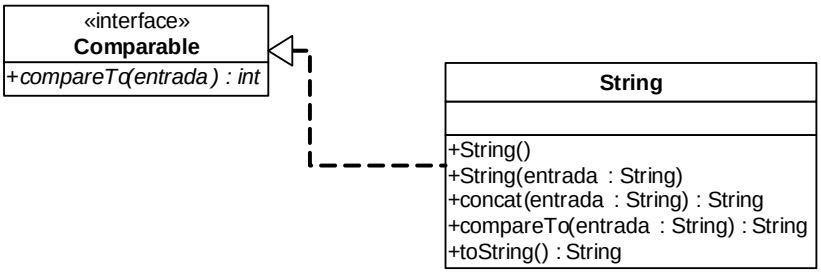
Una pregunta natural ahora sería por qué son necesarios los “interfaces” en Java, cuando el lenguaje contiene ya clases abstractas que puede contener métodos abstractos. En los ejemplos que hemos visto hasta ahora, una clase siempre heredaba de otra única clase. Sin embargo, puede haber situaciones en las que nos interese que una clase herede al mismo tiempo de varias clases distintas. Por ejemplo, una clase “CircunferenciaDibujable” podría heredar al mismo tiempo de una clase “Circunferencia” (que contenga los atributos y métodos necesarios para trabajar con objetos de tipo “Circunferencia”) y de otra clase “Dibujable” (que contuviera métodos que permitan dibujar un objeto). La herencia múltiple no está permitida en Java (sí lo está en C++, pero por ejemplo tampoco lo está en C#). Una de las razones para que no esté permitida la herencia múltiple en Java es que, si una clase pudiera heredar un mismo método definido en diversas clases de forma distinta, ¿qué definición (o comportamiento) del mismo debería heredar? Para evitar esa situación, una clase en Java sólo puede heredar de otra clase (sólo hay herencia simple).

Sin embargo, una clase en Java no “hereda” de una “interface”. En realidad, una “interface” no contiene ni atributos ni comportamiento, y por tanto sus propiedades no son “heredadas”. En terminología Java, se dice que una clase “implementa” una “interface”, es decir, “implementa” la lista de métodos que contiene dicha “interface” (pero, insistimos, no hereda de dicha “interface”).

Java sí permite que una clase “implemente” múltiples “interfaces”, ya que al no tener éstas definidos sus métodos, se puede dar la situación en que una misma clase implemente dos “interfaces” que contienen un mismo método, pero como ninguna de las dos copias del método ha sido todavía definida, la definición válida del mismo será la provista en la clase.

Por tanto, una de las razones para usar “interfaces” en Java (en lugar de clases abstractas) es que una misma clase puede implementar varios interfaces al mismo tiempo. Como C++ permite herencia múltiple, este comportamiento también puede ser conseguido en C++ simplemente heredando de varias clases que sean completamente abstractas.

Veamos lo anterior con un sencillo ejemplo de la propia API de Java. Observamos primero cómo se representa en UML una “interface” así como la relación de “implementación” (no de herencia) correspondiente:



La clase “String” es una clase propia de la librería de Java con la que ya hemos trabajado que nos permite representar cadenas de caracteres (http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html). Si observamos la especificación de la misma en la página web anterior, podremos observar cómo dicha clase “implementa” la “interface” “Comparable”. En notación UML, una “interface” se denota de modo similar a una clase, pero con el calificativo “<<interface>>” sobre el nombre para poder distinguirla. Su lista de métodos aparecerá siempre en cursiva, ya que todos ellos deben ser abstractos.

En este caso, la “interface” “Comparable” sólo contiene un único método, de nombre “compareTo(T): int” (http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Comparable.html). El tipo de dato “T” que admite como parámetro es lo que se conoce en Java como un tipo genérico. Por el momento no vamos a explicar su significado, pero debemos ser capaces de trabajar con las clases de la API que hacen uso del mismo.

Las relaciones de implementación de una “interface” por parte de una clase se denotan por medio de una flecha (igual que las relaciones de herencia) pero en la cual la línea está punteada. Como podemos observar en el diagrama anterior, el hecho de que la clase “String” implemente la “interface” “Comparable” quiere decir que debe definir el método “compareTo(String): int”, que como podemos observar aparece en el diagrama UML de la misma (en caso contrario, el método “*compareTo(T): int*” seguiría siendo abstracto en la clase “String” y por tanto la clase sería abstracta, que no es el caso ya que el nombre de la misma no aparece en cursiva en el diagrama UML).

Veamos ahora la especificación que del método “*compareTo(T): int*” se hace en API de Java, tanto en la “interface” “Comparable” como la que se da del mismo método, ya definido, (“compareTo(String): int”) en la clase “String”:

En la interface “Comparable” encontramos la siguiente especificación del método,que todavía es abstracto (http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Comparable.html#compareTo(T)):

“Compara este objeto con el objeto especificado como parámetro para comprobar su orden. Devolverá un entero negativo, un cero, o un entero positivo si el objeto es menor que, igual a, o mayor que el objeto especificado”

(Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.)

En la clase “String” cuando ya el método “comparable(String): int” es definido, la especificación del mismo dice (http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html#compareTo(java.lang.String))):

“Compara dos cadenas lexicográficamente. La comparación está basada en el valor Unicode de cada carácter en las cadenas ... El resultado es un entero negativo si el objeto “this” lexicográficamente precede al argumento. El resultado es un entero negativo si el objeto “this” lexicográficamente sigue al argumento. El resultado es cero si ambas cadenas son iguales; ...”

(Compares two strings lexicographically. The comparison is based on the Unicode value of each character in the strings. The character sequence represented by this String object is compared lexicographically to the character sequence represented by the argument string. The result is a negative integer if this String object lexicographically precedes the argument string. The result is a positive integer if this String object lexicographically follows the argument string. The result is zero if the strings are equal; compareTo returns 0 exactly when the equals (Object) method would return true.)

Como puedes observar, el comportamiento de la función “compareTo(String): int” es similar al que obtendrías con la función C++ “strcmp”.

Lo que pretendíamos ilustrar con el anterior ejemplo es que, si bien en la “interface” “Comparable” la especificación del método “*compareTo(T): int*” era todavía “abstracta”, en el sentido de que sólo se indicaba cómo debía comportarse tal método de un modo genérico, en la clase “String” el mismo ya ha recibido una definición “concreta”, ya que su comportamiento queda completamente especificado.

Esto nos permite entender mejor la diferencia entre el tipo de funciones que debe cumplir una “interface”, es decir, facilitar una lista de métodos que luego las clases deben definir, y las funciones que debe definir una clase “concreta”, que debe definir todos y cada uno de los métodos que desee implementar.

Veamos ahora alguno de los usos adicionales de las “interfaces” (o de las “clases completamente abstractas” de C++).

Una “interface” permite la declaración de objetos de la misma. Veamos un ejemplo sobre el diagrama de clases anterior:

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

        Comparable c1, c2;
        c1 = new String ("caballo");
        c2 = new String ("pellejo");
```

```
        System.out.println ("El resultado de comparar " + c1 + " con " + c2 + " es
" + c1.compareTo(c2));
}
```

Vemos cómo en el fragmento anterior de código hemos podido declarar tanto “c1” como “c2” como objetos de la “interface” “Comparable”. Posteriormente los hemos podido construir como objetos, por ejemplo, de la clase “String”, ya que “String” es una clase que implementa “Comparable”. Como ambos objetos han sido declarados de la “interface” “Comparable”, ahora podemos invocar sobre cualquiera de ellos al método “compareTo(String): int”.

El resultado de ejecutar el anterior código sería (por ejemplo):

El resultado de comparar caballo con pellejo es -13

El resultado nos está diciendo que la cadena “caballo” es menor, con respecto al orden lexicográfico, que la cadena “pellejo”, ya que el valor devuelto por el método es menor que 0.

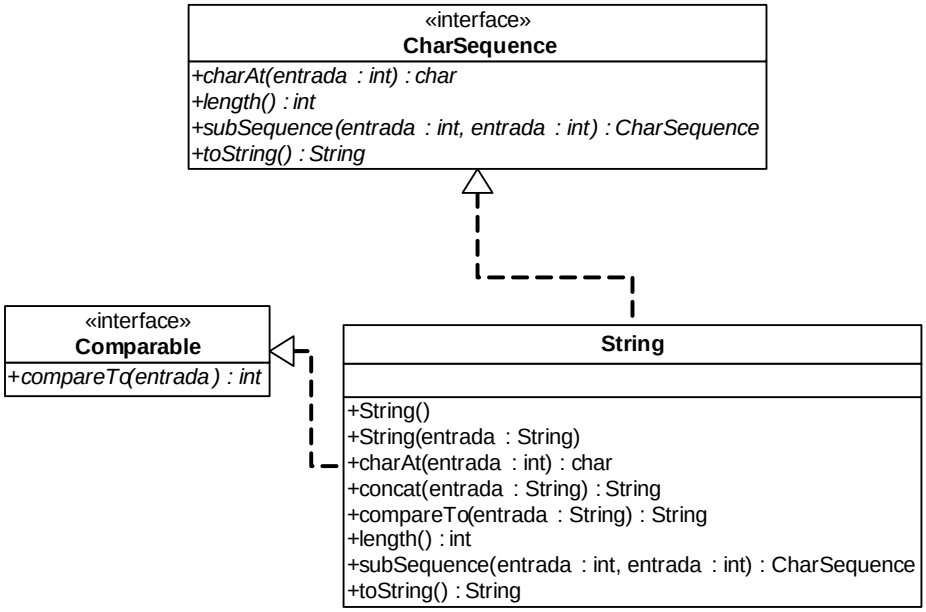
Por cierto, el haber declarado “c1” y “c2” como de la “interface” “Comparable” quiere decir que ambos objetos poseen únicamente el método “compareTo(String): int” y a los propios de la clase “Object”. Por ejemplo, observa lo que pasa al intentar invocar al método “concat(String): String” de la clase “String” sobre cualquiera de los objetos:

```
//c1.concat(c2);
//La orden provoca un error de compilación, ya que c1 es de tipo “Comparable”
```

Lo anterior nos sirve para remarcar una vez más la importancia de declarar un objeto de una clase o “interface”, ya que esto decide el conjunto de métodos que vamos a poder utilizar sobre el mismo.

De igual modo que podemos declarar objetos de una “interface”, se puede utilizar una “interface” para declarar estructuras genéricas (como “arrays”) o también para declarar el tipo de los argumentos que se pasan a una función o método.

Un último punto al que deberíamos prestar atención es al hecho de que, en Java, una misma clase puede implementar distintas “interfaces” (mientras que no puede heredar de distintas clases, situación que sí es posible en C++). Si observamos de nuevo la especificación en la API de Java de la clase “String” (http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html), podemos ver cómo la misma implementa la “interface” “Comparable”, y también hace lo propio con “CharSequence” (y “Serializable”, de la que no nos ocuparemos ahora), por lo que el diagrama anterior UML de la clase “String” se podría ahora mostrar como:



Vemos que la "interface" "CharSequence" declara algunos métodos (abstractos) útiles para calcular la longitud de una cadena ("*length(): int*"), capturar subsecuencias de una cadena ("*subsequence(int, int): CharSequence*"), devolver una carácter en una posición determinada ("*charAt(int): char*") y convertir un objeto a una cadena ("*toString(): String*"). Puedes encontrar más detalles sobre la misma en http://java.sun.com/j2se/1.5.0/docs/api/java/lang/CharSequence.html).

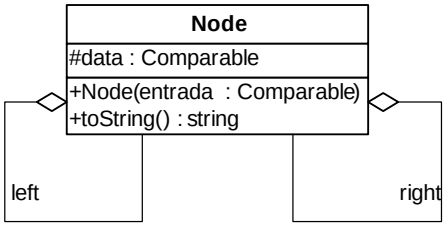
La clase "String", al implementar dicha "interface", debe dar una definición concreta de los mismos (salvo que sea una clase abstracta, que no es el caso).

Por tanto, por el simple hecho de saber que la clase "String" implementa las "interfaces" "Comparable" y "CharSequence", conocemos ya parte de los métodos que tendrá la misma. En general, los "interfaces" nos proveen de una forma muy útil de organizar información, ya que sabiendo qué "interfaces" implementa una clase, conocemos algunos de los métodos que debe contener.

4.3.2 SINTAXIS DE INTERFACES EN JAVA; UN EJEMPLO DESARROLLADO

Para introducir la sintaxis propia de Java para el uso de interfaces vamos a introducir un ejemplo de desarrollo basado en árboles binarios de búsqueda.

Un árbol binario es una estructura formada por nodos que poseen un dato (al que nos referiremos por "data"), y que a su vez contienen dos subnodos (implementados por medio de una relación de agregación), uno llamado "left" y otro llamado "right". El tipo de dato "Node" (nodo) se podría representar por el siguiente diagrama UML:



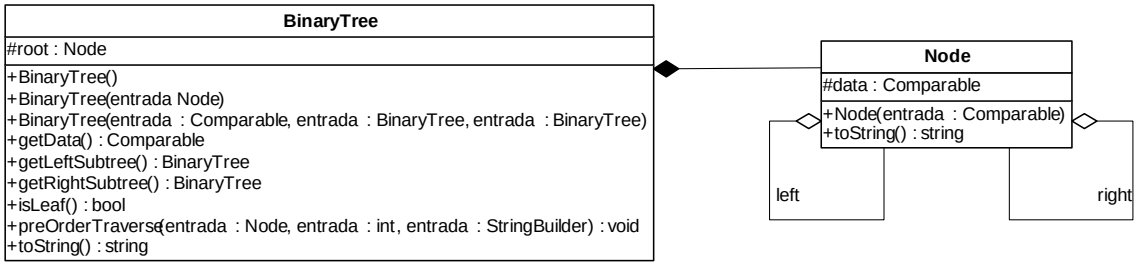
Como puedes observar, el valor de “data” (declarado “protected”) ha sido declarado como de tipo “Comparable”, correspondiente con la “interface” Java que nos permite comparar objetos. Vemos aquí una de las utilidades propias de los “interfaces” en Java, que es la de usarlos para declarar tipos, al igual que podíamos hacer con las clases o con las clases abstractas (aunque, como ya sabemos, ni de las “interfaces” ni de las clases abstractas se puedan crear objetos).

Una posible representación en Java para dicho diagrama UML sería la siguiente:

```
//Fichero Node.java

public class Node{

        protected Comparable data;
        protected Node left;
        protected Node right;

        public Node (Comparable data){
                this.data = data;
                this.left = null;
                this.right = null;
        }

        public String toString (){
                return this.data.toString();
        }
}
```

Como puedes observar, los tres atributos de la misma han sido declarados como “protected”, lo cual quiere decir que serán accesibles desde las subclases (y desde el mismo “pakage”). De eso modo, en la clase “Node” hemos podido evitar usar métodos de acceso y modificación de los atributos de la misma. En general esta práctica no es muy recomendable, ya que no preserva el principio de ocultación de la información que introdujimos en el Tema 1; la razón para haberla usado en este ejemplo es simplificar en parte el código resultante del mismo.

A partir del tipo de dato anterior “Node”, y por medio de ir enlazando objetos del mismo tipo, representaremos un árbol binario (en la clase “BinaryTree”). Pasemos a ver ahora la clase “BinaryTree” que podemos construir a partir de la clase “Node” anterior:



Podemos ver como la clase “BinaryTree” se puede especificar por medio de una relación de composición a partir de la clase “Node”. La misma contiene un único atributo, el objeto de la clase “Node” utilizado como raíz (o “root”) del árbol (y declarado de nuevo de tipo protegido). Además, contiene tres constructores, y luego métodos de acceso al valor de “data” en el “root” (“getData(): Comparable”), así como métodos de acceso a los dos “árboles” que cuelgan de “root”, “getLeftSubtree(): BinaryTree” y “getRightSubtree(): BinaryTree”. El método “isLeaf(): bool” es un predicado que nos permite saber si de “root” no cuelgan más nodos ni a izquierda ni a derecha. El método “preOrderTraverse(Node, int, StringBuider): void” lo que hace es un recorrido del árbol, de forma recursiva, tanto a izquierda como a derecha, a la vez que va acumulando en el parámetro de tipo “StringBuilder” una cadena que contenga todos los datos que contiene el árbol. El método “toString(): String” básicamente invoca a “preOrderTraverse(Node, int, StringBuider): void” y devuelve el valor de “StringBuilder”, convertido a “String” por pantalla.

Hablaremos de la clase “StringBuilder” tras introducir el código propio de la clase “BinaryTree”. Una posible representación en Java de la misma sería:

```
public class BinaryTree{

      protected Node root;

       public BinaryTree(){
          this.root = null;
       }

       public BinaryTree(Node root){
            this.root = root;
       }

public BinaryTree(Comparable data, BinaryTree leftTree, BinaryTree rightTree){
       this.root = new Node(data);
       if (leftTree!=null){
             root.left = leftTree.root;
       }
       else {
             root.left = null;
       }
       if (rightTree!=null){
             root.right = rightTree.root;
```

```
        }
        else {
            root.right = null;
        }
}

    public Comparable getData(){
        return this.root.data;
    }

    public BinaryTree getLeftSubtree(){
        if (root != null && root.left !=null){
            return new BinaryTree (root.left);
        }
        else {
            return null;
        }
    }

    public BinaryTree getRightSubtree(){
        if (root != null && root.right !=null){
            return new BinaryTree (root.right);
        }
        else {
            return null;
        }
    }

    public boolean isLeaf(){
       return (root.left == null && root.right == null);
    }

    private void preOrderTraverse (Node node, int depth, StringBuilder sb){
        for (int i = 1; i < depth; i++){
            sb.append(" ");
        }
        if (node == null){
            sb.append ("null\n");
        }
        else {
            sb.append (node.toString());
            sb.append ("\n");
            this.preOrderTraverse (node.left, depth + 1, sb);
            this.preOrderTraverse (node.right, depth + 1, sb);
        }
    }

    public String toString(){
      StringBuilder sb = new StringBuilder ();
       preOrderTraverse (root, 1, sb);
```

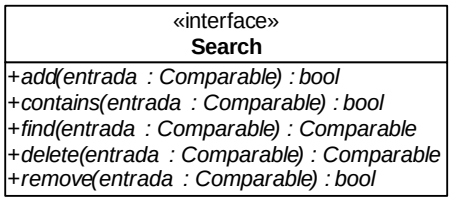
```
            return sb.toString();
        }
}
```

Puedes encontrar información relativa a la clase “StringBuilder” en http://java.sun.com/j2se/1.5.0/docs/api/java/lang/StringBuilder.html. La principal ventaja de trabajar en este caso con “StringBuilder” en lugar de trabajar con “String” es que los objetos de la primera son mutables, y sobre un mismo objeto podemos ir añadiendo nuevos fragmentos de cadena por medio del método “append(String): StringBuilder” (cosa que no resulta posible sobre un objeto de tipo “String”, que es inmutable). Una vez hemos terminado de operar con el objeto de tipo “StringBuilder”, podemos convertirlo en un objeto de la clase “String” por medio del método “toString(): String”.

El comportamiento del resto de métodos de la clase “BinaryTree” se puede entender con una lectura atenta de los mismos.

Si observas los métodos que hemos introducido hasta ahora para trabajar con la clase “BinaryTree”, verás que los mismos sólo permiten construir objetos de la clase, acceder (métodos de acceso) a sus distintos atributos, y escribir un objeto por pantalla (ni siquiera hemos definido métodos que permitan añadir o suprimir objetos del árbol).

Queremos ampliar ahora la funcionalidad de la clase anterior con ciertas características adicionales, que son las que vamos a especificar en la siguiente “interface” “Search”:



Como puedes observar, la “interface” anterior es lo suficientemente genérica para que sea útil en nuestro caso de uso, árboles binarios, así como en otro tipo de estructuras de datos, como pueden ser listas enlazadas, conjuntos, … .

Ésta suele ser una de las funciones características de las “interfaces”: proveernos de una interfaz de uso para nuestras clases (en este caso, la clase que implemente la “interface” “Search”) suficientemente genérica como para que sea de utilidad en diversos contextos. De este modo, nos podremos encontrar con diversos tipos de datos de características dispares que comparten una misma “interface” (y por tanto, podremos trabajar con ellos de forma similar).

Por tanto, partiendo de la estructura “BinaryTree” previamente definida, para implementar la “interface” “Search”, deberemos implementar (o definir) los métodos señalados en la misma. Su comportamiento debería ser el siguiente:

a) “add(Comparable): bool”, que permita introducir un nuevo objeto en un árbol en el lugar que le corresponda. Debe devolver “verdadero” si el objeto no estaba, y “falso” si el objeto se encontraba en el mismo.
b) “contains(Comparable): bool”, que nos permita saber si un árbol contiene un objeto.
c) “find(Comparable): Comparable” que devuelve una referencia al nodo que se pasa como argumento (si está en el árbol), y en caso contrario devuelve “null” (la referencia al objeto nulo).
d) “delete (Comparable): Comparable” que elimina el objeto que se pasa como atributo (si está en el árbol) y lo devuelve. En caso contrario devuelve “null”.
e) “remove(Comparable): bool” que elimina del árbol el objeto que se pasa como argumento y devuelve “verdadero”. Si no lo encuentra, devuelve “falso”.

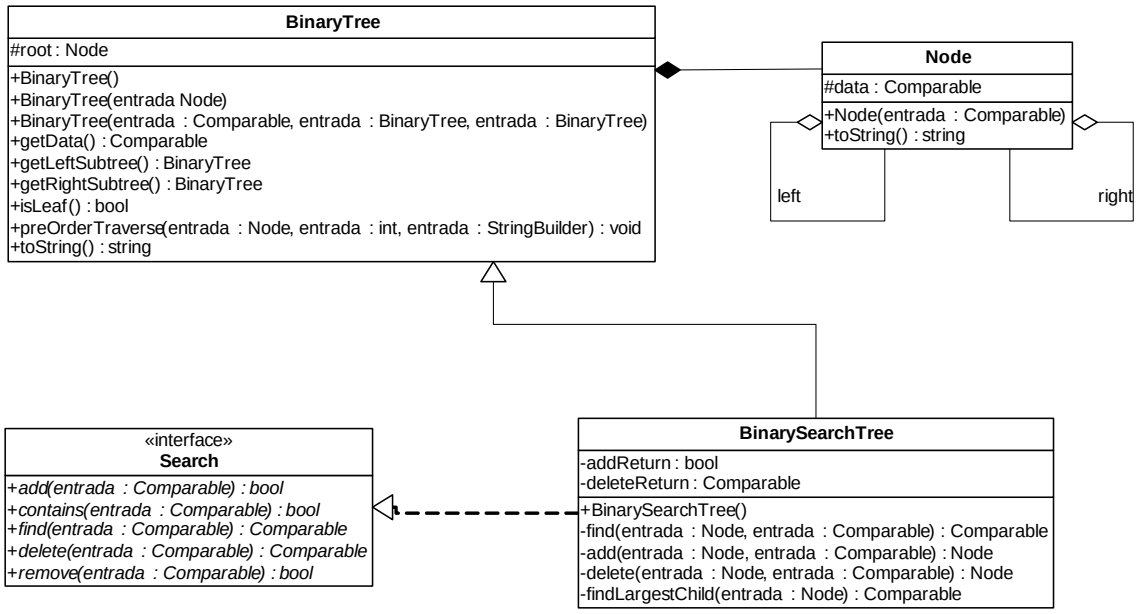
Veamos el aspecto que tendría ahora en Java la anterior “interface” “Search”:

```
public interface Search{
        public boolean add (Comparable item);
        public boolean contains (Comparable target);
        public Comparable find (Comparable target);
        public Comparable delete (Comparable target);
        public boolean remove (Comparable target);
}
```

La sintaxis Java para “interfaces” hace uso de la palabra clave “interface”. Como se puede observar, la misma puede añadir el modificador de visibilidad correspondiente (en este caso, “public”, o la opción por defecto, package, que se omite). Los modificadores “protected” y “private” no pueden ser utilizados en una “interface”, dando lugar a un error de compilación.

Dentro de una “interface”, como ya hemos señalado con anterioridad, sólo se pueden encontrar declaración y definición de constantes de clase (aunque no sea recomendable hacerlo), y declaraciones de métodos. Por defecto, los métodos que se encuentran en una “interface” son “public” (aunque lo podamos escribir de nuevo como en el ejemplo anterior) y también son “abstract”, lo cual quiere decir que ningún método en una “interface” puede tener definición; conviene recordar que, en Java, los métodos de una clase (abstracta o no), por defecto, tienen modificador de acceso package. La declaración de los métodos, por lo demás, es similar a la que se realiza en una clase abstracta. Aunque podemos utilizar los modificadores “abstract” y “public” aplicados a todos los métodos declarados en una “interface”, no es necesario hacerlo, serán considerados así.

Ahora definiremos una clase que sea capaz de heredar de “BinayTree”, y que además contenga una definición de los métodos que hemos propuesto en la anterior “interface” “Search”. Llamaremos a esa clase “BinarySearchTree”. Su especificación en UML, con la de las clases de las que hereda y los interfaces que implementa, sería:



Observemos primero las relaciones que parten de la clase “BinarySearchTree”. En primer lugar, “BinarySearchTree” posee una relación de herencia con respecto a la clase “BinaryTree”. Esto quiere decir que “BinarySearchTree” contiene todos y cada uno de los métodos de la clase “BinayTree”. Además, podemos observar cómo en este caso no redefine ninguno de ellos. Conviene recordar que los constructores de la clase derivada (“BinarySearchTree()”) deberán invocar a alguno de los constructores de la clase base.

Por otra parte, la clase “BinarySearchTree” también está relacionada con la “interface” “Search”, por medio de lo que se conoce como una relación de implementación. “BinarySearchTree” implementa la “interface” “Search”. Esto quiere decir que “BinarySearchTree” contiene todos los métodos abstractos propios de la “interface” “BinaryTree”. Como en este caso “BinarySearchTree” no es una clase abstracta, la misma deberá definir todos los métodos de la “interface”. Aparte de los métodos heredados y los que provienen de la relación de implementación, vemos que la clase “BinarySearchTree” define una serie de métodos auxiliares (de ahí que los defina como “private”) que son los siguientes:

a) “find(Comparable, Node): Comparable” es un método recursivo en el que se va a apoyar “find(Comparable): Comparable” para encontrar un objeto en un árbol.
b) “add(Comparable, Node): Comparable” es un método recursivo en el que se apoya “add(Comparable): Comparable” para introducir un objeto en el árbol.
c) “delete(Comparable, Node): Comparable” es un método recursivo en el que se apoya “delete(Comparable): Node” para eliminar un objeto del árbol.
d) “findLargestChild(Node): Comparable” es un método recursivo que ayuda a encontrar el hijo “mayor” de un nodo. Lo utilizaremos al hacer uso del método “delete(Comparable, Node): Comparable” anterior.

Por tanto, la clase “BinarySearchTree” debe ser capaz de dar una implementación de todos los métodos de la “interface” “Search”, y de los que acabamos de nombrar.

Veamos qué aspecto tiene ahora la cabecera de dicha clase:

public class BinarySearchTree extends BinaryTree implements Search{…}

Vemos cómo la clase “BinarySearchTree” define la relación de herencia, por medio del comando “extends”, y cómo también define la relación de implementación con respecto a “Search”, por medio del comando “implements”.

Conviene recordar ahora que, si bien una clase en Java sólo puede heredar de otra clase, puede implementar tantas “interfaces” como se quiera.

Veamos ahora el código de la clase completa. La clase “BinarySearchTree” está programada de tal modo que no puede haber dos objetos “Node” en la misma que contengan el mismo “data”. Además, para poder optimizar las funciones de búsqueda, sobre el mismo se ha elegido la siguiente ordenación. Dado un “Node” cualquiera dentro de “BinarySearchTree”, el objeto “Node” “left”, si lo hay, siempre será menor (con respecto al método “compareTo(entrada): int”) que el valor de “data”, y el objeto “Node” “right”, si lo hay, siempre será mayor, con respecto al método “compareTo(entrada): int” que el valor de “data”. Esta ordenación debe ser tenida en cuenta a la hora de insertar nodos en un objeto “BinarySearchTree”, y nos será de utilidad en las operaciones de búsqueda e inserción sobre el mismo.

```
public class BinarySearchTree extends BinaryTree implements Search{
	private boolean addReturn;
	private Comparable deleteReturn;

	public BinarySearchTree(){
		super();
	}

	public Comparable find (Comparable target){
		return find (root, target);
	}

	private Comparable find(Node localRoot, Comparable target){
		if (localRoot == null){
			return null;
		}
		int compResult = target.compareTo(localRoot.data);
		if (compResult == 0){
			return localRoot.data;
		}
		else if (compResult < 0){
			return find (localRoot.left, target);
		}
		else return find (localRoot.right, target);
	}
```

```
public boolean contains(Comparable target){
	Comparable aux = this.find (target);
	if (aux == null){
		return false;
	}
	else {
		return true;
	}
}

public boolean add (Comparable item){
	root = add (root, item);
	return addReturn;
}

private Node add (Node localRoot, Comparable item){
	if (localRoot == null){
		addReturn = true;
		return new Node (item);
	}
	else if (item.compareTo (localRoot.data) == 0){
		addReturn = false;
		return localRoot;
	}
	else if (item.compareTo (localRoot.data) < 0){
		localRoot.left = add (localRoot.left, item);
		return localRoot;
	}
	else {
		localRoot.right = add (localRoot.right, item);
		return localRoot;
	}
}

public Comparable delete (Comparable target){
	root = delete (root,target);
	return deleteReturn;
}

private Node delete (Node localRoot, Comparable item){
	if (localRoot == null){
		deleteReturn = null;
		return localRoot;
	}
	int compResult = item.compareTo (localRoot.data);
	if (compResult < 0){
		localRoot.left = delete (localRoot.left, item);
		return localRoot;
	}
	else if (compResult > 0){
```

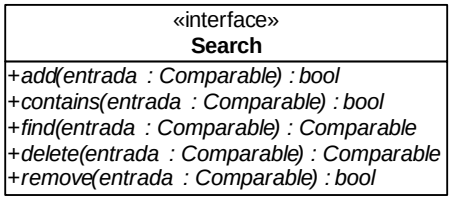
```
                localRoot.right = delete(localRoot.right, item);
                return localRoot;
            }
            else {
                deleteReturn = localRoot.data;
                if (localRoot.left == null){
                    return localRoot.right;
                }
                else if (localRoot.right == null){
                    return localRoot.left;
                }
                else {
                    if (localRoot.left.right == null){
                        localRoot.data = localRoot.left.data;
                        localRoot.left = localRoot.left.left;
                        return localRoot;
                    }
                    else {
                        localRoot.data = findLargestChild(localRoot.left);
                        return localRoot;
                    }
                }
            }
        }

        private Comparable findLargestChild(Node parent){
            if (parent.right.right == null){
                Comparable returnValue = parent.right.data;
                parent.right = parent.right.left;
                return returnValue;
            }
            else {
                return findLargestChild(parent.right);
            }
        }

        public boolean remove (Comparable target){
            Comparable aux = delete (target);
            if (aux == null){
                return false;
            }
            else{
                return true;
            }
        }
}
```

Algunos comentarios se pueden hacer sobre el código anterior:

1) En primer lugar, se puede observar cómo hemos reducido el número de constructores de la clase “BinaryTree” (que eran 3) a la clase “BinarySearchTree”, donde sólo hemos definido uno, el vacío. El hecho de que sólo haya un constructor y además construya un árbol vacío nos obliga a ir incluyendo objetos en los árboles de la clase “BinarySearchTree” exclusivamente a través del método “add(Comparable): boolean”, que, si hace la inserción de forma adecuada, nos asegura que todos los árboles de la clase “BinarySearchTree” que podamos crear estarán ordenados (con respecto al método “compareTo(String): int”, que sigue el orden lexicográfico).

2) En segundo lugar, se puede mencionar que el atributo de la clase “addReturn: bool” es un atributo auxiliar que se utiliza en el método “add(Node, Comparable): Node” para guardar el booleano que indica si un objeto ya existía en el árbol o no. Igualmente, “deleteReturn: bool”, se utiliza en el método “delete (Node, Comparable): Node”, y guarda una copia del objeto que debe ser devuelto por el método “delete(Comparable): Comparable”.

3) En tercer lugar, convendría observar cómo en la implementación de la clase “BinarySearchTree” hemos definido todos y cada uno de los métodos propios de la “interface” “Search”:



En caso contrario, la clase obtenida hubiera sido abstracta (y deberíamos haber utilizado el modificador “abstract” en su cabecera). Puedes notar también como todos los métodos de la “interface” han aparecido en la clase “BinarySearchTree” con el modificador “public”, el mismo que tenían en la “interface”. En caso contrario, hubiésemos obtenido un error de compilación.

3) También convendría observar cómo funcionan cada uno de los métodos auxiliares que hemos debido programar en la clase “BinarySearchTree”, que hemos definido como “private”. Por ejemplo, nos detendremos en el método “add(Node, Comparable): Comparable”:

```
private Node add (Node localRoot, Comparable item){
        if (localRoot == null){
                addReturn = true;
                return new Node (item);
        }
        else if (item.compareTo (localRoot.data) == 0){
                addReturn = false;
                return localRoot;
        }
        else if (item.compareTo (localRoot.data) < 0){
                localRoot.left = add (localRoot.left, item);
                return localRoot;
```

```
            }
            else {
                  localRoot.right = add (localRoot.right, item);
                  return localRoot;
            }
}
```

Este método recibe como parámetros un nodo “localRoot” y un objeto “item” que debe ser añadido en su posición correspondiente con respecto al orden lexicográfico. El proceso para añadirlo es como sigue:

1) Si hemos llegado a una hoja vacía del árbol, añadimos un nuevo nodo con el objeto y en al atributo auxiliar ponemos “true”.

2) Si estamos en un nodo que tiene un “data” distinto de “null”:

a) Si ese “data” es igual que nuestro objeto, el objeto ya está en el árbol, y no debemos añadirlo (y devolvemos “false”).
b) Si nuestro objeto es menor que “data” (con respecto a “compareTo(String): int”), llamamos de nuevo al método “add(Node, Comparable): Node” en el subárbol de la izquierda (que es donde se encuentran los objetos menores que “data”).
c) En caso contrario, debemos intentar añadir el objeto en el subárbol de la derecha, por medio de “add(Node, Comparable): Node”.

De modo similar funcionan los métodos auxiliares “find(Node, Comparable): Comparable” y “delete(Node, Comparable): Node” (aunque “delete(Node, Comparable): Node” requiera algo más de complicación, ya que cuando elimina un nodo debe tener en cuenta quién ocupará dicho lugar, lo que determina por medio del método “findLargestChild(Node): Comparable”).

Veamos ahora un programa principal que puede actuar sobre el anterior diagrama de clases. En el mismo vamos a crear un objeto de la clase “BinarySearchTree” vacío. Por medio del comando “add(Node): bool” vamos a incluir una secuencia de palabras (se supone que nuestro método “add(Node): bool” hará que las mismas estén ordenadas con respecto al orden lexicográfico, de manera que el dato de la rama izquierda de un árbol siempre sea menor que su “padre”, y el dato de la rama derecha mayor que el mismo). Por último, comprobaremos si algunas de ellas están en el árbol, eliminaremos alguna, e imprimiremos el mismo para ver cómo lo ha ordenado nuestro algoritmo de inserción:

```
public class principal_binarysearchtree{

      public static void main(String [] args){

      BinarySearchTree arbol = new BinarySearchTree();
      System.out.println (arbol.toString());

      arbol.add ("Es");
      arbol.add ("algo");
```

```
        arbol.add ("formidable");
        arbol.add ("que");
        arbol.add ("vio");
        arbol.add ("la");
        arbol.add ("vieja");
        arbol.add ("raza");
        arbol.add ("robusto");
        arbol.add ("tronco");
        arbol.add ("de");
        arbol.add ("arbol");
        arbol.add ("al");
        arbol.add ("hombro");
        arbol.add ("de");
        arbol.add ("un");
        arbol.add ("campeon");
        arbol.add ("salvaje");
        arbol.add ("y");
        arbol.add ("aguerrido");
        arbol.add ("cuya");
        arbol.add ("fornida");
        arbol.add ("maza");
        arbol.add ("blandiera");
        arbol.add ("el");
        arbol.add ("brazo");

        Comparable ss = new String ("fornida");
        System.out.println ("El arbol contiene la palabra " + ss + ":" +
arbol.contains(ss));

        System.out.println (arbol.toString());

        arbol.delete(ss);
        System.out.println ("El arbol contiene la palabra " + ss + ":" +
arbol.contains(ss));
        }
}
```

Al ejecutar el anterior fragmento de código, observamos cómo la primera vez que comprobamos si la cadena “fornida” está en el árbol el resultado es “true”. Sin embargo, la segunda vez, tras eliminarla, el resultado es “false”.

Al imprimir el árbol con el método “toString(): String” podemos observar que también cumple la propiedad que le exigíamos de que el “hijo” izquierdo de un nodo siempre debe ser menor que el “padre”, y el “hijo” derecho siempre debe ser mayor.

Un último hecho reseñable sobre “interfaces” en Java que hemos pasado por alto es el hecho de que una “interface” “B” puede heredar de otra “interface” “A”. Esto quiere decir que tomará todas las declaraciones de métodos propios e la primera, y además añadirá las correspondientes a “B”. En notación Java,

esto se escribiría como “interface A extends B{...}”. Posteriormente, una clase podría implementar (“implements”) la “interface” “B”.

4.3.3 SINTAXIS DE CLASES COMPLETAMENTE ABSTRACTAS EN C++: UN EJEMPLO DESARROLLADO

Pasemos ahora a ver un ejemplo de uso de clases completamente abstractas en C++. Lo primero que conviene recordar es que C++ no dispone de “interfaces”, y lo que vamos a hacer es simular los mismos por medio de “clases completamente abstractas”. Por tanto, la “interface” “Search” que definimos en Java pasará a ser ahora una clase (completamente abstracta) en C++. Por lo demás, el código del ejemplo quedaría como sigue:

```
//Fichero Node.h

#ifndef NODE_H
#define NODE_H 1

class Node{
    public:
            char data [50];
            Node * left;
            Node * right;
    public:
          Node (char []);
};

#endif
```

Sobre la declaración anterior, conviene señalar que los objetos “left” y “right” han sido implementados por medio de punteros a objetos, y no de objetos. En realidad esto no es una decisión voluntaria; intentar hacerlo por medio de objetos habría producido un error de compilación, ya que lo anterior daría lugar a una definición de tipo de dato recursiva. Por medio del uso punteros, un “Node” será un objeto que “apunte” a otros dos “Node”, no que los contenga. Este puede ser un buen ejemplo para ilustrar las diferencias entre el modelo de gestión de memoria de Java y el de C++.

Por lo demás, la ausencia de un “interface” “Comparable” en C++ nos ha obligado a decantarnos por un tipo concreto para representar el “data” de “Node”. Podríamos haber simulado algo parecido a la solución en Java por medio de lo que se conoce en C++ como “Templates”, pero no abordaremos ese tema en este curso. En este caso nos hemos decantado por el tipo “char []”, que cuenta con la función de comparación “strcmp (const char *, const char *): int” (en <cstring>).

Una tercera diferencia con la implementación en Java es que hemos debido declarar los atributos “data”, “left” y “right” como “public”, para que puedan ser visibles directamente desde “BinarySearch” y “BinarySearchTree”. Recuerda que en Java pudimos declararlos como “protected”, ya que en Java el

modificador “protected” hace que sean visibles desde las subclases y también desde el “package” en el que nos encontremos (y no sólo desde las subclases, como en C++).

Veamos ahora cómo queda el fichero “Node.cpp”:

```
//Fichero Node.cpp

#include <cstdlib>
#include <cstring>
#include "Node.h"

using namespace std;

Node::Node (char data []){
        strcpy (this->data, data);
        this->left = NULL;
        this->right = NULL;
}
```

Lo único reseñable sobre el mismo es el uso de “NULL” como “puntero nulo”, frente a “null” en Java, y que hemos evitado el método “toString(): char *”, ya que “data: char [50]” directamente se puede mostrar por pantalla.

Pasamos ahora a la clase “BinaryTree”:

```
//Fichero “BinaryTree.h”:

#ifndef BINARYTREE_H
#define BINARYTREE_H 1

#include "Node.h"

class BinaryTree{
    protected:
            Node * root;
    private:
            char aux [1000];
    public:
        BinaryTree();
        BinaryTree(Node *);
        BinaryTree(char [], BinaryTree *, BinaryTree *);
        BinaryTree * getLeftSubTree();
        BinaryTree * getRightSubTree();
        bool isLeaf();
        void preOrderTraverse(Node *, int, char []);
        char * toString();
};

#endif
```

La principal diferencia reseñable con respecto a la versión en Java de la misma es de nuevo el uso de un puntero a “Node” para albergar la raíz (“root”) de la misma. Una de las ventajas de lo mismo es que la raíz del árbol está representada en memoria igual que los nodos restantes, y por tanto en el procesamiento del árbol no habrá que prestarle especial atención.

El atributo especial “aux: char[1000]” que hemos incluido con respecto a la versión en Java se va a utilizar como “agregador” para la cadena que devuelve el método “toString(): char *”. La solución alternativa sería representarlo como una variable auxiliar en el método “”toString(): char *”, pero al terminar la ejecución de dicho método la variable auxiliar podría ser eliminada de memoria, y por tanto la cadena se perdería.

```
//Fichero BinaryTree.cpp

#include <cstdlib>
#include <cstring>

#include "BinaryTree.h"

using namespace std;

BinaryTree::BinaryTree(){
    this->root = NULL;
}

BinaryTree::BinaryTree(Node * nodo){
    this->root = nodo;
}

BinaryTree::BinaryTree(char data [], BinaryTree * leftTree, BinaryTree *
rightTree){
    this->root = new Node(data);
    if (leftTree != NULL){
        this->root->left = leftTree->root;
    }
    else {
        this->root->left = NULL;
    }
    if (rightTree != NULL){
        this->root->right = rightTree->root;
    }
    else {
        this->root->right = NULL;
    }
}

BinaryTree * BinaryTree::getLeftSubTree(){
    if (this->root != NULL && this->root->left != NULL){
```

```
        return new BinaryTree (this->root->left);
    }
    else {
        return NULL;
    }
}

BinaryTree * BinaryTree::getRightSubTree(){
    if (this->root != NULL && this->root->right != NULL){
        return new BinaryTree (this->root->right);
    }
    else {
        return NULL;
    }
}

bool BinaryTree::isLeaf(){
    return (this->root->left == NULL && this->root->right == NULL);
}

void BinaryTree::preOrderTraverse(Node * node, int depth, char sb[]){
    for (int i = 1; i < depth; i++){
        strcat (sb, " ");
    }
    if (node == NULL){
       strcat (sb, "NULL\n");
    }
    else {
       strcat (sb, node->data);
       strcat (sb, "\n");
       this->preOrderTraverse (node->left, depth + 1, sb);
       this->preOrderTraverse (node->right, depth + 1, sb);
    }
}

char * BinaryTree::toString(){
    strcpy (aux, "\0");
    preOrderTraverse(root, 1, aux);
    return aux;
}
```

La implementación de la clase no guarda apenas diferencias con la dada en Java. Únicamente reseñar el uso del método “strcmp (const char *, const char *): int” en lugar de “compareTo(String): int” para comparar cadenas de caracteres y del método “strcat (const char *, char *): char *” para concatenar la cadena “aux” en el método “preOrderTraverse(Node *, int, char []): void”, en lugar de utilizar “StringBuilder” y el método “append (String): String”, como hicimos en Java.

Pasamos ahora a introducir la “clase completamente abstracta” “Search”, que se corresponde a la “interface” que teníamos en Java. Insistimos una vez más en que, al contener la misma sólo declaraciones de métodos, en C++ sólo da lugar a un fichero de cabeceras, de nombre “Search.h”, y por tanto no habrá fichero “*.cpp”. También conviene destacar que todos los métodos en la misma serán declarados como abstractos, lo cual implica que en C++ deberán llevar el añadido “= 0” en su declaración. Como esperamos comportamiento polimorfo de los mismos, también llevarán el modificador “virtual”. Si bien en Java todos los métodos en una “interface” eran de tipo “public” por defecto, en C++ los elementos de una clase que no llevan modificador de visibilidad son de tipo “private”, así que deberemos añadir también el modificador “public”. Con estas consideraciones, el fichero “Search.h” quedará como sigue:

```
//Fichero Search.h

#ifndef SEARCH_H
#define SEARCH_H 1

#ifndef SEARCH_H
#define SEARCH_H 1

class Search{
  public:
        virtual bool add (char []) = 0;
        virtual bool contains (char []) = 0;
        virtual char * find (char []) = 0;
        virtual char * deletes (char []) = 0;//delete está reservada en C++
        virtual bool remove (char []) = 0;
};

#endif
```

Únicamente añadir a lo dicho en el párrafo anterior que, siendo “delete” una palabra reservada en C++ (para liberar memoria dinámica), hemos tenido que denominar al método correspondiente como “deletes(char []): char *”.

Como podemos observar, el anterior fragmento de código declara una clase (no una “interface”), por lo que el tipo de relación que definiremos ahora a partir de ella será de herencia (no de implementación, como en Java). De este modo, la clase “BinarySearchTree” ahora heredará de las clases “BinaryTree” y “Search”, y nos encontramos con un caso de herencia múltiple. La declaración de la misma será:

```
//Fichero BinarySearchTree.h

#ifndef BINARYSEARCHTREE_H
#define BINARYSEARCHTREE_H 1

#include "BinaryTree.h"
#include "Search.h"
```

```
class BinarySearchTree: public BinaryTree, public Search{
    private: bool addReturn;
    private: char deleteReturn[20];
    private: char returnValue[50];
    public:
        BinarySearchTree();
        char * find (char []);
    private:
        char * find (Node *, char []);
    public:
        bool contains (char []);
        bool add (char []);
    private:
        Node * add (Node *, char []);
    public:
        char * deletes (char []);
    private:
        Node * deletes (Node *, char []);
        char * findLargestChild(Node *);
    public:
        bool remove (char []);
};

#endif
```

En primer lugar deberíamos prestar atención a la cabecera de la misma:

class BinarySearchTree: public BinaryTree, public Search{…}

La forma de declarar que la clase hereda de dos clase distintas es poniendo el modificador “public” delante de ellas y después el nombre de las mismas. Las clases se separan por medio de una coma. Esto quiere decir que la clase “BinarySearchTree” cuenta ahora con todos los métodos propios de la clase “BinaryTree”, con los métodos (abstractos) de la clase “Search”, y con los que pueda definir como propios. Si no queremos que la clase “BinarySearchTree” sea abstracta deberemos definir en la misma todos y cada uno de los métodos de la “clase completamente abstracta” “Search”.

Con respecto a la versión en Java se puede observar que hemos añadido un atributo auxiliar adicional, “returnValue: char [50]”, que nos será de utilidad en el método “char * findLargestChild(Node *)”, para albergar su valor de retorno (por motivos similares a los que apuntamos al añadir “aux: char [50]” en la clase “BinaryTree” para el método “preOrderTraverse (Node *, int, char []): void”).

Sobre la implementación interna de la misma y el uso de métodos auxiliares recursivos (declarados como “private”) nos remitimos a los comentarios que ya introdujimos sobre la misma en la versión de Java (Sección 4.3.2).

Veamos ahora como quedaría la definición de la misma:

```
//Fichero BinarySearchTree.cpp

#include <cstdlib>
#include <cstring>
#include "BinarySearchTree.h"

using namespace std;

BinarySearchTree::BinarySearchTree(): BinaryTree(){

}

char * BinarySearchTree::find (char target[]){
    return this->find (root, target);
}

char * BinarySearchTree::find (Node * localRoot, char target[]){
    if (localRoot == NULL){
       return "NULL";
    }
    int compResult = strcmp (target, localRoot->data);
    if (compResult == 0){
       return localRoot->data;
    }
    else if (compResult < 0){
        return find (localRoot->left, target);
    }
    else return find (localRoot->right, target);
}

bool BinarySearchTree::contains (char target[]){
    char aux [50];
    strcpy (aux, this->find (target));
    if (strcmp (aux, "NULL") == 0){
                return false;
     }
     else {
                return true;
     }
}

bool BinarySearchTree::add (char item[]){
          root = add (root, item);
          return addReturn;
}

Node * BinarySearchTree::add (Node * localRoot, char item []){
          if (localRoot == NULL){
```

```
                addReturn = true;
                return new Node (item);
        }
        else if (strcmp (item, localRoot->data) == 0){
                addReturn = false;
                return localRoot;
        }
        else if (strcmp (item, localRoot->data) < 0){
                localRoot->left = add (localRoot->left, item);
                return localRoot;
        }
        else {
                localRoot->right = add (localRoot->right, item);
                return localRoot;
        }
}

char * BinarySearchTree::deletes (char target[]){
    this->root = deletes (this->root,target);
    return deleteReturn;
}

Node * BinarySearchTree::deletes (Node * localRoot, char item []){
    if (localRoot == NULL){
                strcpy (deleteReturn, "NULL");
                return localRoot;
     }
     int compResult = strcmp (item, localRoot->data);
     if (compResult < 0){
                localRoot->left = deletes (localRoot->left, item);
                return localRoot;
        }
        else if (compResult > 0){
                localRoot->right = deletes(localRoot->right, item);
                return localRoot;
        }
        else {
                strcpy (deleteReturn, localRoot->data);
                if (localRoot->left == NULL){
                        return localRoot->right;
                }
                else if (localRoot->right == NULL){
                        return localRoot->left;
                }
                else {
                        if (localRoot->left->right == NULL){
                                strcpy (localRoot->data,localRoot->left->data);
                                localRoot->left = localRoot->left->left;
                                return localRoot;
                        }
```

```
                        else {
                    strcpy(localRoot->data, findLargestChild(localRoot->left));
                            return localRoot;
                        }
                    }
    }
}

char * BinarySearchTree::findLargestChild(Node * parent){
    if (parent->right->right == NULL){
        strcpy (returnValue, parent->right->data);
                    parent->right = parent->right->left;
                    return returnValue;
            }
            else {
                    return findLargestChild(parent->right);
            }
}

bool BinarySearchTree::remove (char target []){
    char aux [50];
    strcpy (aux, this->deletes (target));
    if (strcmp (aux, NULL)){
      return false;
    }
    else{
        return true;
    }
}
```

La programación de la misma es similar a la realizada en Java, así que los comentarios que hicimos para aquélla (ver Sección 4.3.2) se pueden recuperar también ahora.

Con respecto a los métodos abstractos heredaros de “Search”, únicamente señalar que todos ellos deben ser definidos en “BinarySearchTree” para que la clase deje de ser abstracta.

Veamos ahora un sencillo ejemplo de programa principal programado sobre las anteriores clases (y similar al que presentamos en Java):

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

#include "Node.h"
#include "BinaryTree.h"
#include "Search.h"
#include "BinarySearchTree.h"

using namespace std;
```

```
int main(){

	BinarySearchTree * arbol = new BinarySearchTree();

	arbol->add ("Es");
	arbol->add ("algo");
	arbol->add ("formidable");
	arbol->add ("que");
	arbol->add ("vio");
	arbol->add ("la");
	arbol->add ("vieja");
	arbol->add ("raza");
	arbol->add ("robusto");
	arbol->add ("tronco");
	arbol->add ("de");
	arbol->add ("arbol");
	arbol->add ("al");
	arbol->add ("hombro");
	arbol->add ("de");
	arbol->add ("un");
	arbol->add ("campeon");
	arbol->add ("salvaje");
	arbol->add ("y");
	arbol->add ("aguerrido");
	arbol->add ("cuya");
	arbol->add ("fornida");
	arbol->add ("maza");
	arbol->add ("blandiera");
	arbol->add ("el");
	arbol->add ("brazo");


	char ss [50];
	strcpy (ss, "fornida");
	cout << "El arbol contiene " << ss << ":" << arbol->contains(ss)<<endl;
	cout << arbol->toString()<<endl;

	arbol->deletes(ss);

	cout << "-----------------------------------------------------" << endl;
	cout << "El arbol contiene " << ss << ":" << arbol->contains(ss) <<endl;

	cout << arbol->toString()<<endl;

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

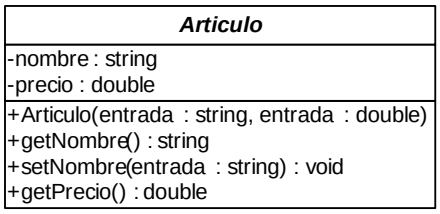
Es importante observar que esperamos que los métodos del objeto creado "arbol" se comporten de modo polimorfo (en este caso no es estrictamente necesario, ya que lo hemos declarado de la clase "BinarySearchTree", pero sí lo sería en caso de haberlo declarado de la clase "Search"), y por tanto es necesario declararlo como un "puntero a objeto", es decir, en memoria dinámica (como enunciábamos en el Tema 3).

Por lo demás, si observas el resultado de ejecutar el mismo, podrás comprobar que en el objeto "arbol" los "hijos" izquierdos son siempre menores que sus padres, con respecto al orden lexicográfico, mientras que los "hijos" derechos son siempre mayores.

4.4 REPRESENTACIÓN EN UML DE MÉTODOS ABSTRACTOS, CLASES ABSTRACTAS E INTERFACES

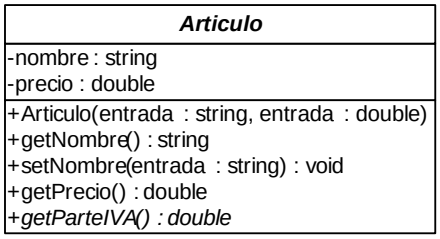
La notación UML de métodos abstractos, clases abstractas e interfaces la hemos introducido a lo largo del Tema, así que aquí únicamente recuperamos algunos de los ejemplos introducidos a modo de breve reseña sobre la misma.

Con respecto a las clases abstractas, ya dijimos que la forma de representarlas en UML era escribiendo su nombre en letra cursiva. Supongamos que tenemos la clase "*Articulo*" declarada como abstracta, aunque tenga todos sus métodos definidos (aprovechamos para incidir en la idea de que esto es posible en Java, pero no así en C++). La representación UML de la misma sería:



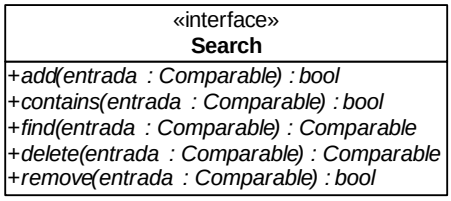
Aprovechamos también para incidir en que, al ser abstracta, no podremos construir objetos de la misma (aunque sí podamos declararlos).

Pasamos ahora a recordar la notación para métodos abstractos. Al igual que con las clases abstractas, debemos tener en cuenta que los mismos se deben representar en UML en cursiva. Supongamos que sobre la clase anterior queremos introducir ahora un método abstracto de nombre "*getParteIVA(): double*". La representación de la misma sería ahora:



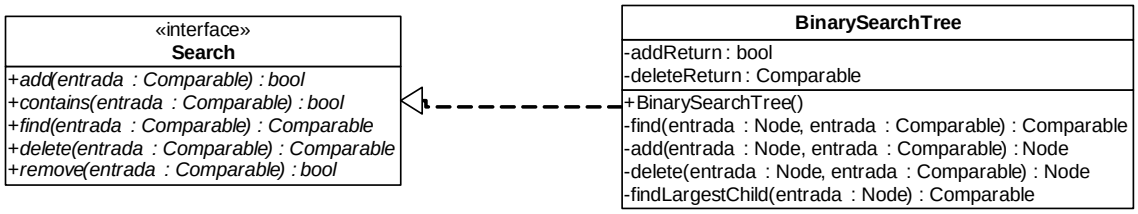
Apuntamos de nuevo que el hecho de que la clase "*Articulo*" sea abstracta es consecuencia de que contiene un método abstracto, "*getParteIVA(): double*".

Por último, pasamos a mostrar la notación propia de “interfaces” en UML (noción que no existe en C++, y que representamos por medio de “clases completamente abstractas”):



Observar únicamente la presencia del modificador “<<interface>>” en la cabecera de la caja. Asimismo, los métodos de la “interface” son abstractos, lo cual quiere decir que deberemos escribirlos en cursiva

Un hecho también reseñable y que requiere de notación específica es que la relación entre una clase y una “interface” es una relación de implementación. Dicha relación se denota en UML por medio de una línea punteada. Por ejemplo, el hecho de que la clase “BinarySearchTree” implemente la “interface” “Search” se denota en UML como:



4.5 IMPLEMENTACIÓN EN C++ DE MÉTODOS ABSTRACTOS Y CLASES ABSTRACTAS

Repasaremos simplemente las notaciones específicas de C++ que hemos ido introduciendo a lo largo de este Tema:

a) Las clases abstractas en C++ no tienen una notación específica. Una clase, tanto si es abstracta como si no, se declara por medio de la palabra reservada “class” antepuesta a su nombre (en el archivo de cabeceras). Por lo tanto, el hecho de que una clase sea abstracta o no depende exclusivamente en que ésta contenga algún método abstracto o no (a diferencia de Java).

b) Un método abstracto se declara en C++ de la siguiente forma:

```
class A {
        ...
        public: virtual TipoRetorno NombreMetodo (parametros) = 0;
        ...
}
```

El modificador de visibilidad puede ser “public” o “protected”, “private” no tendría sentido para un método abstracto.

El modificador “virtual” hace que el método se enlace de forma dinámica, es decir, se comporte de modo polimorfo. De no ser así, no podríamos redefinirlo en las clases derivadas o subclases.

La declaración “=0” le dice al compilador que el método no está definido (es abstracto), y por tanto en el fichero “*.cpp” no deberemos incluirlo.

c) Las clases completamente abstractas se declaran como la clase abstracta en el ejemplo anterior, haciendo que todos los métodos sean abstractos.

Como clases que son, en C++ las “clases completamente abstractas” se heredan, no se implementan.

4.6 IMPLEMENTACIÓN EN JAVA DE MÉTODOS ABSTRACTOS E INTERFACES

Repasamos la notación en Java de los conceptos introducidos en el Tema.

a) Clases abstractas: la forma de declarar en Java una clase como abstracta es por medio del modificador “abstract”. Esto lo podemos hacer incluso si la clase no contiene ningún método abstracto.

```
abstract class A {
        …
}
```

Recordamos también ahora que las clases abstractas no pueden ser instanciadas. No se pueden construir objetos de las mismas, aunque sí declararlos.

b) Métodos abstractos: el mismo modificador “abstract” que hemos introducido para clases nos sirve para declarar un método como abstracto en una clase:

```
abstract class A{
        …
        public abstract tipoRetorno nombreMetodo (parametros);
        …
}
```

En Java, una clase que contenga un método abstracto debe ser declarada como “abstract”. Las clases que hereden de ésta deberán definir el método abstracto para dejar de ser abstractas; en caso de no definirlo, también serán abstractas.

c) Por último, la notación específica de “interfaces”:

```
interface A{
```

```
    ...
    tipoRetorno nombreMetodo (parametros);
    ...
}
```

Una “interface” sólo puede contener declaraciones de métodos abstractos (y definición de constantes, aunque esté desaconsejado hacerlas). Por tanto, todos los métodos que aparezcan en la misma por defecto serán “public” y “abstract”. De este modo, la anterior declaración de “interface” sería equivalente a:

```
interface A{
    ...
    public abstract tipoRetorno nombreMetodo (parametros);
    ...
}
```

Es conveniente recordar de nuevo que la relación entre una clase y una “interface” es de implementación, no de herencia, así que si una clase “B” implementa una “interface” “A”, lo mismo se denota en Java con la palabra reservada “implements”:

```
class B implements A {…}
```