TEMA 1. NOCIONES DE CLASE Y OBJETO EN PROGRAMACIÓN ORIENTADA A OBJETOS

Introducción al tema:

El presente tema introduce las nociones de objeto y de clase como se definen en la Programación Orientada a Objetos (POO). Teniendo en cuenta que los alumnos ya están familiarizados con la programación estructurada y los conceptos de variable, función, o tipo abstracto de datos, trataremos de poner de relieve las diferencias entre esos conceptos y los aquí presentados.

El tema se desarrollará de la siguiente forma. Primero pondremos algunos ejemplos de representación de la información por medio de la estructura conocida como objeto (Sección 1.1), para luego pasar a detallar sus partes más representativas: sus atributos (Sección 1.2) y sus métodos (o interfaz) (Sección 1.3). Presentaremos posteriormente la noción de clase como resultado del proceso de abstracción sobre una colección de objetos con idénticos atributos y métodos (Sección 1.4).

Posteriormente presentaremos los métodos más relevantes de toda clase: en primer lugar, los constructores, que permiten definir el valor inicial que toma un objeto al ser definido (Sección 1.5), y en segundo lugar los métodos de acceso y modificación de cada uno de los atributos de la clase (Sección 1.6). Finalmente también presentaremos un tipo de atributos y métodos particulares que son conocidos como estáticos y que facilitan la definición, por ejemplo, de constantes de clase y de métodos auxiliares (Sección 1.7).

De ahí pasaremos a hablar de los modificadores de acceso dentro de una clase, que nos permiten separar la misma en una parte pública y otra privada, y de las consecuencias de dicha separación (Sección 1.8). De la existencia de una parte privada y de la separación entre métodos y atributos de clase pasaremos a presentar la noción de ocultación de la información por medio de distintos ejemplos en los cuales los alumnos han de ser capaces de modificar los atributos de una clase sin que el comportamiento (o la interfaz o parte pública de la misma) sufran alteración (Sección 1.9).

La última parte del tema será de tipo más práctico y servirá para presentar la representación de las nociones anteriormente introducidas en el lenguaje de especificación UML (Sección 1.10) y en los lenguajes de programación C++ (Sección 1.11) y Java (Sección 1.12). Esta última parte se presentará de forma conjunta en el aula de teoría y en las sesiones uno y dos de prácticas.

1.1 REPRESENTACIÓN DE LA INFORMACIÓN POR MEDIO DE OBJETOS

La programación estructurada, la conocida por los alumnos hasta este momento, está basada en variables que contienen un valor concreto en cada momento (una vez han sido inicializadas). Sobre esas variables actúan una serie de funciones.

La noción de objeto generaliza la noción de variable, en el sentido de que un objeto puede poseer simultáneamente un valor (o estado) y una serie de funciones (llamadas métodos, y que se identifican con el comportamiento del objeto) actuando sobre el mismo.

Definición: un objeto es una entidad de un programa capaz de albergar al mismo tiempo una serie de valores (a los que llamaremos el estado de ese objeto) y de métodos (a los que definiremos como el comportamiento de ese objeto).

Ejemplos:

Consideramos la siguiente declaración de un tipo de dato y de una variable del mismo en un lenguaje estructurado:

```
Registro Película {

        cadena título;
        entero duración;
        cadena director;

}

Película El_señor_de_los_anillos;
El_señor_de_los_anillos.titulo = “El señor de los anillos”;
El_señor_de_los_anillos.duración = 155; (minutos)
El_señor_de_los_anillos.director = “Peter Jackson”;
```

Y la siguiente lista de funciones:

```
cadena getTitulo (Película x) {
        return x.titulo;
}

entero getDuracion (Película x) {
        return x.duración;
}

cadena setDirector (Película x, cadena nombre) {
        x.director = nombre;
}

entero z = getDuracion (El_señor_de_los_anillos);
setDirector (El_señor_de_los_anillos, “P. Jackson”);
```

Como se puede observar, las funciones reciben como parámetro la variable sobre la que actúan.

Solución en POO (y algunos comentarios):

La POO propone unificar el estado y el comportamiento de las variables. De ese modo, los métodos que actúan sobre un objeto pasan a formar parte del mismo objeto. Veamos cómo queda el ejemplo anterior codificado siguiendo POO:

El_señor_de_los_anillos.setTitulo (“El señor de los Anillos II”);
El_señor_de_los_anillos.getTitulo ();
El_señor_de_los_anillos.setDirector (“Peter Jackson”);

El mayor cambio que podemos apreciar en el código anterior es que los métodos son invocados directamente a través de un objeto activo. La variable deja de ser un parámetro, para ser un objeto sobre el cual los métodos son invocados. Haremos mayor hincapié en la definición de métodos en la Sección 1.3 de este capítulo.

Un segundo cambio, más sutil, es que en las invocaciones anteriores no hemos hecho referencia a los atributos del objeto (los atributos se verán más en detalle en la Sección 1.2), sino que hemos accedido a ellos y los hemos manipulado por medio de métodos. Esto es lo que habitualmente se conoce como ocultación de la información, y será detallado en la Sección 1.8.

Como se puede inferir fácilmente, el anterior ejemplo es una concreción de un tipo de dato, del mismo modo que 3 es una concreción del tipo de dato “entero” o “Peter Jackson” es una concreción del tipo de dato “cadena”. A este tipo de dato cuyas concreciones son los objetos antes presentados se le llama clase, y es otra de las nociones centrales de la POO. Veremos una definición más concreta de clase y diversos ejemplos en la Sección 1.4.

Información más relevante de la Sección 1.1:

Un objeto es una entidad dentro de un lenguaje de programación que contiene al mismo tiempo un estado (dado por los valores que toman sus atributos) y un comportamiento (dado por una serie de métodos que permiten comunicarse con dicho objeto)

En particular, una variable en un lenguaje no orientado a objetos se puede entender como un objeto que sólo posee estado. De modo similar, las funciones que actúan sobre esa variable se podrían identificar con los métodos de la misma.

1.2 ATRIBUTOS O ESTADO

De las dos partes que componen un objeto, sus atributos son las más fáciles de identificar, especialmente para aquellos familiarizados con un lenguaje de programación estructurada.

Los atributos nos permiten conocer el valor determinado que tiene un dato en un momento determinado. La elección de los mismos depende del tipo de problema que queramos resolver. En el ejemplo anterior sobre películas, queda

a decisión del usuario considerar como un atributo la nacionalidad de la película. Es inevitable la comparación entre el estado o lista de atributos de un objeto y un dato de tipo registro en programación estructurada.
El tipo de problema que queramos resolver, la cantidad de memoria de la que dispongamos o los tipos básicos del lenguaje en el que estamos programando son algunas de las distintas variables que debemos considerar a la hora de realizar el proceso de abstracción que nos lleva a identificar los atributos que debemos representar de un objeto.

Por ejemplo, si queremos representar los números enteros, uno de los atributos que nos servirán para definir un número entero es el valor que tiene en un momento concreto (i = 14, x = 23). Dependiendo del tipo de problema que queramos resolver, quizá nos podría ser de utilidad tener un atributo explícito, llamado “signo”, de tipo booleano, que nos permitiese distinguir los enteros positivos de los negativos.

Si queremos representar un vector de cinco componentes, nos interesará conocer, al menos, el valor que toman cada una de esas componentes,

v [0] = 2, v [1] = 3, v [2] = 7, v [3] = -6, v [4] = 98

Elegir los atributos de una variable, o de un objeto, puede no ser siempre trivial, y depende del problema que queramos resolver. Generalmente la respuesta “ideal” se encuentra cuando comenzamos a implementar las funciones y métodos que han de trabajar sobre ese tipo de dato, y se ve de forma más clara la información más relevante sobre el mismo en el problema que tratamos de resolver.

Por ejemplo, la longitud del vector anterior se podría incluir como un atributo del mismo:

v.longitud = 5;

Pero, ¿por qué no haberlo considerado como un método del mismo? El considerarlo como un atributo tiene una ventaja, qe es que podemos disponer de esa información de forma inmediata, aumentando el rendimiento de nuestra aplicación. La segunda opción, sin embargo, salvaría memoria porque el dato no es explícitamente guardado, sino que se calcula cuando se considere necesario.

Cuando tratamos con ejemplos más complejos, la toma de decisiones sobre los atributos que debemos guardar de un tipo de dato son más complicadas. Por ejemplo, si queremos desarrollar una aplicación de contratación de billetes a través de Internet para una aerolínea, ¿cuáles con los atributos que debemos codificar sobre cada usuario?

Algunos de ellos parecen obvios, como por ejemplo el nombre, el número de DNI (o de tarjeta de identificación), la edad, ..., ¿sería necesario conocer su dirección? ¿Su número de teléfono? ¿Su dirección de correo electrónico? Si esa información es irrelevante y la guardamos para los miles de usuarios de la

web, estaríamos desperdiciando gran cantidad de recursos de nuestra empresa, lo cual lo convierte en una decisión con importantes consecuencias.

1.3 MÉTODOS O COMPORTAMIENTO

Una de las mayores diferencias entre los lenguajes de programación estructurada y los lenguajes de POO es que en los segundos los objetos disponen de métodos asociados a los mismos (mientras que en los lenguajes estructurados las funciones actúan sobre las variables, pero no forman parte de las mismas). Esto es lo que se conoce en la jerga de POO como encapsulación de la información, ya que todos los atributos y también los métodos asociados a un tipo de dato se encuentran “encapsulados” en una misma entidad, una clase, que introduciremos en la Sección 1.4.

Esta diferencia tiene una serie de consecuencias importantes:

1. Un método de un objeto sólo existe en tanto en cuanto el objeto exista (¿Cómo era la situación en programación estructurada? ¿Tiene sentido tener una función que trabaja sobre un tipo de variables del cual no se ha definido ninguna instancia?)

2. Los métodos (del objeto) nos permiten enmascarar los atributos del mismo. Esto lo veremos en más detalle en la Sección 1.8, cuando hablemos de ocultación de la información. (De nuevo, podemos formular una pregunta similar: ¿Qué pasaría en un programa en el que usas un registro si decides modificar uno de sus atributos?)

3. Cuando queramos ampliar o modificar la funcionalidad de un objeto, sólo tendremos que prestar atención a la clase (Sección 1.4) donde el objeto está definido. Se consigue así aumentar la *Modularidad* del código, o, dicho de otro modo, la encapsulación de la información (cada objeto y sus métodos están unidos en un mismo fragmento de código).

Los métodos también se pueden considerar como la forma de comunicarse con un objeto. Cuando queramos conocer cualquiera de las propiedades del mismo (o cualquier dato sobre su comportamiento), todo lo que tendremos que hacer es invocar a uno de sus métodos. De ahí la importancia de elegir bien el conjunto de métodos de que va a disponer un objeto.

Por ejemplo:

El_señor_de_los_anillos.getTitulo ();

sería la forma natural de conocer el título de una película.

En programación estructurada, la anterior consulta se habría realizado de la forma:

getTitulo (El_señor_de_los_anillos);

Pero uno debería formularse las siguientes preguntas:

1. ¿Qué sentido tiene la existencia de la función “getTitulo(cadena)” si no existe antes un objeto de tipo “Película”? En realidad sería un desperdicio de memoria

2. ¿Por qué la función “getTitulo(cadena)” se ha de definir de forma externa a la clase Pelicula, cuando actúa sobre objetos de dicha clase?

Otros ejemplos se pueden extraer del mundo de las matemáticas:

real mi_numero = 12.25;

En POO:

mi_numero.cuadrado ();

En programación estructurada:

cuadrado (mi_numero);

¿La raíz de un número es una propiedad del mismo, o es más natural verla como una función que actúa de números en números?

Si todo número tiene cuadrado (o toda película tiene un título) ¿por qué no asociar dichas funcionalidades al propio objeto? El asociar un tipo de dato con todas las funcionalidades que esperamos obtener del mismo nos ayuda a que nuestro código esté más estructurado (por una parte los tipos de datos, por otra los clientes del mismo).

El rango de ejemplos que se pueden pensar sobre este tipo de planteamiento es muy amplio.

Por ejemplo, el caso que veíamos previamente con los vectores:

v [0] = 2, v [1] = 3, v [2] = 7, v [3] = -6, v [4] = 98

Ahora podríamos disponer de un método propio del tipo de dato vector que nos permitiera conocer su longitud:

v.longitud ();

Del mismo modo, un método que ordenara las componentes de un vector dentro de la clase haría que el mismo se invocara como:

v.ordenar ();

El planteamiento anterior, en el que cada objeto dispone de una colección propia de métodos, podría dar lugar a una situación un tanto alarmante desde el punto de vista de la memoria que ocupa nuestra aplicación ¿Por cada objeto que tengamos habrá una copia de todos y cada uno de los métodos? (Por

ejemplo, si tuviéramos 100 objetos de tipo vector, ¿habría 100 copias del método “longitud()” en nuestro programa?. Si tuviéramos 50 objetos de tipo real, ¿dispondríamos de 50 copias del método “cuadrado()”?). Desde el punto de vista de la memoria necesitada para un programa, lo anterior sería inviable. En realidad la gestión de memoria está optimizada en el siguiente sentido: como todos los métodos de una colección de objetos tienen similar comportamiento, una única copia de cada método es guardada en la memoria del programa. Cada vez que un objeto es definido, se crea una referencia de dicho objeto a la colección de métodos que ese objeto debe implementar.

La anterior idea de abstracción de una familia de métodos comunes a una serie de objetos en una clase, nos sirve para enlazar con la noción de clase que presentamos en la siguiente Sección.

1.4 ABSTRACCIÓN DE OBJETOS EN CLASES

Hasta ahora hemos hablado de la noción de objeto como una generalización de la de variable, en el sentido de que un objeto puede contener tanto una serie de atributos (como una variable), como una serie de métodos que nos permiten interactuar con dicho objeto (lo que hemos dado en denominar antes como encapsulación de la información).

En programación estructurada, las variables son concreciones de un tipo de datos determinado. Por ejemplo, podemos definir:

entero x = 5;

La sentencia anterior define una variable, cuyo nombre será “x” y cuyo valor concreto al inicializarla es “5”.

De igual modo, en POO, los objetos son concreciones (o instanciaciones) de unos tipos de datos que se conocen como clases. Se puede decir que todo objeto es el resultado de instanciar una clase.

Definición (extraída de “The Essence of Object-Oriented Programming with Java and UML”, Bruce E. Wampler, Addison Wesley, 2002): Una clase es una descripción de un conjunto de objetos. El conjunto de objetos comparten atributos y comportamiento común. Una clase es similar en concepto a los tipos abstractos de datos que se pueden encontrar en los lenguajes no orientados a objetos, pero es más extensa en el sentido de que incluye tanto estado como comportamiento. Una definición de clase describe todos los atributos de los objetos que pertenecen a esa clase, así como los métodos de la clase implementan el comportamiento de los objetos miembro.

Como hemos señalado anteriormente, los objetos pueden tener atributos y métodos, así que una clase tiene que ser capaz de recopilar la información correspondiente a esos atributos y métodos.

La declaración de una clase vendrá dada por los tipos de datos correspondientes a los atributos de la misma y a sus métodos. La definición de

la misma estará dada por la especificación de sus métodos. En algunos lenguajes de programación (C++) la declaración y definición de la clase se pueden separar. En otros (como Java) declaración y definición suelen ser conjuntas.

Veamos un ejemplo basado en el caso anterior sobre películas. Presentamos la declaración de una clase:

```
clase Película {

        // Atributos
        cadena titulo;
        cadena director;
        entero duración;

        //métodos
        cadena getTitulo (); // método que nos devuelve el título de la película
        void setTitulo (cadena); // método que cambia el título de una película
        cadena getDirector (); // método que devuelve el director de una película
        void setDirector (cadena); // cambia el título de una película
        entero getDuración ();//método que devuelve la duración de una película
        void setDuración (entero);// cambia la duración de una película;

}
```

Otro ejemplo de declaración. Si queremos definir un botón en una ventana de interfaz de una aplicación:

```
clase Botón {

        //Atributos
        cadena nombre;
        entero ancho;
        entero alto;
        entero posición_x;
        entero posición_y;
        cadena color;

        //Métodos
        getNombre (); //
        setNombre (cadena); //
        ...
}
```

Un tercer ejemplo de declaración de una clase que recuperaremos en la Sección 1.8:

```
Clase Circunferencia {

        //Atributos
```

```
        radio real;

        //Métodos
        real getRadio ();
        void setRadio (real);
        real getDiametro ();
        void setDiametro (real);
        real getArea ();
        void setArea (real);
        real getLongitud ();
        void setLongitud (real);
}
```

¿Por qué no hemos elegido la siguiente representación?

```
Clase Circunferencia {

        //Atributos
        longitud real;

        //Métodos
        real getRadio ();
        void setRadio (real);
        real getDiametro ();
        void setDiametro (real);
        real getArea ();
        void setArea (real);
        real getLongitud ();
        void setLongitud (real);
    }
```

¿O por qué no guardar cualquier otro de los valores significativos de una circunferencia?

1.5 NECESIDAD Y RELEVANCIA DE LOS CONSTRUCTORES DE CLASE: CONSTRUCTOR POR DEFECTO, CONSTRUCTORES PROPIOS

Hasta ahora hemos visto cómo los objetos pueden representar información en los lenguajes de POO, cuáles con las partes más importantes de un objeto (atributos y métodos), y cómo la información relativa a una familia de objetos con características comunes se puede abstraer dando lugar a una clase.

En esta Sección y la siguiente veremos algunas de las funcionalidades básicas que debe ofrecer una clase.

La primera de ellas está relacionada con la inicialización (o construcción) de objetos. Para **declarar un objeto** perteneciente a una clase sólo hemos de dar un identificador (o nombre) para dicho objeto, y decir a qué clase pertenece ese objeto:

Película Un_día_en_las_carreras;

Sin embargo, declarar un objeto no es suficiente para poder trabajar con el mismo de una forma natural. Surgen dos preguntas a partir de la anterior declaración.

1. ¿Cuál sería el comportamiento ahora esperado para la siguiente orden?

Un_día_en_las_carreras.getDirector ();

¿O para la siguiente?

Un_día_en_las_carreras.getDuracion();

2. Si tratamos de ser un poco más específicos, ¿en qué momento de la declaración del objeto se ha reservado memoria para el mismo?

Para poder responder a las dos preguntas anteriores, parece obvio que antes de poder utilizar un objeto, debemos ser capaces de reservar un espacio en memoria para el mismo y de "construirlo", o iniciarlo.

¿Cuál es la forma de hacerlo en la POO? Por medio de un *método* característico denominado "constructor". La función primordial de un constructor es la de dar un valor inicial a todos y cada uno de los atributos de la clase (esto nos asegurará el buen comportamiento de los objetos de esa clase cuando invoquemos a cada uno de sus métodos).

Manual de buenas prácticas: Los constructores de una clase deben asignar un valor inicial a todos y cada uno de los atributos de dicha clase.

Veamos ahora el comando que nos permite declarar un objeto, reservar espacio en memoria para el mismo, y construirlo:

Película Un_día_en_las_carreras =
    new Película ("Un día en las carreras", "Sam Wood", 111);

¿Cuál sería el comportamiento ahora esperado para la siguiente orden?

Un_día_en_las_carreras.getDirector ();

Del fragmento de código previo se pueden sacar una serie de conclusiones:

1. El nombre de los constructores de una clase debe coincidir con el nombre de dicha clase. Por eso la sintaxis "Película ("Un dia en las carreras", "Sam Wood", 111)" lo que hace realmente es invocar a un método de nombre "Película" (es decir, el mismo nombre que la clase) que recibe tres parámetros, dos de ellos de tipo cadena y el tercero de tipo entero (ese método debe estar en la clase "Película")

2. El anterior comando no sólo construye el objeto Un_día_en_las_carreras, sino que reserva un espacio en memoria para el mismo por medio del comando “new”. El comando “new” tanto en Java como en C++ sirve para reservar memoria de forma dinámica. Más adelante veremos como C++ admite la construcción de (punteros a) objetos de forma dinámica y también la construcción de objetos por medio de memoria no dinámica.

Todavía hay otros dos hechos relevantes sobre los constructores que conviene tener en cuenta:

1. Una clase puede tener varios constructores, siempre y cuando la lista de parámetros de cada uno de ellos sea diferente (es decir, esté formada por distintos tipos o éstos estén en distinto orden):

Por ejemplo, es posible tener varios constructores para la clase Película:

Película ();
Película (cadena);
Película (cadena, cadena, entero);
Película (entero, cadena, cadena)

Pero no sería posible tener dos constructores:

Película (cadena);
Película (cadena);

En este segundo caso, el compilador no sabría a cuál llamar en cada situación.

Se llama *sobrecarga de método* a la capacidad que dan algunos lenguajes de programación para utilizar un mismo identificador (o nombre) de método diversas veces, variando únicamente su lista de parámetros (como ejemplo, puedes observar que el constructor Película en el ejemplo anterior se encuentra sobrecargado).

2. Si nosotros definimos una clase sin ningún constructor, el lenguaje de programación correspondiente (Java o C++) definirá uno por nosotros. El *constructor por defecto* que facilitan los lenguajes de programación, generalmente, es un constructor sin ningún parámetro, y cuyo comportamiento es difícilmente predecible (puedes experimentarlo en los lenguajes de programación que utilizamos en el curso). Esto tiene una desventaja clara, y es que no sabemos (a priori) cuáles serán los valores que asignará a los distintos atributos.

Como consecuencia de los dos puntos anteriores:

Manual de buenas prácticas: Es recomendable definir siempre al menos un constructor en nuestras clases. Esto nos permite controlar cómo se construyen (con qué valores iniciales) los objetos de dicha clase. También puede ser recomendable definir un constructor sin parámetros, ya que en ciertas ocasiones puede simplificar nuestro código (especialmente en C++).

(Veremos diversos ejemplos de construcción de objetos en C++ y Java en las Secciones 1.11 y 1.12, y también en el Tema 2 cuando abordemos ejemplos más complejos)

Como se puede imaginar, una clase sin constructor carece de sentido (por el momento, en el Tema 4 veremos casos de lo contrario) ya que no se puede construir ningún objeto de esa clase.

1.6 MÉTODOS DE ACCESO Y MODIFICACIÓN DEL ESTADO DE UN OBJETO

En la Sección anterior hemos visto un tipo de métodos de gran importancia en la definición de toda clase, como son los constructores.

Una vez que sabemos que un objeto está bien construido, parece obvio que debemos tener medios para acceder a los distintos valores de sus atributos (dicho de otra forma, ¿de qué nos sirve tener una serie de datos almacenados en un objeto a los que no somos capaces de acceder?). Por ejemplo, una vez hemos creado un objeto:

```
Película Un_día_en_las_carreras =
        new Película (“Un día en las carreras”, “Sam Wood”, 111);
```

Es muy probable que los programas clientes de nuestra clase “Pelicula” necesiten acceder a los atributos “titulo”, “director”, “duracion” a lo largo de su ejecución. Aprovechamos este punto para volver a comentar una de las grandes virtudes de la POO, que es la *modularidad* o *encapsulación de la información*. En los grandes desarrollos de software basados en POO, es bastante habitual que una persona se encargue de programar una clase que va a ser utilizada por otros programas (que llamaremos *clientes de la clase*, que ni siquiera serán desarrollados por la misma persona). Esto es posible siempre y cuando la clase tenga una serie de métodos que permitan comunicarse con la misma y cuyo comportamiento debe ser conocido tanto por los programadores de la clase como por los programadores de los clientes de la misma. Después de este breve comentario, ¿qué utilidad tendría un objeto si no podemos conocer el valor de su atributos?

Por ese motivo, casi siempre que definamos una clase, definiremos un método de acceso a cada uno de sus atributos (en inglés, estos métodos son conocidos como “getters”).

Estos métodos no tienen ningún parámetro, y simplemente devuelven el atributo que les señalemos:

```
getTitulo () {return this.titulo;};
getDirector () {return this.director;};
getDuracion () {return this.duracion;};
```

La definición de los métodos anteriores es trivial, simplemente retornan el atributo correspondiente.

No sólo definimos métodos de acceso a los atributos de la clase, sino también en algunos casos es posible que una clase deba ofrecer métodos de acceso a otros tipos de información relacionada con sus atributos. Por ejemplo, en la clase Circunferencia que definimos previamente:

```
Clase Circunferencia {

        //Atributos
        real radio;

        //Constructores de clase

        Circunferencia () {this.radio = 0;};
        Circunferencia (real radio) {this.radio = radio;};

        //Métodos
        real getRadio (){return this.radio;};

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

        real getArea (){return 3.14 * (this->radio) * (this.radio);};

        real getLongitud (){return 2 * 3.14 * (this.radio);};

}
```

No todos los atributos de una clase deben tener un método de acceso, ni los únicos métodos de acceso de una clase corresponden con atributos de la misma. Siempre debe existir una fase de diseño de la clase que permita decidir qué métodos de acceso debe ofrecer la misma.

Las definiciones de los métodos previos nos sirven para introducir un nuevo elemento de la POO, la palabra reservada “this”.

“this” es un puntero (o una referencia) sólo accesible desde los métodos (no estáticos) de una clase. Este puntero apunta al objeto para el cual el método está siendo llamado. La dirección de memoria de dicho objeto es pasada a los métodos como un parámetro implícito.

También habrá situaciones en las que queramos modificar alguno de los atributos de nuestro objeto, o todos ellos, porque éstos son erróneos o porque en el período de vida del objeto, el estado del mismo sufra modificaciones.

Para ello se suele definir, al crear una nueva clase, una serie de métodos que nos permitan modificar el valor de cada uno de los atributos de dicho objeto. Si como decíamos antes, el estado de un método viene dado por el valor de sus atributos, estos métodos nos permiten modificar el estado de dicho objeto.

En inglés, estos métodos se conocen como setters. Su definición contiene un único parámetro, que es el nuevo valor del atributo que vamos a modificar:

setTitulo (cadena nuevo_titulo) { this.titulo = nuevo_titulo;};

setDirector (cadena nuevo_director) {this.director = nuevo_director;};

setDuracion (entero nueva_duracion) { this.duración = nueva_duracion;};

De nuevo se puede observar en la definición de los métodos anteriores el uso de “this”, que sirve para aclarar que el nuevo título va a parar al objeto que está activo en ese momento, e igual con el resto de parámetros.

Dependiendo de la funcionalidad que deba tener nuestra clase es posible que también tengamos que definir métodos de modificación para algunas propiedades del objeto que no coincidan directamente con atributos del mismo:

```
Clase Circunferencia {

        //Atributos
        real radio;

        //Métodos
        void setRadio (real nuevo_radio){this.radio = nuevo_radio;};

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

        void setArea (real nuevo_area){this.radio = (sqrt (nuevo_area / PI));};

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

}
```

Resumen. Después de lo explicado en las dos secciones anteriores, y antes de seguir incluyendo nuevos elementos en nuestra clase, veamos el aspecto aproximado que tiene con lo aprendido hasta ahora.

```
Clase Película {

        //atributos

        cadena titulo;
        cadena director;
        entero duración;

        //Constructores
        Película (){
                this.titulo = “”;
                this.director = “”;
```

```
            this.duracion = 0;
        }

Película (cadena nuevo_titulo, cadena nuevo_director, entero nueva_duracion){
            this.titulo = nuevo_titulo;
            this.director =nuevo_director;
            this.duracion = nueva_duracion;
        }

        //Accesores o getters

        getTitulo () {
            return this.titulo;
        };

        getDirector () {
            return this.director;
        };

        getDuracion () {
            return this.duracion;
        };

        //Modificadores o setters

        setTitulo (cadena nuevo_titulo) {
            this.titulo = nuevo_titulo;
        };

        setDirector (cadena nuevo_director) {
            this.director = nuevo_director;
        };

        setDuracion (entero nueva_duracion) {
            this.duración = nueva_duracion;
        };

        // Métodos adicionales ¿?

}
```

Dos hechos relevantes sobre la anterior definición de la clase Pelicula:

1. Nuestra clase dispone de dos métodos constructores de nombre “Película” que difieren en sus listas de parámetros, ya que una está vacía y la otra tiene 3 parámetros. Como ya explicamos con antelación, diremos que el constructor está *sobrecargado*.

2. El uso del identificador “this”. El identificador “this” sólo tiene sentido dentro de la definición de una clase (en este caso, dentro de la definición de la clase

Película), y más concretamente, en la definición de los métodos (no estáticos) de la misma. “this” actúa como una referencia o un puntero al objeto sobre el cual ha sido invocado el método correspondiente de la clase (esta situación la observaremos mejor cuando definamos un programa principal en el que se construyan diversos objetos). Por último, su uso es obligatorio, únicamente, cuando haya un parámetro en el mismo ámbito que tenga el mismo nombre que un atributo de la clase.

Ejemplo:

setDirector (cadena director) {this.director = director;};

¿Qué pasaría en la anterior definición si omitimos el identificador “this”?

La forma en la que los lenguajes de programación nos permiten diferenciar entre el atributo de la clase (this.director) y la variable local que se le pasa al parámetro (director) es por medio del uso explícito de la referencia this.

1.7 ATRIBUTOS Y MÉTODOS ESTÁTICOS O DE CLASE

Existe otro tipo de atributos reseñables cuyo comportamiento es un poco distinto al de los atributos vistos hasta ahora; son los conocidos como atributos estáticos.

Los atributos estáticos son aquellos que poseen un único valor para todos los objetos (o instancias) de una clase determinada.

El ejemplo típico de atributo estático son las constantes (de clase). Una constante de una clase puede ser implementada como un atributo que posea un mismo valor para todos los objetos de dicha clase. Por lo tanto, podría ser implementada por medio de un atributo estático.

Un ejemplo podría provenir de la propia clase Circunferencia que hemos definido anteriormente:

```
Clase Circunferencia {

        //Atributos estáticos:
        real PI = 3.14;

        //Atributos
        real radio;

        //Constructores de clase

        Circunferencia (){this.radio = 0;};
        Circunferencia (real radio){this.radio = radio;};

        //Métodos
        real getRadio (){return this.radio;};
```

```
        real getDiametro (){return 2 * (this.radio);};

        real getArea (){return 3.14 * (this->radio) * (this.radio);};

        real getLongitud (){return 2 * 3.14 * (this.radio);};

        void setRadio (real nuevo_radio){this.radio = nuevo_radio;};

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

        void setArea (real nuevo_area){this.radio = (sqrt (nuevo_area / PI));};

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

}
```

Sin embargo, los atributos estáticos no sólo nos sirven para representar constantes de una clase. También nos sirven para representar atributos de dicha clase. Un atributo de clase es un atributo cuyo valor es igual para todos los objetos o instancias que haya definidos de la misma. Por ejemplo, si queremos guardar información sobre el número de objetos de una clase que han sido creados, dicho valor es compartido por todos los objetos de la clase.

Esto nos lo permitiría la declaración del mismo como estático:

```
Clase Circunferencia {

        //Atributos estáticos:
        entero contadorCircunferencias = 0;
        real PI = 3.14;

        //Atributos
        real radio;

        //Constructores de clase

        Circunferencia (){
                this.radio = 0;
                contadorCircunferencias = contadorCircunferencias + 1;
        };

        Circunferencia (real radio){
                this.radio = radio;
                contadorCircunferencias = contadorCircunferencias + 1;
        };

        //Métodos
        real getRadio (){return this.radio;};
```

```
        real getDiametro (){return 2 * (this.radio);};

        real getArea (){return 3.14 * (this->radio) * (this.radio);};

        real getLongitud (){return 2 * 3.14 * (this.radio);};

        void setRadio (real nuevo_radio){this.radio = nuevo_radio;};

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

        void setArea (real nuevo_area){this.radio = (sqrt (nuevo_area / PI));};

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

}
```

La implementación interna de los atributos estáticos en la mayor parte de los lenguajes de POO puede ayudar a comprender su comportamiento: de un atributo estático existe una única copia para todos los valores de la clase.

Así como atributos estáticos, existe también un tipo de métodos que no acceden al estado de ningún objeto. Estos métodos son conocidos como estáticos o métodos de clase. En particular, un método estático no puede acceder a ningún atributo no estático de una clase.

Existen diversas situaciones donde un método estático puede ser de utilidad. Por ejemplo, para implementar un método auxiliar que procese información externa a una clase. O también para acceder o modificar los atributos estáticos de la misma:

```
Clase Circunferencia {

        //Atributos estáticos:
        entero contadorCircunferencias = 0;
        real PI = 3.14;

        //Atributos
        real radio;

        //Constructores de clase

        Circunferencia (){
                this.radio = 0;
                contadorCircunferencias = contadorCircunferencias + 1;
        };

        Circunferencia (real radio){
                this.radio = radio;
                contadorCircunferencias = contadorCircunferencias + 1;
        };
```

```
        //Métodos
        real getRadio (){return this.radio;};

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

        real getArea (){return 3.14 * (this->radio) * (this.radio);};

        real getLongitud (){return 2 * 3.14 * (this.radio);};

        void setRadio (real nuevo_radio){this.radio = nuevo_radio;};

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

        void setArea (real nuevo_area){this.radio = (sqrt (nuevo_area / PI));};

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

        //Métodos estáticos:

        real getPI () {return PI;};

        real setPI (real nuevo_PI) { PI = nuevo_PI;};

        void contadorCircunferenciasNulo { contadorCircunferencias = 0;};

}
```

En la siguiente Sección vemos cómo una clase se puede seguir enriqueciendo con más información, ahora con modificadores sobre el acceso a sus parámetros.

1.8 MODIFICADORES DE ACCESO: RELEVANCIA Y NECESIDAD DE LOS MODIFICADORES PÚBLICO Y PRIVADO

Hasta ahora hemos incidido en la idea de que en una clase la información está dividida en atributos, que permiten representar el estado de un objeto, y métodos que nos permiten simular el comportamiento del mismo.

También hemos comentado que los métodos son la forma natural de comunicarse con un objeto, en lugar de usar directamente los atributos. Recuperamos ahora la clase “Circunferencia” que definimos en la Sección 1.4.

```
Clase Circunferencia {
        //Atributos
        radio real;

        //Declaramos ahora los constructores
        Circunferencia ();
        Circunferencia (real radio);
```

```
        //Métodos
        real getRadio ();
        void setRadio (real);
        real getDiametro ();
        void setDiametro (real);
        real getArea ();
        void setArea (real);
        real getLongitud ();
        void setLongitud (real);
    }
```

De este modo, si queremos conocer el radio de una circunferencia, lo que haremos será solicitárselo al objeto por medio de la consulta:

Una_circunferencia.getRadio ();

en lugar de utilizar directamente:

Una_circunferencia.radio;

¿Por qué?

Se pueden aportar varios motivos para esa elección.

1. En primer lugar, porque si cualquier cliente de nuestra clase puede acceder al atributo Una_circunferencia.radio directamente, podría también borrar ese dato, sobrescribirlo, ..., perdiendo una información relevante para nuestro programa. Es decir, el estado de los objetos que aparecen en el programa podría ser mutable por cualquier cliente o usuario del mismo.

2. En segundo lugar, ¿qué pasaría si el programador de la clase “Circunferencia” decide que su clase funcionará mejor con la siguiente representación?

```
Clase Circunferencia {

        //Atributos
        longitud real;

        //Constructores de la clase
        Circunferencia ();
        Circunferencia (real radio);

        //Métodos
        real getRadio ();
        void setRadio (real);
        real getDiametro ();
        void setDiametro (real);
        real getArea ();
```

```
        void setArea (real);
        real getLongitud ();
        void setLongitud (real);

    }
```

¿Cuál sería ahora el comportamiento de la invocación Una_circunferencia.radio?

¿Y de Una_circunferencia.getRadio();?

Cualquier modificación en la representación interna de los atributos de una clase obligaría a modificar todos los clientes de dicha clase, con el consiguiente costo que dicha modificación conllevaría.

En POO (y también en otros tipos de programación) hay una herramienta denominada *modificadores de acceso* que permite definir una clase de tal forma que ciertas partes de la misma (atributos y/o métodos) no sean accesibles desde fuera de la misma, y otras partes (atributos y/o métodos) sí que lo sean.

Siguiendo con el ejemplo anterior, el atributo “radio” lo haríamos “no accesible”, para evitar que los usuarios de la clase pudieran tanto accederlo como modificarlo; sin embargo, el método “getRadio()” lo haríamos “accesible” para que los clientes de la clase pudieran conocer el radio de una circunferencia.

Desde el punto de vista del programador de una clase, los dos siguientes requisitos se deben aunar en la misma:

1. Hay una parte de la clase que no debe ser accesible desde fuera de la misma. Esta parte será la parte *privada* de la clase. Generalmente está formada por los *atributos* y por los *métodos auxiliares*.

2. Hay una parte de la clase que los usuarios deben ser capaces de utilizar para comunicarse con los objetos de la misma (si no, ¿para qué sirve dicha clase?). Esta parte de la clase es conocida como parte *pública* o *interfaz* de la misma. Por ejemplo, alguno de los constructores debería formar parte de ella (si los usuarios de nuestra clase no pueden crear objetos de la misma, ¿cómo la podrán utilizar?). Pregunta: ¿todos los constructores deberían formar parte de la misma?

Veamos ahora cómo podría quedar la definición de la clase Circunferencia con las consideraciones anteriores:

```
Clase Circunferencia {

        //Atributos
        privado:        radio real;

        //Añadimos ahora los constructores
```

```
        público:    Circunferencia ();
        público:    Circunferencia (real radio);

        //Métodos
        público:    real getRadio ();
        público:    void setRadio (real);
        público:    real getDiametro ();
        público:    void setDiametro (real);
        público:    real getArea ();
        público:    void setArea (real);
        público:    real getLongitud ();
        público:    void setLongitud (real);

    }
```

La división entre métodos públicos y privados es una decisión del programador, y suele seguir las indicaciones que hemos expuesto antes (privados los atributos de la clase y los métodos auxiliares, públicos los métodos que se han de usar fuera de la misma).

La parte pública de una clase actúa como una especie de envoltorio que permite enmascarar la parte privada de la misma. Para los clientes de la misma, sólo la parte pública es relevante. El programador de la clase debe responsabilizarse de que la parte privada y la parte pública mantengan la coherencia y asegurar que el comportamiento de la clase se mantenga.

Sin embargo, hay casos en los que esta división entre público y privado se convierte en una decisión de matiz. ¿Qué sucedería ahora si nuestra clase tuviera una constante llamada PI para albergar el valor 3.1416? ¿Deberían los usuarios de la clase poder acceder a dicha constante para conocer el valor que le hemos asignado? ¿Deberían poder modificarla, por ejemplo, para dotarla de mayor precisión?

```
Clase Circunferencia {

        //Constantes de clase
        constante privado: PI real = 3.1415;

        //Atributos
        privado:    radio real;

        //Añadimos ahora los constructores
        público:    Circunferencia ();
        público:    Circunferencia (real radio);

        //Métodos
        público:    real getRadio ();
        público:    void setRadio (real);
        público:    real getDiametro ();
        público:    void setDiametro (real);
```

```
    público:    real getArea ();
    público:    void setArea (real);
    público:    real getLongitud ();
    público:    void setLongitud (real);

}
```

¿Tendría sentido disponer de un método getPI()? ¿Debería ser público o privado? ¿Tendría sentido disponer de un método setPI()?

Del mismo modo, ¿qué supondría cambiar el modificador del constructor sin parámetros ”Circunferencia()”? ¿En qué afectaría al funcionamiento de la clase?

Conclusión: los modificadores de acceso (privado y público) nos permiten separar las partes que corresponden a la implementación interna de una clase (atributos, métodos auxiliares) de las partes que deben servir como interfaz de la misma (métodos) para los usuarios y clientes de dicha clase. De esta forma, se aumenta la seguridad de los datos que contiene una clase (sólo el programador de la clase puede cambiar los atributos, no los usuarios). La idea de “esconder” la implementación interna de una clase nos permite hablar de “ocultación de la información”, que es el eje de la siguiente Sección.

Nota: Aun hay más modificadores de acceso que veremos más adelante (como protected, que lo veremos en la Sección 2.7, o package, que es exclusivo de Java).

1.9 OCULTACIÓN DE LA INFORMACIÓN: DISTINTAS FORMAS DE REPRESENTAR UNA MISMA CLASE MANTENIENDO SU COMPORTAMIENTO

La ocultación de la información es tenida como una de las grandes claves de la POO (y una de sus grandes ventajas con respecto a otros paradigmas de programación).

La idea de la ocultación de la información sería la siguiente: si un programador crea una clase con unos atributos cualesquiera, y en un momento determinado decide modificarlos, los usuarios de dicha clase no deberían observar ningún cambio en el comportamiento (o en la interfaz o parte pública) de la misma.

Dicho de otro modo, aunque los atributos (parte privada) de una clase sufran modificaciones, los métodos de la misma (parte pública) deben mantener el comportamiento de la misma.

Una vez que hemos conseguido ocultar los datos (o atributos) de una clase, podemos decir que la información de la misma está oculta (al menos, para los usuarios externos de la clase).

Para ilustrarlo con un ejemplo, recuperamos la definición (no la declaración) de la clase Circunferencia:

```
Clase Circunferencia {

        //Constantes de clase
        constante privado: PI real = 3.1415;

        //Atributos
        privado:        radio real;

        //Añadimos ahora los constructores
        público:        Circunferencia () {
                this.radio = 0;
        };
        público:        Circunferencia (real radio){
        this.radio = radio;
        };

        //Métodos
        público:        real getRadio (){
                return this.radio;
        };

        público:        void setRadio (real nuevo_radio){
                this.radio = nuevo_radio;
        };

        público:        real getDiametro (){
        return (2 * this.radio);
        };
        público:        void setDiametro (real nuevo_diametro){
        this.radio = nuevo_diámetro / 2;
        };

        público:        real getArea (){
        return (this.radio * PI * PI);
        };

        público:        void setArea (real nuevo_area){
                this.radio = raiz (nuevo_area / PI);
        };

        público:        real getLongitud (){
                return (this.radio * 2 * PI);
        };

        público:        void setLongitud (real nueva_longitud){
                this.radio = nueva_longitud / (2 * PI);
        };
}
```

Como se puede observar en la definición de la clase, hemos ajustado el comportamiento de los distintos métodos (métodos de acceso y modificación) para que se comporten de forma correcta con respecto al atributo que hemos definido en nuestra clase (el radio).

Supongamos que ahora, por exigencias de cualquier tipo, o simplemente por decisión personal, decidimos modificar la representación de la clase anterior. Evidentemente, la parte pública de la misma debe ser mantenida (es la parte que utilizan todos los programadores para comunicarse con lo objetos de la clase).

Por tanto, veamos cómo hemos de realizar dichos cambios. Supongamos que decidimos guardar la longitud de la circunferencia como atributo, en lugar del radio. La definición de la clase sería la siguiente:

Clase Circunferencia {

```
//Constantes de clase
constante privado: PI real = 3.1416;

//Atributos
privado:    longitud real;

//Definimos ahora los constructores
público:    Circunferencia () {
        this.longitud = 0;
};
público:    Circunferencia (real radio){
        this.longitud = (radio * 2 * PI);
};

//Métodos de acceso y modificación
público:    real getRadio (){
        return this.longitud / (2 * PI);
};

público:    void setRadio (real nuevo_radio){
        this.longitud = nuevo_radio * 2 * PI;
};

público:    real getDiametro (){
        return (this.longitud / PI);
};
público:    void setDiametro (real nuevo_diametro){
this.longitud = nuevo_diámetro / PI;
};

público:    real getArea (){
return ((this.longitud * this.longitud) / 4 * PI);
};
```

```
        público:        void setArea (real nuevo_area){
                this.longitud = 2 * raiz (nuevo_area * PI);
        };

        público:        real getLongitud (){
                return (this.longitud);
        };

        público:        void setLongitud (real nueva_longitud){
                this.longitud = nueva_longitud;
        };
}
```

Observaciones sobre los cambios anteriores:

1. Como se puede ver, los cambios que hemos realizado en la definición de la clase sólo afectan a la parte privada (de un atributo **radio** hemos pasado a un atributo **longitud**) y a la implementación de los métodos. Ningún cambio se ha realizado sobre la **cabecera de los métodos.** Las cabeceras de los métodos son las que proporcionan la interfaz o parte pública de la misma, y por lo tanto deben ser preservadas. Sólo los cuerpos de los métodos pueden sufrir variaciones.

2. Por medio de la idea de la ocultación de la información, lo que perseguimos es que una clase a la que le modificamos los atributos, siga teniendo el mismo comportamiento. ¿Eso cómo se consigue? Por medio de la modificación del **cuerpo de los métodos**. El cuerpo de los métodos puede sufrir tantas variaciones como consideremos necesarias, en tanto en cuanto el comportamiento de los mismos sea idéntico.

Algunos motivos por los cuales se modifican los atributos de una clase:

1. Porque se comprueba que un método es usado de forma más frecuente que otros (por ejemplo, en el caso anterior, si “longitud” se usa más a menudo que “radio”, guardando el atributo “longitud” optimizaremos el uso de la clase circunferencia, ya que reducimos el número de operaciones mateméticas a realizar).

2. Porque decidimos hacer un cambio de en la representación del mismo. Por ejemplo, un campo cadena, en C++, en lugar de usar un char [20] decidimos pasarlo a un string porque eso va a simplificar algunos métodos.

3. Por necesidades de la aplicación. Si tenemos un atributo “char [30]” y comprobamos que hay muchos casos en los que debemos representar cadenas de más caracteres, el atributo debería pasar a ser “char [40]”. ¿Deberían los usuarios de la clase percibir tal cambio?

1.10 INTRODUCCIÓN AL LENGUAJE DE ESPECIFICACIÓN UML: UTILIZACIÓN PARA REPRESENTAR CLASES Y OBJETOS

UML son las siglas en inglés de Lenguaje Unificado de Modelado (Unified Modelling Language). Lo primero que debe quedar claro es que UML no es un lenguaje de programación, sino de especificación.

Esto quiere decir que en UML no se puede detallar el comportamiento de los métodos y funciones que nuestro programa pueda poseer.

UML sirve para representar un sistema software, sus distintas entidades, las relaciones existentes entre las mismas, y se utiliza sobre todo para documentar un programa.

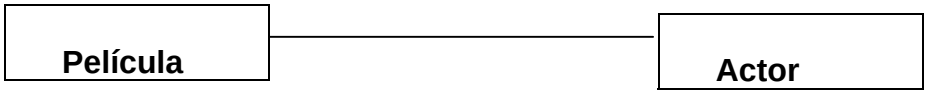
De los distintos diagramas que ofrece UML, nosotros vamos a estudiar por el momento únicamente los diagramas de clases. Otros tipos de diagramas que se pueden realizar son los diagramas de actividad, de despliegue, de casos de uso, de secuencia (muchos de ellos tendréis oportunidad de conocerlos en "Ingeniería del Software").

Los diagramas de clases se centran en la estructura de un programa. Generalmente, se realizan en la fase de Análisis (o a veces en el Diseño) de un proyecto.

Veamos cómo se puede representar una clase en UML. Una de las peculiaridades de UML es que existen diferentes niveles de detalle para representar las clases, según cuál sea nuestro objetivo. Por ejemplo:

1. La representación puede ser tan simple como una caja que contenga únicamente el nombre de la clase:

La anterior sería una representación válida de la clase Pelicula. Evidentemente, el diagrama anterior no aporta excesiva información sobre el sistema software que se quiere representar, aunque sí que nos indica ya que habrá una clase llamada "Película" en el mismo. El siguiente tendría un poco más de información:



En el diagrama anterior se pueden observar dos clases, "Pelicula" y "Actor", y una línea que expresa que entre ambas clases hay una relación. Las relaciones entre clases las estudiaremos en el Tema 2, y entonces veremos también cómo representarlas más detalladamente en UML.

2. En un segundo nivel de detalle, podemos representar una clase en UML junto con su lista de atributos. Esto se pude deber a que quizá no hayamos definido todavía la lista de métodos que va a tener dicha clase. La representación sería entonces:

Película
- titulo: string - director: string - duración: int
....

Dentro de la lista de atributos, se puede observar que hemos etiquetado cada uno de los nombres con un símbolo "-" delante. Dicho símbolo significa que el atributo que sigue es de tipo privado (siguiendo la definición que dimos en la Sección 1.8).

Finalmente, cuando ya todos los métodos de una clase han sido especificados, el aspecto que tiene el diagrama UML de una clase será el siguiente. La clase consiste de una caja dividida en tres partes, la primera de ellas contiene el nombre de la misma, la segunda la lista de atributos con sus correspondientes modificadores (privado), y finalmente hay una tercera caja que contiene el nombre de los métodos de la clase, con su modificador de acceso correspondiente (en este caso público se representa con un +):

Película
- titulo: string - director: string - duración: int
+ Pelicula () + Película (string, string, int) + getTitulo (): string + setTitulo (string): void + getDirector (): string + setDirector (string): void + getDuracion (): int + setDuración (int): void

En cualquier caso, una representación de una clase en un diagrama de clase UML nunca contendrá información referente a cómo han sido programados los constructores o métodos de la misma. Como hemos dicho, es únicamente un lenguaje de especificación.

Ampliación: Una posible referencia para aprender más sobre UML sería "UML para programadores en Java", R.C. Martín, Prentice Hall 2004, o también el libro "UML : el lenguaje unificado de modelado" de Grady Booch, James Rumbaugh, Ivar Jacobson, en Addison Wesley 2000.

1.11 LENGUAJE DE PROGRAMACIÓN C++: DECLARACIÓN DE CLASES Y CONSTRUCCIÓN DE OBJETOS

Breve introducción a C++:

C++ es un lenguaje de programación diseñado en los años 80's por Bjarne Stroustrup.

(Referencia: El lenguaje de programación C++, Bjarne Stroustrup, Addison Wesley 2002 o 1993)

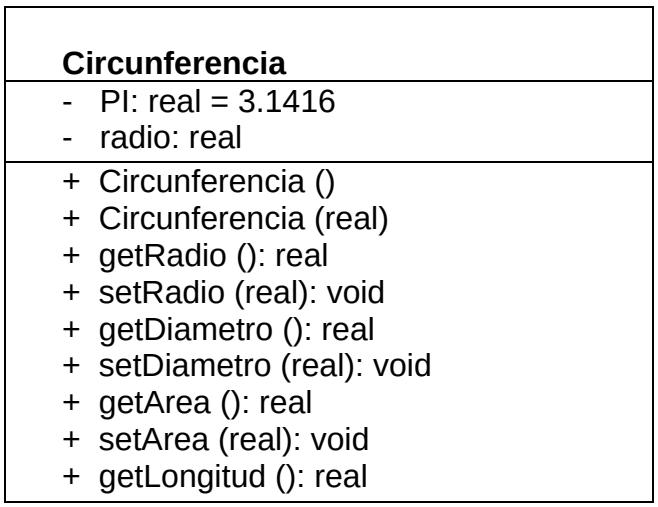
El propósito original de C++ era ampliar el lenguaje de programación C con las características propias de Programación Orientada a Objetos. Por lo tanto, C++ no es un lenguaje puro de orientación a objetos, sino que comparte características propias de la POO y de la programación estructurada (que ya conocéis).

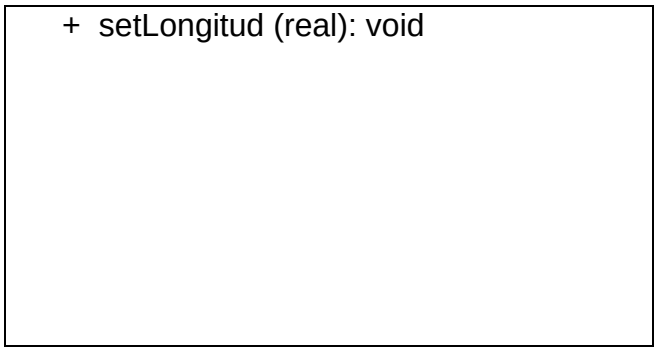
No vamos a dedicar especial atención a las características de C++ que no son propias de la POO, ya que las damos por aprendidas de las asignaturas ya cursadas. Esto incluye la sintaxis de los tipos de datos definidos por el lenguaje (int, double, char *, ...), las sentencias de control (for, while, ...), las órdenes de precompilación (include para hacer uso de librerías).

Para explicar la sintaxis particular de C++ en los conceptos hasta ahora vistos de POO lo que haremos será traducir los ejemplos que hemos ido viendo a lo largo del tema a C++.

Dentro de un lenguaje de programación es más propio introducir la definición de una clase, para pasar luego a ver cómo podemos construir objetos de la misma (aunque en la explicación teórica hemos pasado del concepto particular de objeto a su abstracción, que es la clase).

Partiremos de la especificación en UML de una de las clases vistas hasta ahora y veremos cómo se pude traducir a lenguaje C++:





La especificación anterior de una clase en UML se puede implementar en C++ de la siguiente forma (esta forma no es única, pero en general sí que se puede considerar como la más clara y sencilla de entender).

En primer lugar, generamos un archivo “Circunferencia.h” cuyo contenido será el siguiente. Los archivos “*.h” se denominan generalmente ficheros de cabeceras, y en ellos se puede encontrar la especificación de una clase:

```
#ifndef CIRCUNFERENCIA_H
#define CIRCUNFERENCIA_H 1

class Circunferencia{

    //Constantes
    private:
    const static double PI = 3.1416;

    //Atributos de Clase
    private:
    double radio;

    //Constructores de clase:
    public:
    Circunferencia ();
    Circunferencia (double);

    //Métodos de Clase
    double getRadio ();
    void setRadio (double);
    double getDiametro ();
    void setDiametro (double);
    double getArea ();
    void setArea (double);
    double getLongitud ();
    void setLongitud (double);

};

#endif
```

Los siguientes comentarios se pueden hacer sobre el anterior fichero de cabeceras:

1. Antes de empezar el fichero de cabeceras (y al final del mismo) hemos situado unas órdenes de precompilación de la siguiente forma:

```
#ifndef CIRCUNFERENCIA_H
#define CIRCUNFERENCIA_H 1
...
#endif
```

Las órdenes previas de precompilación lo que hacen es, en primer lugar, comprobar que no haya ninguna variable de precompilación denominada “CIRCUNFERENCIA_H” (#ifndef CIRCUNFERENCIA_H). La situación normal es que la variable “CIRCUNFERENCIA_H” no esté definida; entonces el comando (#define CIRCUNFERENCIA_H 1) la define, le asigna el valor 1, y se lee la especificación de la clase circunferencia.

Si ya existía dicha variable, el resto del código hasta el comando #endif se omite. Eso querría decir que el precompilador ya habría encontrado un fichero CIRCUNFERENCIA_H en el que las cabeceras de la clase Circunferencia ya están definidas, y entonces el fichero “Circunferencia.h” no debería ser leído. Leerlo produciría un fallo de compilación ya que las cabeceras de dicha clase se habrían incluido dos veces.

La razón de ser de estos comandos es que la precompilación de C++, si encuentra la misma definición de un método en dos ficheros *.h, produce un error de compilación. Cuando uno trabaja con múltiples ficheros que interactúan unos con otros, la situación anterior es muy probable (por ejemplo, hacer uso de dos ficheros que utilizan la misma librería). Para evitar que una misma cabecera sea especificada 2 veces, se suele utilizar en todos los archivos de cabeceras (es decir, donde se especifican funciones, clases y métodos), los comandos anteriores

```
#ifndef NOMBRE_DE_LA_CLASE_H
#define NOMBRE_DE_LA_CLASE_H 1
...
#endif
```

A partir de ahora deberías introducirlo en todos tus ficheros de cabeceras simplemente como una medida de seguridad de que la especificación de las clases que definas sólo se incluirá una vez.

2. La palabra clave en C++ para declarar una clase es “class” (fácil de recordar!). En realidad, la palabra reservada “class” sólo va a aparecer en el archivo de cabeceras al principio de la declaración (no la definición!) de la clase.

3. Después de la palabra class aparece el nombre de la clase que estemos definiendo. El nombre de las clases debe ser clarificador del contenido de la

clase. Recuerda que además el nombre de la clase va a ser el nombre que tengan los constructores de la misma. En nuestro caso, el nombre elegido es “Circunferencia”.

4. Lo siguiente que aparece en la definición de la clase es una llave, que delimita el principio de la declaración (de nuevo, no definición!!) de la misma.

La estructura hasta ahora sería:

```
#ifndef CIRCUNFERENCIA_H
#define CIRCUNFERENCIA_H 1

class Circunferencia {
        ...
};

#endif
```

Entre las dos llaves debe ir la declaración de constantes, atributos y métodos de la clase. Otra condición importante dentro de la sintaxis de C++ es que al final de la declaración de la clase debe ir situado un “;”, punto y coma, como finalizador de la definición de la misma. En caso contrario ...

5. Con respecto a la declaración de constantes, atributos y métodos de la clase.

Las constantes de clase se declaran como:

```
//Constantes
private:
const static double PI = 3.1416;
```

En primer lugar, se puede observar el modificador de acceso “private: “ que equivale al “privado” que definimos anteriormente.

Adicionalmente, para definir una constante hay que utilizar el modificador “const” propio de C para definir constantes.

Para que la constante sea de clase, hace falta además utilizar el modificador “static”; el modificador static hace que exista una única copia del atributo “PI” para todos y cada uno de los objetos que se declaren como instancias de esta clase.

Como pretendemos declarar una constante, al mismo tiempo que la declaramos debemos inicializarla. Al ser una variable de clase, el valor de PI ya no podrá ser modificado en adelante.

Los atributos de clase se declaran como:

```
//Atributos de Clase
```

```
private:
double radio;
```

En este caso se puede ver como con asignar el modificador de acceso (private) y el tipo (double) e identificador (radio) del atributo, es suficiente.

Lo siguiente que hemos de definir son los constructores de la clase (recordar lo dicho en la Sección 1.5 sobre constructores de clase). Una clase puede tener tantos constructores como nosotros consideremos necesario (sobrecarga de métodos), siempre que su lista de parámetros sea diferente. El modificador de acceso “public:” nos hace ver que los constructores de dicha clase son de tipo público (Pregunta: ¿cuál sería la razón de ser de un constructor privado?).

En este caso contamos con dos constructores, uno de ellos sin parámetros y el otro con un único parámetro de tipo double.

```
//Constructores de Clase
public:
Circunferencia ();
Circunferencia (double);
```

Mientras no se indique lo contrario, los métodos y atributos que vengan a continuación serán de tipo “public”.

La lista de métodos siguiente corresponde con los especificados en UML:

```
//Métodos de clase
double getRadio ();
void setRadio (double);
double getDiametro ();
void setDiametro (double);
double getArea ();
void setArea (double);
double getLongitud ();
void setLongitud (double);
```

El fichero “Circunferencia.h”, como hemos indicado, sólo proporciona la especificación de los métodos de una clase. La definición de dichos métodos estará en el fichero “Circunferencia.cpp”, cuyo aspecto sería el siguiente:

```
#include <cmath>
#include "Circunferencia.h"

using namespace std;

//Constructores

Circunferencia::Circunferencia () {
   this->radio = 0.0;
```

```
};

Circunferencia::Circunferencia (double radio){
	this->radio = radio;
};

//Métodos

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

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

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

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

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

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

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

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

Algunos comentarios relativos al código que hemos mostrado:

1. Lo primero que debemos hacer en C++ es incluir la librería en la que hemos especificado la clase Circunferencia. En este caso, ésta sería “Circunferencia.h”.

2. A la hora de definir (no declarar!) los métodos, el orden no es relevante, aunque lo natural es empezar por los constructores:

```
Circunferencia::Circunferencia () {
```

```
    this->radio = 0.0;
};
```

El primer constructor que nos encontramos es el que hemos dado en llamar “constructor por defecto”, que no tiene ningún parámetro.

La forma de acceder al método, “Circunferencia::Circunferencia () {...}”, quiere decir que vamos a definir, dentro de la clase “Circunferencia”, el método “Circunferencia ()”. El uso de los “::” es un desambiguador de ámbito.

Los constructores deben definir el valor de los atributos de un objeto. En este caso, el único atributo con el que cuenta el objeto es “radio”. Como no tenemos ningún valor inicial especial que asignarle, supondremos que en principio todas las circunferencias tienen radio 0.0.

Todavía queda por aclarar el uso de “this->radio”. En este caso concreto, el uso de “this” sería opcional, ya que en el mismo contexto no hay ningún otro identificador de nombre “radio”. Veremos casos en los que su uso es obligatorio. En cualquier caso, su uso es siempre recomendable.

“this” hace referencia al objeto que estamos definiendo en ese momento. Una forma de entenderlo, sería leerlo como: En el objeto que estamos “creando”, el campo “radio” tendrá como valor “0.0”.

3. El segundo constructor que estamos definiendo tiene un único parámetro, que es el radio que le queremos asignar al objeto a la hora de crearlo:

```
Circunferencia::Circunferencia (double radio){
        this->radio = radio;
};
```

De nuevo se puede observar el uso de la sintaxis “Circunferencia::Circunferencia (double radio){...};” para señalar el método que estamos definiendo (el método “Circunferencia (double)” de la clase “Circunferencia”).

En este caso, el uso de “this” en “this->radio = radio” sí es obligatorio, ya que de otro modo colisionaría con la variable local “radio” existente en el mismo ámbito. (Como ejercicio podrías probar qué pasa en el caso de que no lo escribas)

4. El resto de métodos que encontramos en esta clase, métodos de acceso y modificación de los atributos, se definen de la misma forma que los constructores. Lo único relevante es que cada uno de los métodos debe aparecer con el tipo de su valor de retorno:

```
//Métodos

double Circunferencia::getRadio (){
        return this->radio;
```

```
};

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

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

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

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

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

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

Con lo anterior hemos terminado la declaración (fichero de cabeceras, *.h) y definición (fichero *.cpp) de la clase Circunferencia.

Con esto, el ejercicio se podría dar por completado. Aunque, evidentemente, una clase sirve de poco si no hay algún cliente (es decir, un programa externo) que haga uso de tal clase.

Veamos ahora como podemos definir un cliente que utilice la clase “Circunferencia”. En este caso el cliente será un programa principal (main en C++) que construya y opere con varios objetos de la clase “Circunferencia”.

Otro ejemplo de cliente podría ser una clase diferente que usara “Circunferencia” como uno de sus atributos (este tipo de ejemplos aparecerán en el Tema 2).

```
#include <iostream> //para los comandos cin, cout, ...
#include "Circunferencia.h" //para la declaración de la clase Circunferencia

using namespace std; //"Abrimos" el espacio de nombre "std",
                     //en el que están funciones como cin, cout, endl,...

int main (){ //Comienzo del programa cliente de la clase Circunferencia

	//Invocación al constructor sin parámetros
```

```
        //creando un puntero a una circunferencia:

        Circunferencia * punt_a_circ_radio_0 = new Circunferencia;

        //Invocación al constructor sin parámetros
        //creando un objeto circunferencia:

        Circunferencia circ_radio_0;

        //Invocación al constructor con un parámetro
        //creando un puntero a circunferencia de radio 3

        Circunferencia * punt_a_circ_radio_3 = new Circunferencia (3.0);

        //Invocación al constructor con un parámetro
        //creando un objeto circunferencia de radio 3

        Circunferencia circ_radio_3 (3.0);

        cout << "La circ. tiene radio " << punt_a_circ_radio_0->getRadio ();
        cout << endl;
        cout << "La circunferencia tiene radio " << circ_radio_0.getRadio ();
        cout << endl;

        cout << "La circ. tiene radio " << punt_a_circ_radio_3->getRadio ();
        cout << endl;
        cout << "La circ. tiene radio " << circ_radio_3.getRadio ();
        cout << endl;

        //A modo de comprobación, puedes ver lo que sucede al acceder a algún
        //atributo de declarado "private" de la clase:
        //circ_radio_3.radio;

        //Del mismo modo, puedes intentar ver los problemas que
        //ocasiona modificar un atributo constante:

        //circ_radio_3.PI++;

        //En la anterior sentencia encontramos dos errores
        //Uno por intentar acceder a un atributo "private"
        //El segundo por intentar modificar el valor de una constante

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

A medida que vayamos avanzando el curso veremos ejemplos más elaborados de utilización de clases en métodos y de acceso a las mismas. Por el momento, las ideas más importantes que debes retener son:

1. La sintaxis propia de C++ para la construcción de objetos (la diferencia entre el uso de punteros a objetos o de objetos, con las consecuencias que ello conlleva sobre el uso de “new”)

2. La utilización de los modificadores de acceso, “private”, “public”, dentro de la declaración de una clase, y la diferencias que esto implica en el acceso a atributos y métodos por parte de los clientes de dicha clase

1.12 LENGUAJE DE PROGRAMACIÓN JAVA: DECLARACIÓN DE CLASES Y CONSTRUCCIÓN DE OBJETOS

1.12.1 INTRODUCCIÓN A JAVA: ORIGEN Y DESCRIPCIÓN DEL LENGUAJE

Java (http://www.sun.com/java) es un lenguaje de programación desarrollado por la empresa Sun Microsystems (http://www.sun.com). La primera versión del mismo apareció en el año 1995. Algunas de las características principales de Java son que hereda la mayor parte de su sintaxis de lenguajes como C y C++, pero partiendo desde el origen del mismo de un diseño basado en los principios de la POO. Otro de sus características principales es que enmascara parte de las funcionalidades de bajo nivel que ofrecía C++ (por ejemplo, uso explícito de punteros, ...).

Con el paso del tiempo Java se ha convertido en un lenguaje ampliamente extendido tanto en el ámbito de la educación (debido a su simplicidad original que lo convierte en un buen candidato como primer lenguaje de programación a aprender) como en el de la industria, por su potencia para desarrollar grandes aplicaciones de una forma estructurada, su capacidad para reutilizar código, y otra de sus conocidas virtudes, el hecho de ser *multiplataforma*.

Para adentrarnos un poco más en el concepto de lo que quiere decir un lenguaje multiplataforma, haremos una sencilla comparación de un programa en C++ y un programa Java. Cuando compilamos un programa en C++, el resultado es una serie de ficheros “*.o” que es lo que llamamos generalmente “ficheros objeto”. Estos ficheros sufren luego un proceso de “linkado”, en el que se les unen ciertas librerías y, en una máquina Windows, dan lugar a un fichero “*.exe”. Los propios ficheros “*.o” ya se pueden considerar “código máquina”, y cómo sean ejecutados depende de la máquina que tengamos debajo (Unix, Windows, Mac, ...).

Sin embargo, Java es lo que se conoce como un lenguaje semi-interpretado. Esto quiere decir que hay un intérprete instalado en nuestra máquina que es el encargado de “interpretar” los programas en Java. Entonces, el proceso de ejecución de un programa en Java es como sigue. Nuestro código Java estará escrito generalmente en un fichero “*.java”. Este fichero “*.java”, al compilar, da lugar a un fichero “*.class”. Este fichero “*.class” es el que se ejecuta, pero no directamente sobre nuestra máquina (por lo cual no es dependiente de la plataforma en la que nos hallemos), sino sobre lo que Java ha dado en llamar “Java Virtual Machine“ (o JVM). La JVM está disponible para diversos sistemas

operativos (Windows, Unix, Mac,...), y plataformas (por ejemplo, sistemas operativos corriendo en móviles o Palms). Esta facilidad para portar el código escrito en Java ha sido una de las razones de su rápido establecimiento en el mercado, dando lugar a la máxima popularizada por Sun de “write once, run anywhere” (http://en.wikipedia.org/wiki/Write_once,_run_anywhere).

Lo anterior es cierto con algunos matices, ya que, por ejemplo, la versión de Java aceptada por dispositivos móviles es la Java ME, y tiene ciertas limitaciones con respecto a la versión estándar de un ordenador de sobremesa, que es la Java SE; en este curso no prestaremos atención a ese tipo de consideraciones.

1.12.2 PRIMEROS EJEMPLOS DE SINTAXIS EN JAVA

Después de la breve introducción sobre características generales de Java, pasamos a consideraciones propias de la programación en Java. Siendo Java un lenguaje orientado a objetos, veamos para empezar qué aspecto tendría la clase Circunferencia introducida en la Sección 1.10 en UML programada en Java.

La primera diferencia fundamental entre Java y C++ es que en Java la forma natural de programar una clase es dar al mismo tiempo su declaración y definición (a diferencia de C++, donde creábamos un fichero de cabeceras “*.h” y otro de definición de la clase, “*.cpp”).

El fichero con el que nos encontramos en este caso es un fichero “Circunferencia.java”. La extensión “*.java” es la propia de los ficheros del lenguaje:

```
import java.lang.Math;

public class Circunferencia {

	//Declaración e inicialización de constantes
	private static final double PI = 3.1416;

	//Declaración de atributos
	private double radio;

	//Declaración y definición de constructores
	public Circunferencia(){
		this.radio = 0.0;
	}

	public Circunferencia(double radio){
		this.radio = radio;
	}

	//Declaración y definición de métodos de acceso y modificación
	public double getRadio(){
```

```
		return this.radio;
	}

	public void setRadio(double nuevo_radio){
		this.radio = nuevo_radio;
	}

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

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

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

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

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

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

}
```

Lo primero que podemos observar en el fichero “Circunferencia.java” es el comando:

```
import java.lang.Math;
```

El comando “import” en Java es el equivalente en C++ a “#include”. A pesar de ello, se puede observar que aquí la orden “import” no está realizada como una orden de precompilación. Podemos encontrar una primera diferencia entre ambos lenguajes, ya que Java no posee un lenguaje de precompilación.

La estructura del código en dicha clase mantiene similitudes con las de los ficheros propios de C++ (aunque la sintaxis es distinta en algunos puntos). En primer lugar, nos encontramos con el siguiente armazón:

```
public class Circunferencia {
…
}
```

Las dos llaves de principio y fin marcan el comienzo y el fin de la definición de la clase correspondiente. ¿Cómo se hace lo anterior en la definición de Tipos Abstractos de Datos que conoces en C++? De ahí una de las principales características de la POO, que es la modularidad (o lo que también hemos llamado encapsulación de la información) que aporta en la definición de clases, y que ya introdujimos en la Sección 1.3.

También se puede observar una segunda diferencia con C++, y es que hemos declarado nuestra clase como “public”. Esto en C++ no era posible, las clases tenían partes privadas o públicas, pero las clases como tales no tenían modificadores de acceso. En realidad el modificador de acceso “private” en una clase no tendría mucho sentido, porque ningún cliente podría hacer uso de ella. Sin embargo, las clases en Java admiten otro modificador de acceso (“package”), que es el modificador de acceso por defecto para clases. Por el momento, podemos identificar “package” con la carpeta donde está alojado nuestro proyecto, mientras que “public” quiere decir que nuestra clase es accesible desde cualquier parte. Una consecuencia de declarar nuestra clase “Circunferencia” como “public” es que nuestro fichero ha de ser llamado igual que la clase, “Circunferencia.java”.

Entre las dos llaves de principio y fin de la clase debemos incluir la declaración e inicialización de constantes, la declaración de atributos y la definición de los métodos de la clase (notar de nuevo que declaraciones y definiciones van unidas):

```
//Declaración e inicialización de constantes
private static final double PI = 3.1416;

//Declaración de atributos
private double radio;
```

En la declaración de atributos y constantes se puede observar cómo cada uno de los atributos correspondientes lleva su modificador de acceso, en este caso “private”, y sin usar “:”, como se hacía en C++.

Una segunda diferencia está en los modificadores que nos permiten definir una constante de clase. En este caso son “static final”. El modificador “const” no existe en Java.

El modificador “static” para un atributo significa que ese atributo estará disponible para todos los objetos de la clase con el mismo valor (no hay una copia por objeto de la clase, sino una única copia para todos los objetos de la misma). Una forma de comprobar la anterior afirmación es acceder a la constante de clase “PI” por medio de la sintaxis “Circunferencia.PI”, o de “this.PI”, y ver que obtenemos el mismo resultado. Una única copia de la constante existe para todas las instancias de la misma, y en particular para la clase.

El modificador “final” hace que el atributo no pueda ser modificado después de su inicialización. Por tanto, un atributo “static final” es un atributo del que sólo hay una copia en la clase, y cuyo valor no puede ser modificado. Las anteriores propiedades convierten a dicho atributo en una constante de clase.

Veamos ahora las principales características de los constructores:

```
//Declaración y definición de constructores
public Circunferencia(){
        this.radio = 0.0;
}

public Circunferencia(double radio){
        this.radio = radio;
}
```

La mayor diferencia apreciable con C++ es que en Java los atributos se acceden como “this.radio”, en lugar de “this->radio” (observar la notación propia de C++ en la Sección 1.11). Esta pequeña diferencia de notación entre C++ y Java corresponde a una diferencia mayor entre ambos lenguajes a nivel conceptual. El modificador “this”, que en C++ actuaba como un puntero o una referencia al objeto sobre el cual se había llamado a un determinado método (y de ahí que lo usáramos con el operador “->”, propio de punteros en C++), se enmascara en Java como si fuera un objeto, y de ahí el uso de la notación “.” para acceder a sus métodos y atributos.

La filosofía general en Java con respecto a la definición y uso de objetos sería la siguiente: todos los objetos en Java son guardados en memoria por medio de referencias (o punteros) a dichos objetos. Sin embargo, estos objetos son “desreferenciados”, es decir, convertidos a la dirección de memoria que ocupan, cada vez que intentamos acceder a alguna de sus propiedades o métodos. De ahí que la forma de acceder a su estado o comportamiento sea a través del “.”.

Lo que se consigue de este modo es que todos los objetos existentes en un programa estén guardados en memoria como “punteros” a los mismos, optimizando el uso de memoria y facilitando ciertas operaciones propias de la POO (como el polimorfismo, Tema 3, la reescritura de métodos, Tema 2, o la liberación de memoria).

Por otra parte, supone una pequeña desventaja con respecto a C++, ya que C++ permite la declaración y uso tanto de punteros (objetos guardados por referencia) como de objetos guardados por valor.

Volveremos a incidir en este hecho cuando hablemos del polimorfismo en Java y C++ en el Tema 3.

Una vez explicado lo anterior, la definición de los métodos de acceso y modificación no tiene especiales dificultades:

```
	//Declaración y definición de métodos de acceso y modificación
	public double getRadio(){
		return this.radio;
	}

	public void setRadio(double nuevo_radio){
		this.radio = nuevo_radio;
	}

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

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

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

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

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

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

El aspecto final que tiene la clase Circunferencia tras las anteriores consideraciones sería:

```
import java.lang.Math;

public class Circunferencia {

	//Declaración e inicialización de constantes
	private static final double PI = 3.1416;

	//Declaración de atributos
	private double radio;

	//Declaración y definición de constructores
	public Circunferencia(){
		this.radio = 0.0;
	}
```

```
	public Circunferencia(double radio){
		this.radio = radio;
	}

	//Declaración y definición de métodos de acceso y modificación
	public double getRadio(){
		return this.radio;
	}

	public void setRadio(double nuevo_radio){
		this.radio = nuevo_radio;
	}

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

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

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

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

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

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

Del mismo modo que hicimos en C++, debemos crear ahora un cliente para nuestra clase que sea el que utilice los objetos de esta clase (los cree y manipule). De nuevo el cliente será el programa “main” (también veremos en el Tema 2 como unas clases pueden hacer de clientes de otras) que presentamos a continuación.

En primer lugar, hay que tener en cuenta que en Java cualquier fragmento de código debe ser parte de una clase (ésta es una de las razones por las que decíamos que Java es más orientado a objetos en su diseño de lo que era C++). Por tanto, el método “main” ha de estar metido en una clase. En nuestro

ejemplo hemos creado un fichero “Principal.java” dentro de la cual declaramos una clase “Principal”:

```
public class Principal {
…
}
```

Una vez creada esta clase, en su interior definimos el método “main”. Dicho método tiene las siguientes peculiaridades que conviene recordar:

1. En primer lugar, este método debe ser declarado como “public” ya que ha de ser accesible desde fuera de esta clase. Si no, no podríamos ver su comportamiento al ejecutar el código.

2. El método ha de ser declarado como “static”, ya que el método pertenece a la clase “Principal”, no a ninguno de los objetos de dicha clase. Visto de otra forma, ¿se crea algún objeto o instancia de la clase “Principal”? La respuesta es que no. En este caso, la única forma de que el método se pueda utilizar es declarándolo “static”.

3. El tipo de retorno del método es “void”, ya que no devuelve ningún valor.

4. Finalmente, el parámetro “(String [] args)” indica que al método “main” se le pasa un vector de cadenas de caracteres como parámetro inicial. Por lo general no haremos uso del paso de parámetros formales a programas, pero podría resultar de utilidad en el futuro.

La clase principal con la cabecera del método “main” queda como sigue:

```
public class Principal {

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

Veamos ahora el aspecto del método “main”:

```
public class Principal {

	public static void main (String [] args){
	//Construcción de objetos
	Circunferencia circ_radio_0 = new Circunferencia ();
	Circunferencia circ_radio_3 = new Circunferencia (3.0);

	//Uso de los objetos
	System.out.println ("El objeto circ_radio_0 tiene como radio "
		+ circ_radio_0.getRadio());
	System.out.println ("El objeto circ_radio_3 tiene como radio "
		+ circ_radio_3.getRadio());
```

```
        }
}
```

Como principal hecho relevante en comparación a lo hecho en C++, se debe observar de nuevo la ausencia de punteros en nuestro código. Se define un objeto “circ_radio_0”, que se construye por medio del constructor sin parámetros y para el cual se reserva memoria de forma dinámica por medio del uso de “new”.

Se puede observar aquí una sutil diferencia entre C++ y Java, ya que en el primero, cuando queremos reservar espacio en memoria para un puntero a un objeto haciendo uso del constructor sin parámetros, la invocación la realizamos como “new Circunferencia; //(C++)” mientras que en Java se realiza por medio de “new Circunferencia (); // (Java)”. En Java es más evidente que estamos llamando a un método sin parámetros que en la notación de C++.

Posteriormente, definimos un nuevo objeto “circ_radio_3” que construimos por medio del constructor con un parámetro al que le pasamos el valor “3.0”.

Para hacer uso de los métodos de dichos objetos, se puede observar que no usamos la sintaxis de C++ para punteros, “->” sino el “.” propio de los objetos.

En el fragmento de código anterior correspondiente al método “main” se pueden observar las dos situaciones que mencionábamos al principio de esta Sección cuando hablábamos de la gestión de memoria en Java.

Primero, la reserva de memoria para un objeto cuando lo creamos se realiza por medio del comando “new”, que indica que reservamos memoria dinámica para el mismo.

En segundo lugar, cuando hacemos uso de sus métodos y atributos, utilizamos la notación propia de objetos (y de registros), por medio del operador “.”. Esto nos sirve para hacer hincapié en la idea ya comentada de que en Java los objetos son guardados en memoria por medio de referencias a los mismos, pero esas referencias son transparentes al programador, ya que el propio lenguaje desreferencia los objetos cuando hacemos uso de los mismos.

1.12.3 GESTIÓN DE LA MEMORIA EN JAVA

Siguiendo con la gestión de memoria en Java, otra de las diferencias importantes entre Java y C++ es la relativa a la gestión de memoria de los programas.

En C++ es el usuario el que se debe encargar de gestionar la reserva y también la liberación de memoria de los programas; en particular, siempre que una variable (u objeto) haya sido alojada utilizando memoria dinámica (es decir, usando punteros y por medio del comando “new”), si queremos “liberar” la zona de memoria reservada para esa variable, hemos de utilizar la sintaxis:

```
Circunferencia * punt_circ_radio_0 = new Circunferencia;
```

...

delete punt_circ_radio_0;

El comando “delete” no “borra” la variable “punt_circ_radio_0”, sino que lo que hace es marcar la zona de memoria ocupada por dicha variable como no reservada, lo cual quiere decir que posteriormente podría ser reescrita. Esto debería prevenirnos de volver a utilizar la variable “punt_circ_radio_0”, ya que su comportamiento tras ejecutar el comando “delete” sería imprevisible.

La gestión de memoria en Java es un tanto más simple desde el punto de vista del programador de aplicaciones. Ya estamos familiarizados con el comando “new” en Java para reservar memoria de forma dinámica para todo objeto que construyamos en nuestros programas (recuerda que en Java no existe la posibilidad de definir objetos o punteros a objetos, sino que todo objeto es implementado en memoria como una referencia al mismo). Java dispone de un mecanismo de gestión de memoria conocido como “garbage collector” (o recolector de basura) que se encarga automáticamente de liberar la memoria.

Su funcionamiento se podría entender como sigue: Java dispone de una memoria inicial para ejecutar un determinado código. A medida que se van creando objetos en nuestro código, Java los va alojando en la porción de memoria que le ha sido asignada y definiendo las referencias a los mismos. Si en algún momento Java considera que se va a quedar sin memoria para ejecutar el código, comprueba todos los objetos que han sido alojados en la memoria y cuántas referencias siguen existiendo a cada uno de ellos. Si algún objeto en la memoria no tiene ninguna referencia apuntándolo, esto quiere decir que esa memoria (ese objeto) ya no está siendo utilizado, y por lo tanto la memoria ocupada puede ser liberada. Entonces el recolector de basura libera esas zonas de memoria y, si es posible, reordena los objetos que quedan en la memoria para optimizar el espacio libre disponible.

Veamos un ejemplo muy sencillo de lo mismo:

```
01 public static void main (String [] args){
02
03          System.out.println ("La cantidad de memoria libre es "
                                + Runtime.getRuntime().freeMemory());

04          //Construcción de objetos
05          Circunferencia circ_radio_0 = new Circunferencia();
06          Circunferencia circ_radio_3 = new Circunferencia (3.0);

07          System.out.println ("La cantidad de memoria libre es "
                                + Runtime.getRuntime().freeMemory());

08          circ_radio_0 = new Circunferencia (5.0);
09          System.gc();
10          System.out.println ("La cantidad de memoria libre es "
```

```
                                    + Runtime.getRuntime().freeMemory());
11      }
```

El resultado de ejecutar el código anterior (la cantidad de memoria disponible depende de la máquina en la cual lo ejecutemos), sería:

```
//Cantidad de memoria libre antes de crear los dos objetos:
“La cantidad de memoria libre es 4973120”

//Cantidad de memoria libre tras construir los dos objetos:
“La cantidad de memoria libre es 4954360”

//Cantidad de memoria libre después de modificar la referencia en
//“circ_radio_0”:
“La cantidad de memoria libre es 5043512”

//Se puede observar como el objeto creado en la línea “05”
//ya no es apuntado por ninguna referencia,
//y el recolector de basura decide liberarlo
//Además libera otras referencias que había en el programa
```

El “garbage collector” de Java está implementado como una rutina interna que la propia JVM decide cuándo debe ser ejecutado, así que su comportamiento es difícil de predecir. En el fragmento de código anterior nosotros hemos decidido invocarlo en la línea “09” para “forzarlo” a liberar las referencias no utilizadas. Generalmente dejaremos que sea la propia JVM la que lo invoque cuando lo considere necesario. Como conclusión, podemos enunciar que el “garbage collector” libera al programador de la ardua tarea de tener que liberar la toda la memoria dinámica alojada por sus programas.

1.12.4 ENTRADA Y SALIDA POR CONSOLA EN JAVA

La entrada y salida de caracteres por teclado en Java también presenta ciertas diferencias con respecto a la de C++. Veamos en primer lugar cómo se puede hacer la salida de caracteres por la consola de MS-DOS recuperando uno de los ejemplos que introdujimos en las Subsecciones anteriores:

```
public class Principal {

        public static void main (String [] args){
        //Construcción de objetos
        Circunferencia circ_radio_0 = new Circunferencia ();
        Circunferencia circ_radio_3 = new Circunferencia (3.0);

        //Uso de los objetos
        System.out.println ("El objeto circ_radio_0 tiene como radio "
                + circ_radio_0.getRadio());
        System.out.println ("El objeto circ_radio_3 tiene como radio "
                + circ_radio_3.getRadio());
```

```
        }
}
```

El comando que nos permite mostrar por pantalla una cadena de caracteres (un objeto de tipo “String” en Java) es:

```
System.out.println ("El objeto circ_radio_0 tiene como radio "
                                    + circ_radio_0.getRadio());
```

Veamos de forma más detallada lo que significa:

En primer lugar, podemos encontrar una descripción detallada de la clase System en http://java.sun.com/j2se/1.5.0/docs/api/ que es la página web en la que Sun ofrece información sobre la especificación de la API (Application Programming Interface) de Java (en este caso, de la versión 1.5).

Más concretamente, la clase System está descrita en la página http://java.sun.com/j2se/1.5.0/docs/api/java/lang/System.html. Si buscamos en esta página la descripción de la clase, nos encontramos con que “System” es la clase que provee de facilidades para la entrada y salida estándar de caracteres (en el caso de una máquina Windows, la entrada y salida estándar es la consola de MS-DOS).

Si seguimos explorando la página anterior, encontramos que “out” es un atributo de la case “System” (de ahí la notación “System.out. ...”) que pertenece a la clase “PrintStream”. Por tanto, para conocer el comportamiento del método “pintln(String)” del objeto “out”, debemos ahora movernos a la clase “PrintStream” dentro de la especificación de la API de Java (http://java.sun.com/j2se/1.5.0/docs/api/java/io/PrintStream.html). La página ofrece cierta información sobre el propósito de la clase, y en particular su lista de métodos. Veamos ahora cuál es el comportamiento del método “println(String)”. Como se puede observar, el método “println ()” está *sobrecargado*, ya que existen 10 declaraciones del mismo que sólo varían en el tipo de los parámetros (boolean, char, char [], double, float, ...) y en particular, el que nosotros buscábamos, que era “String”.

El comportamiento del mismo era el que se podía esperar, es decir, el método “imprime” en la consola de MS-DOS el objeto de tipo “String” recibido como parámetro, y después un salto de línea.

Una última puntualización que hemos omitido sobre el comportamiento del método anterior. El parámetro que tenía el método “System.out.println (...)” en nuestro caso era

```
"El objeto circ_radio_0 tiene como radio " + circ_radio_0.getRadio()
```

que hemos dicho que era de tipo “String”.

En Java, las comillas “”...”” sirven para definir cadenas de caracteres. El operador “+” está sobrecargado, y cuando lo aplicamos a una cadena y un

objeto de cualquier otro tipo (o una variable de un tipo básico), produce una nueva cadena, que en nuestro caso, es la que ha sido impresa por pantalla.

(Puedes encontrar más detalles sobre el trabajo con cadenas de caracteres en Java en http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html)

El anterior ejemplo nos ha servido para dos propósitos. En primer lugar, para detallar el proceso de escritura por consola en MS-DOS. En segundo, para aprender a utilizar la especificación de la API de Java. El uso de la misma es imprescindible cuando pretendemos construir aplicaciones de dimensiones considerables en Java.

Veamos ahora como se puede realizar la lectura desde la consola de MS-DOS en Java. Lo "natural" sería pensar que exista un método "System.in.readln(String)" que nos permitiera hacerlo, pero éste no es el caso (puedes comprobarlo tú mismo en la especificación de la clase InputStream en http://java.sun.com/j2se/1.5.0/docs/api/java/io/InputStream.html). Lo más que podemos encontrar es un método que lee bytes, que luego deberían ser convertidos al tipo de objeto que pretendamos leer.

Hay una variedad de métodos y clases en Java que permiten facilitar la lectura diferentes del anterior. En este curso presentaremos una opción que está disponible desde la versión 1.5 de Java, que consiste en usar la clase Scanner (http://java.sun.com/j2se/1.5.0/docs/api/java/util/Scanner.html). Nosotros aquí no haremos una descripción detallada de todos y cada uno de los métodos de la misma, sino que nos centraremos en los métodos que nos resultarán de utilidad a lo largo del curso.

Para explicar los mismos haremos uso de un sencillo ejemplo.

```
import java.util.Scanner;

public class Principal {

        public static void main (String [] args){

                //Construcción de objetos
                Circunferencia circ_radio_0 = new Circunferencia();
                Circunferencia circ_radio_3 = new Circunferencia (3.0);

                //Declara una instancia de nombre "entrada" de la clase Scanner
                Scanner entrada = new Scanner(System.in);

                System.out.println ("Escribe un nuevo radio para circ_radio_0");

                //Declara una variable de tipo double y
                //la inicializamos con el valor leído por teclado
                double aux = entrada.nextDouble ();

                //Lo guarda en el objeto circ_radio_0
```

```
                circ_radio_0.setRadio (aux);

                System.out.println ("El objeto circ_radio_0 tiene como radio "
                        + circ_radio_0.getRadio());

                System.out.println ("Escribe un nuevo area para el circ_radio_3");

                //Utiliza la propia variable aux para leer un
                //nuevo area para el objeto circ_radio_3
                aux = entrada.nextDouble ();

                circ_radio_3.setArea (aux);

                System.out.println ("El objeto circ_radio_3 tiene como radio "
                        + circ_radio_3.getRadio());

        }

}
```

Algunos comentarios sobre el fragmento de código anterior:

1. En primer lugar, el uso de la librería java.util.Scanner que hemos “importado” en la primera línea de nuestro código (la especificación de la clase la tienes en http://java.sun.com/j2se/1.5.0/docs/api/java/util/Scanner.html).

2. En segundo lugar, por medio del comando

```
Scanner entrada = new Scanner (System.in);
```

lo que conseguimos es declarar un objeto de nombre “entrada” de la clase “Scanner”, y construirlo con un parámetro que en nuestro caso será la consola de MS-DOS, que está representada por el objeto “System.in” (de un modo más o menos similar se podría construir con parámetro inicial un fichero del cual queramos leer datos).

3. A partir de aquí, sobre el objeto “entrada” podemos hacer uso de los objetos propios de la case Scanner (de nuevo puedes repasar la lista en http://java.sun.com/j2se/1.5.0/docs/api/java/util/Scanner.html)

Nosotros pretendemos leer datos de tipo “double” de la consola de MS-DOS, luego utilizaremos el método “nextDouble()” de la clase. Su uso es:

```
double aux = entrada.nextDouble ();
```

Aquí debemos señalar una particularidad del mismo. El método “nextDouble()” lee valores de tipo double de la forma “parte_entera , parte_decimal”, en lugar de lo habitual “parte_entera . parte_decimal”. Sin embargo, el método “System.out.println (double)” mostrará los valores de tipo double en la forma

“parte_entera . parte_decimal”. Debes tener lo anterior en cuenta a la hora de leer y mostrar números reales por la consola de MS-DOS.

4. Aparte del método hasta ahora refererido, “nextDouble()”, la clase “Scanner” cuenta con otros métodos: “next()”, que devuelve el siguiente objeto de tipo “String” que encuentre en el objeto asociado al Scanner (en nuestro caso, la consola de MS-DOS), “nextBoolean()” que nos devuelve el booleano correspondiente al siguiente elemento en el objeto asociado al Scanner, “nextInt()”, ... que nos permiten leer el siguiente objeto de la entrada escogida y convertirlo al tipo elegido. Iremos trabajando con todos ellos a lo largo del curso.

Hay otras formas de conseguir programar la entrada de caracteres de la consola en Java. La más extendida (hasta Java 1.5) era asociar un objeto de tipo “BufferedReader” al “InputStreamReader”, pero exigía el tratamiento de excepciones (que no veremos hasta el Tema 5) haciendo el código más farragoso. Otra forma, disponible a partir de Java 1.6, es utilizar la clase “Console” (http://java.sun.com/javase/6/docs/api/java/io/Console.html) que unifica la entrada y salida desde la salida de la consola de MS-DOS.

1.12.5 ALGUNAS REFERENCIAS ADICIONALES EN JAVA

Hay múltiples referencias sobre Java que te permitirán ampliar tu conocimiento sobre el lenguaje.

La primera y más importante es http://java.sun.com donde se puede encontrar el compilador de Java, la JDK, múltiples versiones de Java, foros de expertos, manuales de uso del lenguaje, la API que hemos utilizado en la Sección anterior,...

También hay múltiples libros que puedes consultar al respecto. Una de las posibles referencias sería la obra “Piensa en Java”, de Bruce Eckel (que está también disponible en inglés en Internet en http://www.mindviewinc.com). También puede resultar interesante “Java in a nutshell” de David Flanagan.

APÉNDICE 1. COMPARACIÓN ENTRE LA IMPLEMENTACIÓN DEL TIPO ABSTRACTO DE DATO “PILA” POR MEDIO DE REGISTROS Y POR MEDIO DE CLASES

En este apéndice pretendemos mostrar las diferencias de uso entre la representación de un Tipo Abstracto de Datos (TAD) por medio del uso de registros, que era el conocido antes de empezar este curso, y por medio del uso de clases, como hemos aprendido en el presente Tema. El objetivo es ilustrar las diferencias entre ambas aproximaciones y mostrar las ventajas del uso de clases y las diversas características de las mismas que hemos presentado en este Tema.

Ambas implementaciones del TAD serán presentadas en C++ para facilitar la comparación, y representarán una pila de forma estática.

Veamos en primer lugar el aspecto que tiene el fichero “Pila.h” de una posible implementación de una Pila haciendo uso de registros:

```
//Vamos a trabajar con pilas de enteros
typedef int telemento;

//número máximo de datos que habrá en la pila
const int MAX= 1000;

struct pila
{
        int num;//número de datos que hay en la pila
        telemento  datos [MAX];
};

//Inicia P como una pila vacía
void iniciarPila(pila & P);

//Apila en la pila P el elemento d
void apilar(pila & P,telemento d);

//Si la pila P está vacía devuelve el valor verdad
//En caso contrario devuelve el valor falso
bool pilaVacia(pila P);

//Devuelve el elemento más reciente en la pila P y no modifica la pila
telemento cima(pila P);

//Modifica la pila P eliminando el último elemento apilado
void desapilar(pila & P);
```

Pasemos ahora a compararlo con el aspecto que tendría una representación similar realizada por medio de clases. Veamos el fichero “Clase_Pila.h” correspondiente:

```
#ifndef PILA_H
#define PILA_H 1

typedef int telemento;

class Pila {

    //Constantes de clase
    private:
    //número máximo de datos que habrá en la pila
    const static int MAX= 1000;

    //Atributos de clase
    private:
    int num;
    telemento datos [MAX];
```

```
    //Constructores de clase
    public:
    Pila ();

    //Métodos propios de la clase
    public:
    void apilar(telemento d);
    bool pilaVacia();
    telemento cima();
    void desapilar();
};

#endif
```

Señalamos ahora alguna de las diferencias que se observan entre los ficheros de cabeceras:

1. En primer lugar, la **encapsulación de la información** o **modularidad** que hemos destacado en el uso de las clases. Si bien en la representación por medio de registros, dentro del registro sólo hemos incluido los **atributos** de una pila, en la representación por clases hemos incluido los mismos **atributos y todos los métodos** que van a trabajar sobre pilas.

2. En segundo lugar, se puede observar como en las clases tenemos la posibilidad de representar las distintas partes de la misma como "**private**" o como "**public**", dando lugar así a lo que hemos llamado **ocultación de la información**. De este modo conseguimos que los atributos no puedan ser accedidos desde fuera de la clase, aumentando la seguridad de los mismos. Sólo el programador de la clase puede acceder a los campos declarados "**private**". ¿Conoces alguna forma de simular este comportamiento con los registros?

3. En tercer lugar, si observas las cabeceras de los métodos en el fichero "Pila.h" y en "Clase_Pila.h", observarás que en la representación por medio de registros el parámetro de tipo "Pila" aparece de forma explícita en todas ellas. Sin embargo, en la POO (y en nuestro caso en el fichero "Clase_Pila.h"), como **los métodos pertenecen a una clase,** cada vez que invoquemos a un método, esta invocación será sobre un determinado objeto (referenciado por medio de **this**), en este caso, la pila que deseamos iniciar, desapilar, ...

4. La existencia de constructores. En la representación por medio de registros, hemos decidido declarar un método ""iniciarPila (Pila &)" que inicie una pila vacía. Sin embargo, podíamos haber elegido no hacerlo, ha sido una decisión de diseño. En la POO, toda clase debe tener al menos un constructor (con las salvedades que veremos en el Tema 4), lo que nos obliga a construir un objeto (es decir, a reservar memoria para el mismo y a darle un valor inicial), antes de usarlo

Veamos ahora la definición de los anteriores ficheros de cabeceras. En primer lugar, el fichero “Pila.cpp”, que define las funciones sobre pilas representadas con registros:

```
#include "pila.h"

   void iniciarPila(pila & P) {
   //Inicia P como una pila vacía de datos
        P.num=0;
   }

   void apilar(pila & P ,telemento d){
   //Apila en la pila P el elemento d
        if (P.num<MAX-1)
        {

                P.datos[P.num]=d;
                P.num=P.num+1;
        }
   }

   bool pilaVacia(pila P) {
   //Si la pila P está vacía devuelve el valor verdad
   //En caso contrario devuelve el valor falso
        return (P.num==0);
   }

   telemento cima(pila P) {
   //Devuelve el elemento más reciente en la pila P y no modifica la pila}
        return(P.datos[P.num-1]);
   }

   void desapilar(pila & P){
   //Modifica la pila P eliminando el último elemento apilado
        P.num=P.num-1;
   }
```

Y veamos ahora cómo se definirían los métodos de la clase “Clase_Pila” declarada en el fichero “Clase_Pila.h”:

```
#include "Clase_Pila.h"

Pila::Pila(){
        this->num=0;
}

void Pila::apilar(telemento d){
        if (this->num<MAX-1)
        {
```

```
                this->datos[this->num]=d;
                this->num=this->num + 1;
        }
}

bool Pila::pilaVacia(){
        return (this->num==0);
}

telemento Pila::cima(){
        return(this->datos[this->num-1]);
}

void Pila::desapilar(){
        this->num=this->num-1;
}
```

De nuevo podemos señalar aquí tres diferencias:

1. De nuevo, incidir en la **modularidad** que ya hemos mencionado. Todos los métodos declarados en el fichero “Clase_Pila.h” como parte de la clase Pila aparecen ahora en su definición con el identificador

```
Pila::pilaVacia (){ ... }
```

2. En segundo lugar, observamos la aparición de “this” en la definición de los métodos de la clase. Lo que en el uso de registros se conseguía por medio del paso de **parámetros explícitos** (como “pila P”) ahora se consigue por medio del **parámetro implícito** que tienen todos los métodos (no estáticos) de una clase cuando son invocados sobre un objeto (y que conocemos por “this”). Como “pila P” era un registro y, sin embargo “this” es una referencia (o puntero) al objeto sobre el que se invoca un método, de ahí la diferencia adicional de usar “P.datos [...]” o usar “this->datos [...]” y “P.num = P.num + 1” o “this->num = this->num + 1”

Finalmente, un breve programa principal para ambas representaciones puede ayudar a ilustrar mejor algunas de las diferencias presentadas:

El siguiente programa principal (sin usar POO) declara una variable de tipo “pila”, la inicia vacía, y luego apila en la misma los primeros 35 números pares, para después mostrarlos:

```
#include <iostream>
#include <cstdlib>
#include "Pila.h"

using namespace std;

int main (){
```

```
    pila mi_pila;
    iniciarPila (mi_pila);

    for (int i = 0; i < 35; i++){
    apilar (mi_pila, 2 * i);
    }

    for (int i = 0; i < 35; i++){
    cout << "El elemento en la cima es " << cima (mi_pila) << endl;
    desapilar (mi_pila);
    }



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

El mismo programa principal haciendo uso de clases y de características propias de la POO:

```
#include <iostream>
#include <cstdlib>
#include "Clase_Pila.h"

using namespace std;

int main (){

    Pila mi_pila;

    for (int i = 0; i < 35; i++){
    mi_pila.apilar (2 * i);
    }

    for (int i = 0; i < 35; i++){
    cout << "El elemento en la cima es " << mi_pila.cima() << endl;
    mi_pila.desapilar();
    }

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

Pasamos a señalar algunas diferencias entre ambos:

1. En primer lugar, la definición de una pila vacía. Haciendo uso de registros, hemos tenido que hacer:

```
        pila mi_pila;
```

```
iniciarPila (mi_pila);
```

para conseguir una pila vacía ¿Cuáles hubiesen sido las consecuencias de no hacer uso de “iniciarPila (mi_pila)”? Sin embargo, haciendo uso de clases, en un único comando

```
Pila mi_pila;
```

hemos conseguido la declaración de un objeto Pila, y la construcción del mismo por medio del constructor sin parámetros de la clase Pila. Eso aumenta la fiabilidad en el uso de la POO con respecto al uso de registros, ya que nos asegura que cuando utilicemos cualquiera de los métodos de un objeto éste habrá sido inicializado por medio de algún constructor.

2. En segundo lugar, de nuevo la **encapsulación de la información**, ya que las invocaciones a funciones que en la versión con registros hacíamos por medio de

```
apilar (mi_pila, 2 * i);
```

y en las cuales el parámetro “pila” era pasado como un parámetro más, pasan a ser ahora, haciendo uso de POO:

```
mi_pila.apilar (2 * i);
```

Es decir, el objeto sobre el que invocamos a un método pasa a ser un “parámetro implícito” que no hace falta situar en la lista de parámetros. De nuevo, se puede observar que los métodos “pertenecen” al objeto (haciendo explícita la modularidad de la clase).

APÉNDICE 2. DIFERENCIAS ENTRE C++ Y JAVA EN LA DECLARACIÓN Y DEFINICIÓN DE CLASES Y EN LA CONSTRUCCIÓN DE OBJETOS

Uno de los objetivos del curso es aprender a codificar los distintos conceptos vistos en el mismo sobre POO tanto en C++ como en Java. Por eso, es importante ser capaz de distinguir las diferencias que van surgiendo entre ambos lenguajes (tanto de sintaxis como a nivel conceptual) a medida que introducimos nuevos elementos de los mismos.

La lista que presentamos aquí no pretende ser exhaustiva, sino que se centrará en las nociones que hemos presentado en este Tema. A partir de los ejemplos que hemos introducido en este Tema y de los desarrollados en las prácticas 1 y 2 el propio alumno debería ser capaz de percibir bastantes de estas diferencias. Aquí presentamos algunas de ellas.

Principales diferencias a nivel conceptual:

1. En Java podemos aplicar modificadores de acceso (public o package) a una clase, mientras que en C++ sólo son aplicables a atributos y métodos

2. En Java todos los objetos se implementan de forma interna por medio de referencias a los mismos, de tal modo que dichas referencias son transparentes al usuario. De ahí, por ejemplo:

```
Punto mi_punto = new Punto (3.5, 4.6);
mi_punto.setX (4.6);
```

En el ejemplo anterior se puede observar cómo hemos reservado memoria de forma dinámica para el objeto “mi_punto”. Además, el compilador “vigila”, y nos avisa, para que todas las referencias estén inicializadas, aumentando la seguridad de nuestro código.

En C++, sin embargo, el usuario puede elegir entre trabajar con objetos o con punteros a lo mismos. Tanto trabajar en C++ con punteros como hacerlo con objetos tiene sus inconvenientes. Una desventaja de trabajar con objetos en C++ es que la memoria no puede ser posteriormente liberada (no aceptan el comando C++ “delete”). Una desventaja de trabajar con punteros es que nuestros programas no nos advertirán de que no hemos reservado memoria–inicializado el objeto correspondiente.

Por ejemplo:

```
Circunferencia * mi_circunferencia;
mi_circunferencia->getRadio();
```

El compilador no nos advertiría de ningún fallo en el código anterior, y el objeto “mi_circunferencia” estaría sin inicializar.

3. Una consecuencia del punto anterior es también el uso de la referencia (o puntero) “this” tanto en Java como en C++. En C++ “this” es una referencia al objeto sobre el cual ha sido invocado un método, y de ahí su particular notación “this->atributo” o “this->metodo()”. Esta misma referencia en Java es desreferenciada cuando se utiliza, dando lugar a la sintaxis “this.atributo” o “this,metodo()”.

4. Otra de las diferencias importantes entre ambos lenguajes, que hemos dejado entrever en el punto 2 anterior, es la existencia en Java de un “garbage collector” (o recolector de basura) que se encarga de liberar la memoria que no está siendo utilizada en un programa. Esto hace que el programador no deba prestar atención a eliminar las referencias no utilizadas de un programa. Cuando una región de memoria deja de ser utilizada, el “garbage collector” la liberará y dejará disponible para su futuro uso (no de forma instantánea, sino cuando el compilador invoque al “garbage collector”). En C++, sólo la memoria que ha sido reservada de forma dinámica puede ser liberada, y esto se hace por medio del comando “delete”.

5. Con respecto a la construcción de objetos, de nuevo Java nos ofrece más seguridad en nuestro código que C++.

En Java, la declaración y la construcción de un objeto pueden hacerse de forma separada. Por ejemplo:

Pelicula La_vida_de_Brian;

La_vida_de_Brian = new Pelicula (“La vida de Brian”, “Terry Jones”, 94);

En C++, sin embargo, existen dos posibilidades:

Si utilizamos objetos, la declaración y la construcción se deben hacer de forma simultánea:

Pelicula La_vida_de_Brian (“La vida de Brian”, “Terry Jones”, 94);

Si no, obtendrías un error de compilación. De ahí que cuando usas la sintaxis

Pelicula La_vida_de_Brian;

internamente se está realizando una llamada al constructor sin parámetros de la clase “Pelicula”.

Si utilizamos punteros, la situación cambia:

Pelicula * La_vida_de_Brian;

El anterior puntero puede ser utilizado sin que C++ nos advierta de ningún error.

6. En Java el método “main” debe formar parte de una clase. Eso responde a la máxima de la POO de que todos los elementos de un programa deben estar incluidos en alguna clase. El método “main”, en particular, no es un método propio de ningún objeto de la clase, sino que es un método de dicha clase, y de ahí que lo implementemos como un método de clase, es decir “static”.

Sin embargo, en C++, la función “main” no está dentro de ninguna clase.

Algunas diferencias a nivel de sintaxis:

1. En C++ separaremos la declaración y la definición de una clase en ficheros distintos (“*.h” y “*.cpp”), mientras que en Java ambas partes se realizarán de forma conjunta en un fichero “*.java”

2. En Java el tipo más usado para representar cadenas de caracteres es “String” (http://java.sun.com/j2se/1.4.2/docs/api/java/lang/String.html), mientras que en C++ se suele usar “char *“ o “char []” para el mismo fin.

3. La entrada/salida por teclado en Java y C++ es completamente distinta. En C++ disponíamos de la librería <iostream> que nos facilitaba los métodos “cin” y “cout” para tales funciones. En Java la salida por pantalla se puede hacer por

medio de “System.out.println (String)” y la lectura por pantalla se puede hacer, por ejemplo, por medio de la clase Scanner y sus métodos.