UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CONTADURÍA Y ADMINISTRACIÓN

Licenciatura En Informática

Informática V

(Programación Orientada a Objetos)

Autor: L.I. María de Lourdes Isabel Ponce Vásquez

SEPTIEMBRE 2020 – ENERO 2021

Contenido

Unidad 2. CLASES Y OBJETOS

Objetivos Específicos

- Aprender a definir clases y objetos
- Definir los conceptos de miembro, atributo, método, constructor, destructor y paquete
- Usar los modificadores de acceso *private* y *public* apropiadamente como base para encapsulamiento
- Manejar la sobrecarga de métodos
- Invocar un método de un objeto en particular a través de mensajes
- Especificar y emplear constructores de objetos
- Especificar y emplear destructores de objetos
- Comprender el uso de los paquetes de clases

2.1. Introducción

La característica principal de los lenguajes orientados a objetos es la posibilidad de manipular objetos a partir de clases que especifican las propiedades (atributos) y operaciones (métodos) comunes de un grupo de objetos. La manipulación consiste entre otras cosas de crear (mediante un constructor) y eliminar de memoria (mediante un destructor) los objetos. En la unidad anterior se proporcionaron sólo definiciones de estos elementos, en esta unidad se estudian con mayor detalle.

2.2. Definición de clase y objeto

El primer paso para escribir programación orientada a objetos es declarar las clases de objetos que son, como ya se había mencionado, tipos de datos abstractos, estos, son los bloques básicos de construcción. La declaración de la clase consiste en indicar los atributos y métodos dentro de su clase correspondiente.

Como sabemos, la abstracción es un concepto básico en la solución de problemas, ya que permite definir conceptos generales a partir de objetos particulares que representan algún aspecto del mundo real. Los TDA son independientes de la implementación, lo cual hace posible centrarse en los datos y operaciones sin pensar en el lenguaje de programación y posteriormente traducirlos al lenguaje deseado.

En los paradigmas de programación no orientados a objetos, el concepto de módulo y el de tipo permanecen separados. La propiedad más importante de la noción de clase es que incluye los dos conceptos; una clase es un módulo o unidad de descomposición de software y también es un tipo abstracto de datos.

En la manufactura, un diseño es una descripción de un dispositivo del cual muchos dispositivos físicos son construidos. En software, una clase es un diseño de software que se puede usar para *instanciar* muchos objetos individuales. Es una descripción de un objeto:

- Una clase describe los datos que incluye cada objeto (*atributos*).
- Una clase describe el comportamiento que cada objeto realiza (*métodos*).
- En conjunto son llamados *miembros*.

2.2.1. Definición de clases en Java

Los diagramas de clases de UML, se emplean durante la etapa de análisis para indicar las responsabilidades comunes de las entidades que intervienen en el comportamiento de un sistema. Durante la etapa de diseño, sirven para mostrar la estructura de las clases que forman la arquitectura del sistema y finalmente para transformarlas en código en el lenguaje seleccionado, son la representación de los TDA.

El siguiente diagrama muestra la clase MiFecha de acuerdo a un diseño previamente definido:

El diagrama es básicamente un rectángulo dividido en tres secciones, la superior indica el nombre de la clase (MiFecha); la de en medio, incluye la lista de atributos; y la inferior, lista las operaciones del objeto. Un modelo orientado a objetos de una aplicación consiste de una descripción de todas las clases y sus relaciones, los objetos y sus interacciones y la documentación completa de estos.

Los TDAs se traducen a clases, usando las facilidades de los LOO; por ejemplo, la sintaxis general en Java para implementar una clase es:

```
<declaración_de_clase> ::=
        [<modificador>] class <nombre> {
                <declaración_de_atributo>*
                <declaración_de_constructor>*
                <declaración_de_método>*
        }
```

Así, la clase MiFecha quedaría en Java:

```
public class MiFecha {

        private int eDia;
        private int eMes;
        private int eAnio;

        public MiFecha(int dia, int mes, int anio) {
                setDia(dia);
                setMes(mes);
                setAnio(anio);
        }

        public void setDia(int dia) {
                eDia = dia;
        }

        public void setDia(int mes) {
                eMes = mes;
        }
```

```
        public void setDia(int anio) {
            eAnio = anio;
        }

        public int getDia() {
            return eDia;
        }

        public int getMes() {
            return eMes;
        }

        public int getAnio() {
            return eAnio;
        }
    }
```

La definición de una clase comienza con algún modificador, generalmente public y la palabra reservada class. El nombre elegido debe hacer referencia al concepto representado. Dentro del bloque de la clase se definirán los atributos, constructores y métodos correspondientes.

Los miembros (atributos y métodos) estáticos se asocian con la clase que los define y no con sus objetos. Eso implica que se comparten entre todas las instancias (objeto) de la clase.

La mayoría de los LOO proporcionan gran cantidad de clases estándar para procesar texto, crear interfaces, manejar gráficos, etc.

2.2.2. Definición de Objetos en Java

Los conceptos de clase y objeto están profundamente relacionados, ya que no se puede hablar de un objeto sin prestar atención a su clase. Sin embargo, existen diferencias entre ellos:

- Mientras un objeto es una entidad concreta que existe en el tiempo y el espacio, una clase representa sólo una abstracción (la "esencia" de un objeto). Así, se puede hablar de la clase Mamífero para representar las características comunes de ellos, para identificar un mamífero particular de esa clase, hay que hablar de uno específico como los leones, osos, etc.
- Una variable de tipo clase (**instancia de clase**), se llama objeto. Un objeto tiene datos (atributos) y comportamiento (métodos). Los objetos se crean durante la ejecución del programa.

Los objetos son el foco central y la base del PaOO. Una vez declarada una clase, se pueden crear cualquier cantidad de objetos de esa clase. Los objetos tienen la responsabilidad de realizar tareas específicas en colaboración con otros objetos para obtener la solución al problema.

Los objetos pueden ser físicos o tangibles, no tangibles, aunque visibles (una cuenta) o conceptuales, que no existen claramente, pero se usan para representar parte del problema, por ejemplo, el ambiente o temperamento de una persona que afecta las entidades del problema.

Los objetos son conceptos dinámicos porque los objetos muestran un comportamiento e interactúan entre ellos.

En la unidad anterior, se definió identidad como la *propiedad de un objeto que lo distingue de todos los demás*. En un programa, normalmente se trata de un identificador o nombre de variable que distingue cada objeto. El fracaso en reconocer la diferencia entre el nombre de un objeto y el objeto mismo es fuente de muchos errores en la POO.

- Dos objetos con identidades diferentes pueden tener valores idénticos en sus atributos.

- Inversamente, el valor de los campos de un cierto objeto puede cambiar durante la ejecución de un sistema; pero esto no afecta la identidad del objeto.
- A su vez, un objeto puede ser referenciado por varias identidades (alias).

La **compartición estructural** es la capacidad de que un objeto pueda nombrarse de varias maneras, es decir, existen alias para el objeto. La asignación de variables de referencia permite tener un solo objeto referenciado por n variables diferentes. La falta de cuidado al operar un objeto mediante su alias puede llevar a problemas como pérdida de memoria, violaciones de acceso a memoria y, peor aún, cambios de estado inesperados.

El tiempo de vida de un objeto se extiende desde el momento en que se crea por primera vez hasta que ese espacio se recupera. Para crear explícitamente un objeto, hay que declararlo y construirlo.

La declaración crea una variable estática (identidad) en la pila de la memoria, para hacer referencia al objeto en el momento de ejecución; la construcción reserva espacio para el objeto en el montículo (heap).

En Java, la sintaxis para declarar y crear un objeto es:

```
<declaración_de_objeto> ::
        <Nombre_de_clase> <variable_de_referencia> = new <Nombre_de_clase>([<argumento>*]);
```

También se puede declarar primero la variable de referencia y posponer o diferir la creación del objeto hasta el momento en que se requiera hacer uso del mismo:

```
<Nombre_de_clase> <variable_de_referencia>;
. . .
<variable_de_referencia> = new <Nombre_de_clase>([<argumento>*]);
```

Por ejemplo, en base a la clase MiFecha definida previamente, se pueden crear varios objetos de esa clase:

```
public class PruebaMiFecha {

        public static void main (String [] args) {

                MiFecha oHoy = new MiFecha(1, 1, 2019);
                System.out.println("La fecha es: " + oHoy.getDia() + "/" + oHoy.getMes() + "/"  +
                                                oHoy.getAnio());
        }
}
```

El código anterior, crea un objeto de la clase MiFecha referenciado por la variable oHoy, la llamada a **new Xxx**(), en este caso new MiFecha(), obtiene un espacio de memoria para el nuevo objeto. Como se crea con los valores 1, 1, 2019, la fecha mostrada será 1/1/2019.

Se debe tener cuidado y recordar que la declaración de una variable de un tipo (Clase), no da lugar a la creación de un objeto, la creación ocurre hasta que se ejecute **new** en este caso. En otros lenguajes, existen otras técnicas parecidas para la creación de objetos, pero en todos los casos, la declaración y creación del objeto son operaciones diferentes. Si se intenta realizar una operación sobre una referencia que no ha creado un objeto (referencia vacía o nula) se obtendrá una excepción.

2.2.2.1. Referencia a un objeto dentro de sí mismo

En la definición de los métodos existe un parámetro implícito que identifica al objeto mismo, éste suele llamase self, **this**, current o algo similar y hace referencia al objeto sobre el cual se hizo la llamada al método.

Esta referencia suele servir para:

- referirse a miembros de clase que se ocultan o son ambiguos con declaraciones locales,
- pasar el objeto actual (pasarse a sí mismo) como parámetro a otro método,
- para llamar a un constructor desde otro constructor de la misma clase; en este caso, this debe estar en la primera línea del constructor.

this se usa de manera similar al nombre de cualquier otro objeto con el operador punto. Los métodos de clase (static) no pueden usarse con this.

Por ejemplo, se puede modificar el constructor de la clase MiFecha como:

```
public class MiFecha {

        private int eDia;
        private int eMes;
        private int eAnio;

        public MiFecha (int dia, int mes, int anio) {

                this.eDia = eDia;
                this.eMes = eMes;
                this.eAnio = eAnio;

        }

        public MiFecha (MiFecha oFecha) {

                this.eDia = oFecha.eDia;
                this.eMes = oFecha.eMes;
                this.eAnio = oFecha.eAnio;
        }

        public MiFecha agregaDias(int eMasDias) {

                MiFecha oNuevaFecha = new MiFecha(this);

                oNuevaFecha.eDia = oNuevaFecha.eDia + eMasDias;
                // resto del código

                return oNuevaFecha;
        }
        //……métodos
}
```

En el primer caso, los nombres de los atributos y de los parámetros son iguales (eDia, eMes, eAnio), si se indicara eDia = eDia, no se podría definir cuál se refiere al atributo y cuál al parámetro, por eso, se emplea la referencia **this** para eliminar la ambigüedad e identificar a los atributos del objeto, mientras que los parámetros aparecen sin la referencia this. En el segundo, se emplea para referirse nuevamente a los miembros de la clase eliminando la ambigüedad con los nombres del objeto recibido. En el tercero, se pasa el objeto actual para crear otra fecha.

2.3. Atributos

Una vez que un programador empieza a pensar en objetos, todo lo ve como un objeto, sin embargo, no todo lo que encontramos en un sistema necesariamente es un objeto, pueden ser solo propiedades de un objeto.

En la unidad anterior se definió el estado de un objeto como *el conjunto de todas las propiedades distintivas del mismo y los valores de cada una de estas propiedades*. El estado, además, representa los resultados acumulados de su comportamiento. Las propiedades generalmente son estáticas, existen en todo momento mientras exista el objeto, mientras que los valores son dinámicos ya que pueden modificarse en algún momento.

Todos los atributos tienen un valor ya sea una cantidad o un objeto, así se pueden tener atributos que son:

- Variables de tipos primitivos (entero, real, carácter, lógico), que son intemporales, inmutables y no instanciadas.
- Variables de objeto (pertenecientes a una clase), que existen en el tiempo, son modificables, tienen estado, son instanciados y pueden crearse, destruirse y compartirse.

Por ejemplo, se puede tener dos objetos de tipo MiFecha: uno con la fecha 01/01/2019 y otro 31/01/2019, ambos objetos se crean a partir de la misma clase, por lo que ambos tienen como propiedades día, mes y año, sin embargo, el valor del atributo día es diferente en ambos casos.

Las buenas prácticas de ingeniería recomiendan siempre ocultar el estado de un objeto definiendo una interfaz para manipularlo, de modo que sólo ciertos clientes tengan acceso a modificarlos o accederlos. De este modo, se hace ocultación de información mediante el encapsulamiento de sus atributos. Para lograrlo, las clases deben tener a su disposición, para cada atributo, cinco niveles posibles para otorgar privilegios de acceso a sus clientes:

- Nivel 0 - **Sin Acceso**. Es la protección total, los clientes no tienen forma de acceder al atributo.
- Nivel 1 – **Sólo Lectura**. Se permite acceso al atributo, pero no se otorga ningún derecho a modificarlo.
- Nivel 2 – **Escritura Restringida**. Se permite que los clientes modifiquen el atributo a través de métodos específicos.
- Nivel 3 – **Escritura Protegida**. Se permite a los clientes asignar valores, pero sólo si éstos satisfacen ciertas restricciones. Algunos lenguajes como Java no permiten este nivel.
- Nivel 4 – **Sin restricción**. Elimina cualquier restricción.

Los atributos se definen como declaraciones de datos y suelen llamarse campos de datos o variables de instancia.

En Java, la sintaxis para declarar un atributo es:

```
<declaración_de_atributo> ::=
    [<modificador>] <tipo|Clase> <nombre_atributo>*[ = <valor_inicial>;
```

Ejemplo:

```
public class MiFecha {
        private int eDia, eMes, eAnio;
        …
}
```

Generalmente, los LOOs asignan valores por omisión a los atributos al momento de crear nuevos objetos, esto asegura que los atributos tendrán valores, aun cuando no se les hayan asignado.

En el ejemplo anterior de la clase MiFecha se definieron tres atributos privados eDia, eMes y eAnio.

2.4. Métodos

Ningún objeto existe aisladamente, por el contrario, interactúan y colaboran recibiendo mensajes. En la unidad anterior se definió el comportamiento de objeto como *forma en que actúa o reacciona un objeto en términos de cambio de estado, envío y recepción de mensajes.*

De modo que el comportamiento representa su actividad. Una operación es una acción que un objeto efectúa sobre otro con el fin de provocar una reacción, a esto se le llama **invocación de métodos** o llamada a función o pasar un mensaje a un objeto. En la mayoría de los LOO las operaciones que los clientes pueden realizar sobre un objeto suelen declararse como métodos.

Un **método** (método o función miembro o método de instancia) es un procedimiento o función asociado a un tipo de objeto y está disponible para el objeto y sus descendientes. No se pueden asociar métodos a otros tipos de datos. Es también la unidad modular más pequeña y realiza una sola subtarea dentro de una clase.

Los métodos como los constructores, se encuentran asociados con un determinado objeto con el que se invocan, a menos que sean métodos de clase (estáticos); por lo que para invocar a un método siempre se debe utilizar el operador punto asociando el objeto y el método. Las operaciones que un objeto puede realizar corresponden generalmente a alguno de los siguientes tipos:

- Modificador. Es una operación que altera el estado de un objeto. Suelen llamarse setters, e inician con la palabra set<atributo>.
- Selector. Es una operación que accede al estado de un objeto, pero no altera su estado. Suelen llamarse getters, e inician con la palabra get<atributo|objeto>.
- Iterador. Es la operación que permite acceder a todas las partes de un objeto en algún orden perfectamente establecido.
- Constructor. Es una operación que crea un objeto y/o inicializa su estado.
- Destructor. Es una operación que libera el estado de un objeto y/o destruye el propio objeto.

En la práctica de programación, es común encontrar los métodos de los mismos tipos juntos y separados de otros tipos.

Al conjunto de operaciones (métodos) que un objeto puede realizar sobre otro, se conoce como **protocolo**, y define el comportamiento admisible en un objeto

La estructura general de un método es:

- Encabezado, que describe la interfaz de la invocación y siempre tiene un tipo devuelto. El nombre y parámetros conforman la signatura del método, que debe ser diferente para cada método.
- Declaración de datos locales.
- Secuencia de instrucciones, que describe las acciones.

En Java, la sintaxis para declarar un método es:

```
<declaración_de_método> ::=
    [<modificador>] <nombre_método> (<parámetro>*) {
            [<variable_local>*]
            <instrucción>*
    }
```

Ejemplo:

```
public class MiFecha {
    …
    public void setDia(int eDia) {
            this.eDia = eDia;
    }
}
```

Los datos declarados dentro de un método sólo los conoce el método, el alcance de los datos es **local** al método. Los datos locales solo existen durante la ejecución del método; su persistencia se limita al tiempo de vida del método.

Cada método tiene un nombre, que debe ser significativo y mostrar el nombre de la subtarea que realiza. El nombre se usa para invocarlo desde otros objetos.

En general, nunca se realizan instrucciones de asignación y selección de atributos, del modo:

objeto.atributo = x; ó System.out.println(objeto.atributo);

ya que para acceder o modificar atributos siempre se utilizarán métodos que realicen estas operaciones.

2.4.1. El método main (principal)

El método **main** es un método especial; la ejecución de la aplicación inicia y termina en este método. En un programa típico (o aplicación), el método main lleva a cabo la siguiente secuencia general de tareas:

- Declara constantes, variables y objetos locales, tantos como requiera.
- Crea uno o más objetos de clases definidas previamente.
- Invoca uno o más métodos de los objetos creados. Delegando tareas y subtareas a los objetos para llevar a cabo la solución completa de la aplicación.

El método main debe especificarse en dentro de una clase que será la clase raíz o el punto de partida de la ejecución de un sistema. Al ser estático, este método no requiere la creación de un objeto para poder ejecutarse. Generalmente el código contenido en este método es pequeño ya que normalmente sólo creará alguna instancia de alguna clase que mantiene el control general de la aplicación.

2.4.2. Sobrecarga de métodos.

La definición de dos o más métodos con el mismo nombre se conoce como sobrecarga. Esta facilidad de los LOO permite que cualquier función en una clase sea sobrecargada. En otras palabras, es una redefinición de una función con otro proceso con el mismo nombre, pero con diferente número y/o tipo de parámetros. Es un tipo de polimorfismo.

En algunas circunstancias, se puede requerir escribir varios métodos en la misma clase (o subclase) que hacen básicamente el mismo trabajo con diferentes argumentos.

Por ejemplo, un método simple que se requiere para mostrar un texto representando sus argumentos. Estos métodos pueden llamarse imprimir( ). Ahora supongamos que se necesita un método diferente de impresión para imprimir cada tipo de dato int, float y String. Esto es razonable, porque tipos de datos diferentes requieren formatos diferentes y probablemente varíe el manejo. Se pueden crear tres métodos, llamados imprimirEntero(), imprimirFlotante( ) e imprimirCadena( ) respectivamente, sin embargo, esto puede ser tedioso, al permitir reutilizar un nombre de método para más de un método, se facilita la escritura.

Esto funciona, sólo si existe algo en las circunstancias bajo las cuales la llamada es realizada de modo que se distingue el método que se requiere usar. En el caso de los tres métodos de impresión, esta distinción se basa en el número y tipo de argumentos.

Reutilizando el nombre de método, se pueden escribir:

```
public void imprimir(int i)
public void imprimir(float f)
public void imprimir( )
```

Cuando se escribe el código para llamar uno de estos métodos, el método apropiado se selecciona dependiendo del tipo del argumento o argumentos proporcionados. Existen dos reglas para métodos sobrecargados:

- La lista de argumentos de la instrucción de llamada debe diferir suficiente para evitar la ambigüedad del método apropiado a llamar. Las promociones (por ejemplo, float a double) pueden ser aplicadas; esto puede causar confusión bajo ciertas circunstancias.
- El tipo regresado por el método puede ser diferente, pero no es suficiente que el valor de retorno sea diferente. La lista de argumentos debe ser diferente.

A esta característica se suele llamar **polimorfismo en la sobrecarga**.

2.5. Mensajes

Los mensajes son una de las partes que definen el comportamiento de un objeto a la par con el estado del mismo. Los objetos se comunican e interactúan mediante el envío de mensajes, de modo que puedan colaborar en un objetivo común. El objeto que envía el mensaje es el solicitador de un servicio (cliente) que puede ser proporcionado por el objeto receptor o proveedor del servicio (servidor). Un objeto puede ser proveedor de algunos servicios y al mismo tiempo ser cliente de otros servicios que solicita a otros objetos.

El mensaje representa una solicitud de servicio, su propósito es el de solicitar que el servidor lleve a cabo una operación, en otras palabras, la solicitud es una llamada a una de las operaciones que puede realizar el servidor. Por tanto, se puede decir que: “Dada una clase *P*. Una clase *C* que contenga una declaración de la forma *a:P* (a es de tipo P, a puede ser atributo, parámetro o variable local), se dice que *C* es un cliente de *P*; y que *P* es un proveedor de *C*.

El solicitante no precisa conocer cómo el objeto proporciona el servicio pedido la implementación es interna al objeto y la gestiona el suministrador del objeto. El énfasis se produce en qué se puede obtener más bien que en cómo se obtiene.

Un programa orientado a objetos viene definido por la ecuación:

Objetos + Mensajes = Programa

Los objetos realizan operaciones en respuesta a los mensajes. Estas operaciones son específicas del objeto, un mensaje siempre se envía a un objeto en particular y se le conoce como invocación de métodos. Los mensajes contienen cuatro partes:

- La identidad del receptor, que es la referencia al objeto que proporcionará el servicio.
- La operación a invocar o iniciar, que es el servicio solicitado y debe ser una operación accesible.
- Los datos requeridos para ejecutar la operación, que son los argumentos.
- Los datos de salida, que es la respuesta al mensaje y es el resultado actual de la solicitud.

En Java, la sintaxis para invocar un método es:

<invocación_de_método> ::=
<nombre_de_objeto | nombre_de_clase> **.** <nombre_de_método>(<argumento>*);

Ejemplo:

```
oHoy.setDia(14);
System.out.println(“Hola mundo”);
```

Cuando un objeto envía un mensaje a otro objeto, el primero invoca o llama a un método del segundo. En la definición de la clase del segundo objeto, este método debe haber sido definido como público; de otro modo, no sería accesible a otros objetos.

La secuencia de actuación es la siguiente: el emisor envía el mensaje, el recepto recibe y ejecuta el método apropiado, finalmente, el receptor retorna una respuesta al emisor.

Para describir la interacción entre dos o más objetos, existe un diagrama de UML llamado diagrama de colaboración.

El paso de mensajes entre dos objetos es típicamente unidireccional, aunque ocasionalmente puede ser bidireccional. También es importante notar que, aunque el paso de mensajes es iniciado por el cliente y dirigido al servidor, los datos pueden fluir en ambas direcciones.

Como participante de una asociación de objetos, un objeto puede desempeñar uno de tres papeles:

- Actor. Es un objeto que puede operar sobre otros objetos, pero nunca se opera sobre él por parte de otros objetos, en algunos contextos, se le conoce como objeto activo o cliente.
- Servidor. Un objeto que nunca opera sobre otros objetos, solo otros operan sobre él.
- Agente. Un objeto que puede operar sobre otros objetos y además otros objetos pueden operar sobre él; un agente se crea normalmente para realizar algún trabajo en nombre de un actor u otro agente.

La forma de enviar mensajes, es decir, de utilizar los métodos es escribiendo el nombre del objeto, un punto y luego el nombre del método que se requiere y entre paréntesis los parámetros que requiera tal método. Por ejemplo, en la siguiente instrucción:

```
System.out.println("La fecha es: " + oHoy.getDia() + "/" + oHoy.getMes() + "/" + oHoy.getAnio());
```

Se envían tres mensajes al objeto oHoy: getDia(), getMes() y getAnio(), en este caso no se le indican parámetros, pero en caso de necesitarlos, se deben incluir. El objeto oHoy, realizará tres servicios u operaciones a la clase cliente en donde se escribió esta línea, obteniendo los valores de los atributos de día, mes y año respectivamente.

2.6. Encapsulación

Los objetos se representan en la mayoría de los lenguajes como estructuras de datos, en este sentido son similares a los registros, con la diferencia que el objeto encapsula datos y métodos y los registros son sólo datos.

La encapsulación sugiere que un objeto se describe como la *integración de atributos y comportamiento como una sola unidad*. La encapsulación se consigue con las clases.

La encapsulación es la metodología que permite esconder ciertos elementos de la implementación de una clase, pero proporcionando una interfaz pública para el software cliente, a esto se le llama **mecanismo de protección encapsulada**. Obliga a usar una interfaz para acceder a los datos y hace el código más fácil de mantener.

2.6.1. Interfaz de una clase

La **interfaz** de un módulo es la parte que es visible (pública) fuera de un módulo y permite interconectar diversos módulos. Esta interfaz debe ser limitada (pequeña), legible de modo que se pueda conocer fácilmente.

La interfaz puede contener constantes, variables, objetos y métodos. Cuando el acceso a algún atributo u operación no está permitido el modo de acceso se especifica como **privado** (private), de otro modo se define como **público** (public), en cuyo caso es accesible desde otros objetos.

Al describir los objetos de una clase, se deben mostrar sólo los servicios que el objeto proporciona y esconder los detalles de la implementación. De este modo, los objetos tienen dos vistas:

- La **vista externa** del objeto que se puede mostrar a los clientes del objeto. Esta vista consiste de la lista de servicios que otro objeto puede invocar. La lista de servicios puede ser usada como un contrato de servicio entre el objeto servidor y los objetos clientes.
- La **vista interna** representa los detalles de la implementación de los datos y operaciones del objeto. Esta información se esconde de otros objetos.

Para proteger las características de un objeto, se especifica un modo de acceso para cada característica.
La interfaz se puede dividir en cuatro partes:

- **Pública** (public). Una declaración accesible a todos los clientes.
- **Amiga** (friendly o de paquete). Una declaración accesible sólo a la propia clase, subclases y clases amigas (mismo paquete).
- **Protegida** (protected). Una declaración accesible sólo a la propia clase y subclases.
- **Privada** (private). Una declaración accesible sólo a la propia clase.

Ya que cada objeto corresponde a una clase, la descripción completa de los objetos se incluye en las definiciones correspondientes de las clases, particularmente, se emplea el símbolo + para indicar que un miembro es público, - para privados y # para protegidos.

2.7. Constructores

La mayoría de los LOO permiten que las clases definan una operación especial, llamada **inicializador**, cuya función es asignar valores iniciales a los atributos, de manera que aseguren que, al crearse, los atributos tendrán valores válidos. Estos inicializadores pueden tener diferentes nombres dependiendo del LOO, por ejemplo, en Java y C++ se les llama **constructores**.

Los constructores y destructores son rutinas que se asocian con un objeto particular. Sintácticamente estas rutinas son similares a los métodos generales ya que ejecutan una tarea relativa a un objeto. El constructor puede ser usado por los objetos hijos.

Los constructores son ejecutados automáticamente al declarar un objeto como instancia de una clase, por lo que por lo general son públicos, y su propósito es asignar espacio de memoria para el objeto y establecer su estado inicial. En Java, al crear un objeto, se asigna el espacio de memoria necesaria para almacenar ese objeto. Si no hay memoria suficiente, se obtendrá una excepción *OutOfMemoryError*.

En la mayoría de los casos, los constructores tienen el mismo nombre de la clase y no se hereda, por lo que cada subclase debe declarar siempre sus propios métodos; tampoco deben retornar algún valor, esto es lo que los hace particularmente diferentes a los métodos. En la realidad, todos los constructores retornan siempre un valor que es la referencia al objeto creado, pero como siempre se obtiene algo del mismo tipo, no es necesario expresarlo y por eso se omite en la definición de los constructores. Los parámetros pueden pasarse del mismo modo que a un método.

Los constructores no son métodos, no retornan valores, no se heredan y no son miembros de una clase.

2.7.1. Sobrecarga de constructores

Aunque cada clase tiene al menos un constructor, algunas clases pueden tener múltiples constructores (constructores sobrecargados), esto permite mayor flexibilidad al momento de crear objetos de esa clase. Existen tres tipos de constructores:

- **Constructor por omisión**. Es aquel que se proporciona automáticamente, sin necesidad de declarar alguno, no tiene parámetros y su cuerpo no contiene instrucciones. Cuando se crea el objeto inicializa los atributos con los valores por omisión. Permite crear objetos instanciados con *new Xxx( )* sin tener que escribir un constructor.
- **Constructor con parámetros**. Es el que tiene una lista de argumentos, a los cuales se les asigna algún valor al momento de crear el objeto. Estos valores sirven para dar valor a los atributos del objeto creado.
- **Constructor con valores por omisión**. Es el que asigna valores por omisión a los atributos en caso de que no se asignen parámetros.

La existencia, número y tipo de parámetros determina a qué tipo de constructor se llama. Es importante mencionar que los constructores por omisión no existen al momento de declarar algún otro tipo de constructor dentro de una clase.

Es una buena práctica de programación definir al menos un constructor con valores por omisión. La sobrecarga de constructores puede tener una desventaja, si existen dos constructores con el mismo tipo y número de parámetros, no se puede distinguir a cuál invocar, por ejemplo, si se desea crear un círculo con el valor de su radio o con el valor de su diámetro, y ambos valores son de tipo real, no se podría distinguir cuál es el constructor que se desea ejecutar.

En Java, para crear objetos con un constructor determinado, se emplea el operador **new** seguido del nombre de la clase que obviamente corresponde al nombre de un constructor y entre paréntesis se agregan los argumentos. La sintaxis en Java para declarar un constructor es:

```
<declaración_de_constructor> ::=
        <modificador> <nombre_clase> (<parámetro>*) {
                <instrucción>*
        }
```

Y para usarlo es:

```
<declaración_de_objeto> = new <Nombre_de_clase>([<argumento>*]);
```

Ejemplo:

```
public class Cosa {

        private int x;

        public Cosa ( ) {
                x = 47;
        }
        public Cosa(int nueva_x) {
                x = nueva_x;
        }
        public int getX( ) {
                return x;
        }
        public void setX (int new_x) {
                x = new_x;
        }
}
```

```
public class PruebaCosa {

        public static void main (String[ ] args) {

                Cosa cosa1 = new Cosa( );
                Cosa cosa2 = new Cosa(42);

                System.out.println("cosa1.x es " + cosa1.getX());
                System.out.println("cosa2.x es " + cosa2.getX());
        }
}
```

El operador new es el que reserva la memoria para el nuevo objeto. El nuevo objeto llega al constructor para configurarse y los parámetros indican los valores a asignarle.

El siguiente ejemplo muestra un ejemplo de sobrecarga de constructores:

```
public class Empleado {

        private static final double SALARIO_BASE = 15000.00;
        private String nombre;
        private double salario;
        private Fecha fechaNacimiento;

        public Empleado (String nombre, double salario, Fecha fdN) {
                this.nombre = nombre;
                this.salario = salario;
                this.fechaNacimiento = fdN;
        }

        public Empleado (String nombre, double salario) {
                this (nombre, salario, null);
        }

        public Empleado (String nombre, Fecha FdN) {
                this (nombre, SALARIO_BASE, FdN);
        }

        public Empleado (String nombre) {
                this (nombre, SALARIO_BASE);
        }
        // más código de empleado...
}
```

En este fragmento de código, se crearon cuatro constructores sobrecargados. El primero inicializa todas las variables de instancia. En el segundo, se omite la fecha de nacimiento. Nótese el uso de la referencia this: Es usada como una llamada a otro constructor (siempre dentro de la misma clase); en este caso el primer constructor. El tercer constructor llama al primero pasando a la clase la constante SALARIO_BASE. El cuarto constructor llama al segundo pasando el SALARIO_BASE el cual a su vez llama al primero pasando nulo para la fecha de nacimiento.

La referencia this en un constructor debe estar en la primera línea de código del constructor. Puede haber más instrucciones de inicialización después del this, pero no antes.

2.7.2. Construcción e Inicialización de objetos

A continuación, se describe brevemente como se crea e inicializa un objeto en Java:

La llamada a new Xxx () obtiene un espacio de memoria para el nuevo objeto realizando lo siguiente:

- Se obtiene un espacio de memoria para el objeto y las variables de instancia del objeto son inicializadas con sus valores por omisión (0, false, null, etc.).
- Después se realiza la inicialización explicita.
- Luego se ejecuta un constructor (dependiendo de los parámetros especificados).
- Por último, la variable recibe la referencia (de parte del constructor) al objeto en el heap de memoria.

Ejemplo, de acuerdo a la clase MiFecha declarada a continuación:

```
public class MiFecha {
    private int dia = 1;
    private int mes = 1;
    private int anio = 2019;

    public MiFecha (int dia, int mes, int anio) {
            this.dia = dia;
            this.mes = mes;
            this.anio = anio;
    }
    //resto del código
}
```

La siguiente instrucción crea un objeto de clase MiFecha referenciado por mi_cumple:

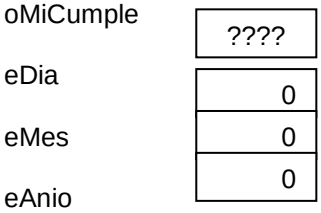
```
MiFecha mi_cumple = new MiFecha(7, 7, 2019)
```

La obtención de espacio en memoria y su disposición se realiza como se muestra a continuación:

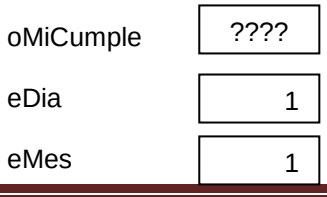
- Una declaración obtiene un espacio de memoria sólo para una referencia:

```
MiFecha oMiCumple = new MiFecha(7, 7, 2019)
```

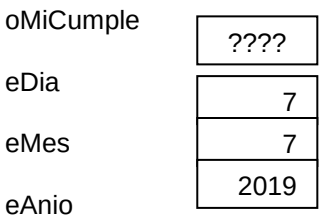
- Al usar el operador new se obtiene un espacio de memoria para MiFecha:



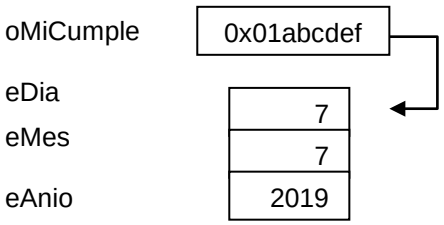
- Se realiza la inicialización explícita, tomando los valores de la declaración de los atributos en la clase:



- Se ejecuta el constructor:



- En caso de una sobrecarga de constructores, un constructor puede llamar a otro.
- Por último, se hace referencia al nuevo objeto mediante la variable de referencia creada inicialmente:



- Aunque este no es todo el proceso, permite ilustrar gran parte del mismo

2.8. Destructores

Cuando a un objeto se le asigna memoria en tiempo de compilación y permanece asignada durante la ejecución ese objeto se asigna estáticamente, pero cuando la memoria se asigna durante la ejecución del programa y sólo permanece asignada mientras se requiera entonces es un objeto dinámico.

La capacidad de crear objetos dinámicamente debe estar ligada a la capacidad de recuperar el espacio de memoria, de modo que se pueda volver a usar para otros objetos en posteriores instrucciones de creación.

La memoria es un recurso limitado que debe tratarse con cuidado para no saturarla y llegar a bloquear la aplicación y hasta el equipo de cómputo.

En la mayoría de los lenguajes, los objetos asignados dinámicamente deben contener métodos que destruyan el objeto o lo eliminen de memoria. Los destructores, realizan las tareas necesarias *para liberar el espacio de memoria usado por un objeto, ya que limpian y disponen de los objetos asignados dinámicamente*.

Los objetos locales (creados dentro de los métodos) se destruyen de forma automática al salir del bloque en el que se creó.

Los destructores se ejecutan también de modo automático al destruir los objetos, y dependiendo del lenguaje, generalmente tienen el mismo nombre de la clase sin argumentos.

Siempre que se destruye un objeto, ya sea implícita o explícitamente, se invoca automáticamente a su destructor, cuyo propósito es devolver el espacio asignado al objeto y sus partes, y llevar a cabo cualquier otra limpieza posterior a la existencia del objeto (como cerrar archivos o liberar recursos).

Los destructores se pueden heredar y se pueden definir múltiples de ellos.

En lenguajes como Smalltalk, CLOS y Java existe una herramienta llamada recolector de basura o de memoria dinámica que localiza los objetos que no tienen una referencia y recobra automáticamente la memoria. En C++ es responsabilidad del programador.

2.9. Paquetes

Los problemas generalmente son demasiado grandes para lidiar con ellos como una sola unidad, por lo que se dividen en pequeños problemas que son más fáciles de resolver. A la división de un problema en pequeñas partes se le conoce como descomposición modular y las pequeñas partes son llamadas módulos.

En la POO existen diferentes niveles de modularidad. Los módulos pueden ser paquetes, clases o métodos. Los **paquetes** son conjuntos de clases. Cada programa contiene una o más clases que definen una colección de objetos similares. Los paquetes y las clases son reutilizables, a diferencia de los métodos, ya que pertenecen a una clase en particular.

Los paquetes permiten tener acceso a un método dentro de una clase sin tener que unir todo el paquete a la aplicación que se está creando al indicar el nombre de paquete, nombre de clase y finalmente el nombre del método separándolos con punto. Por ejemplo:

```
Lang.System.out.println("Hola Mundo");
```

Invoca el método println () del objeto out, que es un atributo de la clase System, la cual está contenida en el paquete Lang. Lang es un paquete de Java que siempre se incorpora a las aplicaciones de modo automático, por lo que, generalmente, no se requiere indicar ni en cláusulas import al principio de la definición de una clase ni al momento de hacer referencia a alguna de sus clases.

Para poder hacer una invocación de este modo, generalmente se emplea la cláusula *import*, que describe al menos el paquete donde se encuentra el método deseado; pudiéndose incluir paquete, subpaquete y clase donde se encuentra o substituir el nombre de la clase por un asterisco que indica que serán todas las clases del paquete o subpaquete indicado. Ejemplo:

```
import javax.swing.*;
```

En este caso, se está indicando que se ponga a disposición del programa todos (*) los recursos de javax.swing.

```
import javax.swing.JOptionPane;
```

En este otro caso, se indica que sólo se requiere la clase JOptionPane.

Para indicar que una clase pertenece a un paquete se emplea la sentencia *package*, esta sentencia indica el nombre del paquete al que pertenece una clase y si se incluye debe ser la primera línea de código, antes de la declaración de la clase.

```
package Clientes;
```

Java proporciona una amplia biblioteca de clases organizadas en paquetes o en la terminología actual *Interfaz de Programación de Aplicaciones* (API), por ejemplo, para estructuras de datos, desarrollo de multimedia, criptografía, telefonía, reconocimiento de voz, etc. Las clases e interfaces del mismo paquete se almacenan en la misma carpeta.

Tarea:

Leer en otras fuentes sobre los temas vistos en esta unidad, hacer un resumen de la unidad (máximo 1 cuartilla), no olvidar conclusiones y bibliografía).

Considérese una oficina de renta de automóviles, un cliente renta un automóvil durante algunos días con un número finito de kilómetros. Identificar los objetos, atributos y métodos y crear un diagrama de clases del problema. Crear un programa que registre la renta de un auto y muestre los datos registrados.

Comparar y explicar las similitudes y diferencias entre:

1. Objeto y clase.
2. Encapsulamiento y ocultamiento de información.
3. Un objeto actor, servidor y agente.
4. Vista externa y vista interna del objeto.
5. Método y atributo.
6. Protocolo, interfaz y signatura.
7. Método y constructor.
8. Constructor por omisión y constructor con valores por omisión.
9. Las sentencias import y package.

Explicar:

10. Cuáles son y cómo se definen en Java los tres elementos principales de los objetos.