Clase Teórica 2 / 12

Índice de contenido

2.1 CLASES

El lenguaje JAVA permite implementar un diseño orientado a objetos, o sea, utiliza la Programación Orientada a Objetos para desarrollar programas.

En JAVA la unidad de programación es la clase a partir de la cual se instancian (crean) los objetos. Las clases en JAVA contienen atributos (datos o campos) y métodos (operaciones o funciones).

Las clases son para los objetos como los planos de construcción, para las casas. Así podemos construir muchas casas a partir de un mismo plano, o sea, podemos instanciar (crear) muchos objetos a partir de una clase. Una clase equivale a la generalización de un tipo especifico de objetos.

Al empaquetar el Software en forma de clases, permite a los programadores reutilizar esas clases en desarrollos posteriores. Los grupos de clases relacionadas forman paquetes que son componentes reutilizables. Reutilizar las clases existentes para la creación de nuevas clases y programas, ahorra tiempo y esfuerzo; también ayuda a los programadores a crear sistemas más confiables y efectivos ya que las clases existentes han pasado por un proceso extenso de pruebas, depuración y optimización.

La clase es la generalización de un tipo especifico de objetos, esto se puede decir como el conjunto de características (atributos) y comportamientos de todos los objetos que componen a la clase.

Por Ej. la clase Marcador tiene todas las características (tamaño, color, grosor del punto, etc) y todos los métodos o acciones (pintar, marcar, subrayar, etc), que pueden tener todos los marcadores existentes en la realidad.

Un Marcador en especial como un marcador permanente para discos compactos (CDs) de color negro y punto fino, es un objeto (o instancia) de la clase Marcador.

En JAVA, cuando empezamos a crear una clase, definimos métodos, así como en el plano de un automóvil se define el diseño del pedal del acelerador. Una clase contendrá uno o más métodos que se encargan de realizar sus tareas. Estos métodos son lo que en los lenguajes imperativos se llaman procedimientos o funciones.

Por Ej. una clase que representa una cuenta bancaria podría contener un método para depositar dinero en una cuenta, un segundo método para retirar dinero de una cuenta y otro método para solicitar el saldo.

Así como no podemos conducir un plano de automóvil y debemos construir un automóvil para poder conducirlo, lo mismo ocurre en programación; no podemos hacer funcionar una clase. Para ello debemos crear un objeto de esa clase e invocar al método para que realice las tareas que la clase describe cómo realizar.

Cuando apretamos el pedal del acelerador, le estamos enviando un mensaje al automóvil para que realice la tarea de aumentar la velocidad. De igual forma, en programación se envían mensajes a un objeto llamando a sus métodos para que realicen las tareas correspondientes.

Además de las tareas que puede realizar un automóvil, éste tiene algunos atributos como ser: su color, cantidad de puertas, cantidad de nafta en el tanque, cantidad de kilómetros recorridos, etc. Estos atributos también se representan en el plano de diseño. Cuando conducimos un automóvil estos atributos están asociados a él. De manera similar, las clases tienen atributos, y los objetos de esas clases tienen

esos atributos asociados.

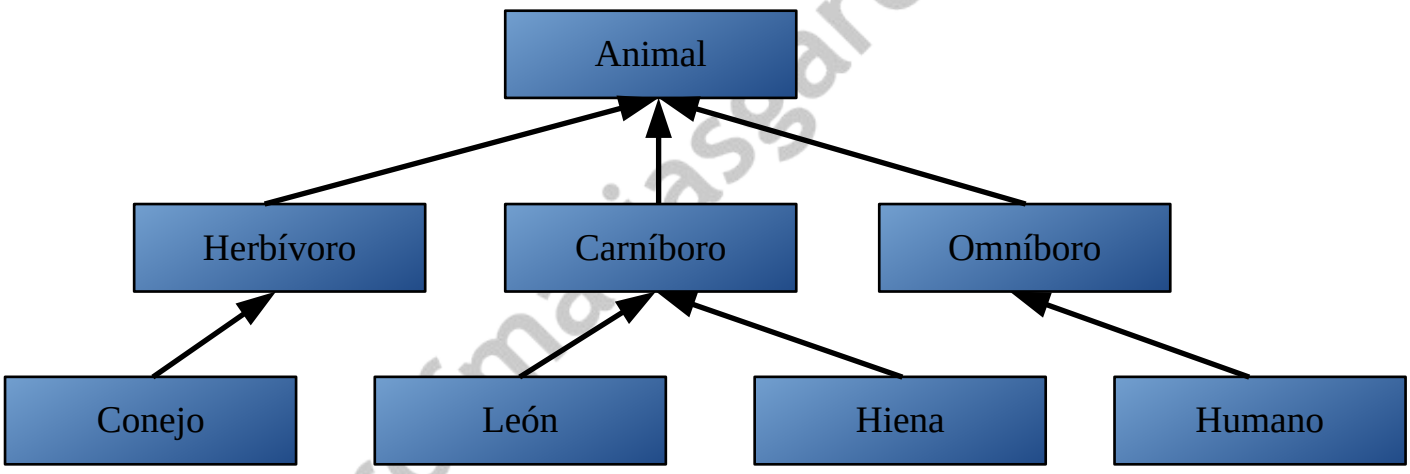
Por Ej. la clase cuenta bancaria tiene un atributo llamado saldo, que representa la cantidad de dinero en la cuenta. Cada objeto del tipo cuenta bancaria conoce el saldo en la cuenta que representa. Los atributos se especifican mediante las variables de instancia de la clase.

Por convención los nombres de las clases empiezan con mayúscula.

```
[Modificador] class NombreClase [extends NombreClasePadre] [implements interface]
{
    Declaración de variables;
    Declaración de métodos;
}
```

2.1.1 HERENCIA

En JAVA se maneja solo la herencia simple, por lo que cada clase solo tiene una clase padre, la cual se conoce como superclase y a la clase hija como subclase. Una subclase hereda las variables y métodos de su superclase, aunque puede llegarse a dar el caso de que se anulen o se agreguen algunos de estos elementos.



La herencia maneja una estructura jerárquica de clases como una estructura de árbol (Estructura de Datos), con lo que la POO todas las relaciones entre clase se ajustan a esta estructura. En esta estructura cada clase tiene una sola clase padre, la cual se conoce como superclase y la clase hija de una superclase se conoce como subclase.

Las distintas clases tienen distintas relaciones de herencia entre si: una clase puede derivarse de otra, en ese caso la clase derivada o clase hija hereda los métodos y variables de la clase de la que se deriva o clase padre. En JAVA todas las clases tienen como primer padre una misma clase: la clase Object.

Para implementar una subclase, se utiliza la palabra clave **extends** en la declaración de la clase. Su sintaxis es la siguiente:

```
[Modificador] class NombreDeSubclase extends NombreDeSuperclase {
  }
```

Una subclase hereda todos los métodos de su superclase a excepción de que la subclase sobrescriba los métodos. Del mismo modo que **this** apunta al objeto actual tenemos otra variable **super** que apunta

a la clase de la cual se deriva nuestra clase.

Se dice que una subclase sobrescribe un método de su superclase cuando crea un método con las mismas características de nombre, número y tipo de argumentos, que el método de la superclase. Y se emplea generalmente para agregar, quitar o modificar la funcionalidad del método que se hereda de la superclase.

```
class Animal {
    protected String nombre;

    void metodoUno(String var){
          // Cuerpo del metodoUno
    }

    void metodoDos( ){
          // Cuerpo del metodoDos
    }
}
public class Herbivoro extends Animal {
    private String tipoEstomago; // variable de instancia particular de Herbivoro

/* Este método sobrescribe al de la clase padre */
    void metodoUno (String var)   {
          // Cuerpo del metodoUno modificado en la subclase B
    }

    void metodoTres(int i, String dato){
          // Cuerpo del metodoDos de la clase Herbivoro
    }
}
```

Cuando en JAVA indicamos que una clase **extends** de otra clase estamos indicando que es una clase hija de esta y que, por lo tanto, hereda todos sus métodos y variables. Este es un poderoso mecanismo para la reusabilidad del código. Podemos heredar de una clase, por lo cual partimos de su estructura de variables y métodos, y luego añadir lo que necesitemos o modificar lo que no se adapte a nuestros requerimientos.

2.1.2 Tipos de Clases

Se tienen tres tipos de clase que son:

- Abstracta.- Es muy general (Ej. Animal).
- Común .- Es intermedia (Ej. Herbívoro).
- Final.- Es muy especifica (Ej. Conejo).

Clase abstracta.- Recordando el árbol jerárquico de las clases, la raíz de todas ellas se conoce como clase abstracta y es la clase que declara la existencia de métodos pero no la implementación de dichos métodos, esto es no lleva las llaves { } y las sentencias entre ellas.

En una clase abstracta por lo menos uno de los métodos debe ser declarado abstracto.

Se utiliza la palabra clave **abstract**, para declarar una clase o un método como abstractos.

```
abstract class A {
    abstract void metodoUno(String var1);

    void metodoDos( ){
          // Cuerpo del metodoDos
    }
}
```

De una clase abstracta no se puede crear objetos, pero si se puede heredar y las subclases, podrán agregar la funcionalidad a los métodos abstractos. Puede darse el caso de que si no lo hacen así, las subclases serán también abstractas.

Una clase común, es la parte del árbol jerárquico que se encuentra entre los niveles intermedios de dicho árbol. En esta clase se puede heredar y crear objetos. Se tiene ascendencia y descendencia.

En la clase final, no se puede heredar, se tiene ascendencia, pero no descendencia y es el nivel que se encuentra mas abajo del árbol jerárquico. Y se utiliza la palabra clave **`final`**.

```
final class Conejo {
    //cuerpo de la clase
}
```

2.1.3 INTERFACES

En JAVA no está soportada la herencia múltiple, esto es, no está permitido que una misma clase pueda heredar las propiedades de varias clases padres. En principio esto pudiera parecer una propiedad interesante que le daría una mayor potencia al lenguaje de programación, sin embargo los creadores de JAVA decidieron no implementar la herencia múltiple por considerar que esta añade al código una gran complejidad (lo que hace que muchas veces los programadores que emplean programas que sí la soportan no lleguen a usarla).

Sin embargo para no privar a JAVA de la potencia de la herencia múltiple sus creadores introdujeron un nuevo concepto: el de interface. Una interface es formalmente como una clase, con dos diferencias: sus métodos están vacíos (métodos abstractos), no hacen nada, y a la hora de definirla en vez de emplear la palabra clave **`class`** se emplea interface.

```
interface Animal {
    public int edad = 10;
    public String nombre = "Lanudo";

    public void nace();

    public void getNombre();

    public void getNombre(int i);
}
```

Cabe preguntarnos cual es el uso de una interface si sus métodos están vacíos. Bien, cuando una clase implementa una interface lo que estamos haciendo es una promesa de que esa clase va a implementar todos los métodos de la interface en cuestión. Si la clase que implementa la interface no “sobrescribiera” alguno de los métodos de la interface automáticamente esta clase se convertiría en abstracta y no podríamos crear ningún objeto de ella. Para que un método sobrescriba a otro ha de tener el mismo nombre, se le han de pasar los mismos datos, ha de devolver el mismo tipo de dato y ha de tener el mismo modificador que el método al que sobrescribe. Si no tuviera el mismo modificador el compilador nos daría un error y no nos dejaría seguir adelante.

```
public class Perro3 implements Animal {
    // Compruévese como si cambiamos el nombre del método a nacio()
    // no compila ya que no hemos sobrescrito todos los métodos
    // de la interfaz.
    public void nace() {
          System.out.println("hola mundo");
    }

    public void getNombre() {
          System.out.println(nombre);
    }
```

```
    public void getNombre(int i) {
        System.out.println(nombre + " " + i);
    }

    public static void main(String[] args) {
        Perro3 dog = new Perro3();
        // Compruévese como esta línea da un error al compilar debido
        // a intentar asignar un valor a una variable final
        dog.edad = 8;
    }
}
```

Las variables que se definen en una interface llevan todas ellas el atributo final implícito, y es obligatorio darles un valor dentro del cuerpo de la interface. Además no pueden llevar modificadores private ni protected, sólo public. Su función es la de ser una especie de constantes para todos los objetos que implementen dicha interface. Por último decir que aunque una clase sólo puede heredar propiedades de otra clase puede implementar cuantas interfaces se desee, recuperándose así en buena parte la potencia de la herencia múltiple.

2.1.4 PROTECCIÓN DE CLASES

La protección de una clase determina la relación que tiene con otras clases de otros paquetes. Se distinguen dos niveles de protección: de paquete y publico. Una clase con nivel de protección **`package`** solo puede ser utilizada por las clases de su paquete. En cambio una clase **`public`** puede ser utilizada por cualquier otra clase de otro paquete, o sea, puede crear objetos y manipularlos. Por omisión todas la clases tienen nivel de protección de paquete, a menos que se especifique con la palabra reservada **`public`** delante.

Una clase de un determinado paquete puede hacer uso de otra clase de otro paquete de dos formas:

1. Utilizando su nombre completo en todas las partes del código donde haya que referirse a ella. Ej `java.lang.System.out.println("Hola Matias");`
2. Importando la clase, lo que posibilita referirse a ella simplemente por su nombre. Puede haber muchas sentencias import en un programa y deben siempre escribirse antes de cualquier definición de clase.

   Ej.
   ```
   import java.lang.System;    //se puede utilizar el comodín *
   ...                         //para poder importar todo java.lang.*
   System.out.println("Hola Matias");
   ```

2.1.5 COMPOSICIÓN

Una clase puede tener referencias a objetos de otras clases como miembros. A dicha capacidad se le conoce como composición y algunas veces como relación “tiene un”. Por ejemplo, un objeto de la clase RelojAlarma necesita saber la hora actual y la hora en la que se supone sonará su alarma, por lo que es razonable incluir dos referencias a objetos Tiempo como miembros del objeto RelojAlarma.

2.2 OBJETOS

Se puede decir que un objeto es una abstracción encapsulada genérica de datos y los procedimientos para manipularlos, también se puede decir que es una cosa o entidad, que tiene atributos (propiedades) y de formas de operar sobre ellos que se conocen como métodos. Todos los objetos tienen características o sea atributos (un auto tiene tamaño, peso, volumen, color) y todos tienen uno o varios comportamientos (un auto acelera, frena, dobla). Así como las personas se envían mensajes unas a otras, los objetos también se comunican mediante mensajes.

Por Ej.: un objeto cuenta bancaria puede recibir un mensaje para disminuir su saldo por cierta cantidad, debido a que el cliente ha retirado esa cantidad.

Los objetos pueden modelar diferentes cosas como; dispositivos, roles, organizaciones, sucesos, diseño o maquetado de ventanas, iconos, etc.

Por lo general, una clase consiste en uno o más métodos que manipulan los atributos pertenecientes a un objeto específico de la clase. Los atributos se representan como variables en la declaración de la clase. Dichas variables se llaman campos y se declaran dentro de la declaración de una clase, pero fuera de los cuerpos de las declaraciones de los métodos de ésta. Cuando cada objeto de una clase mantiene su propia copia de un atributo, el campo que representa a ese atributo se conoce también como variable de instancia; cada objeto (instancia) de la clase tiene una instancia separada de la variable en memoria.

El estado de un objeto se representa por sus variables de instancia. La mayoría de las declaraciones de variables de instancia van precedidas por la palabra clave **`private`**. Las variables o los métodos declarados con el modificador de acceso **`private`** son accesibles sólo para los métodos de la clase en la que se declaran. Este proceso se conoce como ocultamiento de datos, u ocultamiento de información. Cuando un programa crea (instancia) un objeto de la clase, la variable se encapsula (oculta) en el objeto, y sólo está accesible para los métodos de la clase de ese objeto. Esto evita que una clase en otra parte del programa modifique la variable por accidente.

Es necesario colocar un modificador de acceso antes de cada declaración de un campo y de un método. Por lo general las variables de instancia deben declararse como **`private`** y los métodos como **`public`**. (Es apropiado declarar ciertos métodos como **`private`**, si sólo van a estar accesibles para otros métodos de la clase).

En la Programación Orientada a Objetos, los objetos se comunican a través de señales o mensajes, siendo estos mensajes los que hacen que los objetos respondan de diferentes maneras. En otras palabras, un mensaje es una comunicación dirigida a un objeto, que le ordena que ejecute uno de sus métodos pudiendo o no llevar algunos parámetros.

Un método es una acción que determina como debe de actuar un objeto cuando recibe un mensaje. En analogía con un lenguaje procedural se le llamaría “función”. Un método también puede enviar mensajes a otros objetos, para realizar una acción o para pedir información.

En JAVA a un objeto se le conoce como instancia de una clase y para crearlo se llama a una parte de líneas de código conocidas con el nombre constructor de una clase que tienen el mismo nombre de la clase.

Una vez que ya no se ocupa el objeto, se ejecuta el recolector de basura.

2.3 CONTROL DEL ACCESO A LOS MIEMBROS

Los miembros de una clase son los atributos y los métodos, y su nivel de protección determina quien puede acceder a los mismos. Se pueden establecer distintos niveles de encapsulación en función de desde dónde queremos que se pueda acceder a ellos:

Visibilidad	Significado	JAVA	UML
Pública	Se puede acceder al miembro de la clase desde cualquier lugar.	`public`	+
Protegida	Solo se puede acceder al miembro de la clase desde la propia clase o desde una clase que herede de ella.	`protected`	#
Privada	Solo se puede acceder al miembro de la clase desde la propia clase.	`private`	-

Un miembro de una clase declarado `public` es accesible desde cualquier método definido dentro o fuera de la clase o paquete actual.

El principal propósito de los métodos `public` es presentar a los clientes de la clase una vista de los servicios que proporciona (la interfaz publica de la clase). Los clientes de la clase no necesitan preocuparse por la forma en que realiza sus tareas. Por esta razón, las variables y métodos `private` de una clase (es decir, los detalles de implementación de la clase) no son accesibles para sus clientes.

Los miembros de una clase `private`, puede ser accedido unicamente por los métodos de su clase, no son accesibles fuera de la clase. De intentar acceder a estos miembros, el compilador dará un mensaje de error.

El modificador de acceso `protected` puede aplicarse a todos los miembros de una clase, es decir, tanto a campos como a métodos o constructores. En el caso de métodos o constructores protegidos, estos serán visibles/utilizables por las subclases y otras clases del mismo `package`. El acceso protegido suele aplicarse a métodos o constructores, pero preferiblemente no a campos, para evitar debilitar el encapsulamiento.

Para encapsular por completo el estado de un objeto, todos sus atributos se declaran como variables de instancia privadas (usando el modificador de acceso `private`).

A un objeto siempre se accede a través de sus métodos públicos (su interfaz). Para usar el objeto no es necesario conocer qué algoritmos utilizan sus métodos ni qué tipos de datos se emplean para mantener su estado (su implementación).

Existen métodos de uso interno de la clase que también se declaran privados, pues no hacen a la interfaz de la clase.

2.4 MIEMBROS DE CLASE STATIC

Cada objeto tiene su propia copia de todas las variables de instancia de la clase. En ciertos casos, sólo debe compartirse una copia de cierta variable entre todos los objetos de una clase. En esos casos se utiliza un campo **static** (al cual se le conoce como una variable de clase). Una variable **static** representa información en toda la clase: todos los objetos de la clase comparten la misma pieza de datos. Si un objeto modifica su valor, todos los demás objetos tendrán esa modificación de la variable de clase. La declaración de una variable **static** comienza con la palabra clave **static**.

En realidad, los miembros **static** de una clase existen a pesar de que no existan objetos de esa clase; están disponibles tan pronto como la clase se carga en memoria, en tiempo de ejecución. Las variables **static** tienen alcance a nivel de clase.

Para acceder a un miembro **public static** cuando no existen objetos de la clase (y aún cuando sí existen), se debe anteponer el nombre de la clase y un punto (.) al miembro **static** de la clase, como en **Math.*PI***. Para acceder a un miembro **private static** cuando no existen objetos de la clase, debe proporcionarse un método **public static**, y para llamar a este método se debe calificar su nombre con el nombre de la clase y un punto.

Un método **static** no puede acceder a los miembros no **static** de la clase, ya que a un método **static** se le puede invocar aún cuando no se hayan creado instancias de objetos de la clase. Por la misma razón, no es posible usar la referencia **this** en un método **static**. La referencia **this** debe referirse a un objeto específico de la clase, y cuando se hace la llamada a un método **static**, podría darse el caso de que no hubiera objetos de su clase en la memoria. Hacer referencia a **this** en un método **static** es un error de compilación. Se conocen con el nombre de método de clase.

2.4.1 DECLARACIÓN STATIC IMPORT

Para invocar los campos y métodos **static** anteponemos a cada uno de ellos el nombre de la clase y un punto (.). Una declaración **static import** nos permite importar los miembros **static** de una clase o interfaz, para poder acceder a ellos mediante sus nombres no calificados en nuestra clase; el nombre de la clase y el punto (.) no se requieren para usar un miembro **static** importado.

Una declaración **static import** tiene dos formas: una que importa un miembro **static** específico (que se conoce como declaración **static import** individual) y una que importa a todos los miembros **static** de una clase (que se conoce como declaración **static import** sobre demanda).

La siguiente sintaxis importa un miembro **static** específico:

```
import static nombrePaquete.NombreClase.nombreMiembroEstático;
```

en donde nombrePaquete es el paquete de la clase (Ej. java.lang), NombreClase es el nombre de la clase (Ej. **Math**) y nombreMiembroEstático es el nombre del campo o método **static** (Ej. ***PI*** o ***abs***).

La siguiente sintaxis importa todos los miembros **static** de una clase:

```
import static nombrePaquete.NombreClase.*
```

El asterisco (*) indica que todos los miembros **static** de la clase especificada deben estar disponibles para usarlos en el archivo. Las declaraciones **static import** sólo importan miembros de clase **static**. Las instrucciones **import** regulares deben usarse para especificar las clases utilizadas en un programa.

2.5 VARIABLES DE INSTANCIA FINAL

Se puede utilizar la palabra clave final para especificar que una variable de instancia no puede modificarse (es decir, que sea una constante) y que cualquier intento por modificarla sería un error. Por ejemplo,

```
private final int INCREMENTO;
```

declara una variable de instancia final (constante) llamada INCREMENTO, de tipo int. Dichas variables se pueden inicializar al momento de declararse. De lo contrario, se debe hacer en cada uno de los constructores de la clase. Al inicializar las constantes en los constructores, cada objeto de la clase puede tener un valor distinto para la constante. Si una variable final no se inicializa en su declaración o en cada constructor, se produce un error de compilación.

Las variables de instancia pueden declararse como **final** para indicar que no pueden modificarse una vez que se inicializaron. Dichas variables no son variables, son constantes. Para definir una constante dentro de una clase se lo hace con la palabra reservada **final**. Si además se define la constante como **static**, se convierte en una constante de clase, o sea todos los objetos de la clase comparten dicha constante.

```
public class Circulo {
    private float radio;
    private final static float pi = 3.1415f;
    …
    public float perimetro(){
          return 2*pi*this.radio ;
    }
}
```

En este ejemplo pi es una constante, que al ser declarada **static** es compartida con todos los objetos de la clase Circulo. Si no estuviese declarada como **static**, cada objeto de la clase Circulo tendrá una copia de la constante pi (que no tendría sentido, por eso la declaramos static).

2.6 REFERENCIAS A LOS MIEMBROS DEL OBJETO ACTUAL

Cada objeto puede acceder a una referencia a sí mismo mediante la palabra clave **this** (también conocida como referencia this). Cuando se hace una llamada a un método no **static** para un objeto específico, el cuerpo del método utiliza en forma implícita la palabra clave **this** para hacer referencia a las variables de instancia y otros métodos. Esto permite al código de la clase saber qué objeto se debe manipular.

A menudo se produce un error lógico cuando un método contiene un parámetro o variable local con el mismo nombre que un campo de la clase. En tal caso, use la referencia **this** si desea acceder al campo de la clase; de no ser así, se hará referencia al parámetro o variable local del método.

Para conservar la memoria, JAVA mantiene sólo una copia de cada método por clase; todos los objetos de la clase invocan a este método. Por otro lado, cada objeto tiene su propia copia de las variables de instancia de la clase (es decir, las variables no **static**). Cada método de la clase utiliza en forma implícita la referencia **this** para determinar el objeto especifico de la clase que se manipulará.

```
public class CuentaBancaria {
	private double saldo;
	private double limite;

	public void depositar (double importe){
		this.saldo = this.saldo + importe;
	}
}
```

2.7 MÉTODOS

Un método (también conocido como función o procedimiento en otros lenguajes) es una colección de sentencias que ejecutan una tarea especifica. En JAVA, un método siempre pertenece a una clase y sera un comportamiento de los objetos que se creen de la misma.

La definición de un método consta de una cabecera y del cuerpo del método encerrado entre llaves.

```
[modificador] tipo_resultado nombreMetodo ([lista de parámetros])
{
    declaraciones de variables locales;
    sentencias;
    [return [expresion]];
}
```

Las variables declaradas en el cuerpo del método son locales al mismo.

Para promover la reutilización de software, cada método debe limitarse de manera que realice una sola tarea bien definida, y su nombre debe expresar esa tarea con efectividad.

Los argumentos para los métodos pueden ser constantes, variables o expresiones.

Los métodos pueden devolver a lo máximo un valor, pero el valor devuelto puede ser una referencia a un objeto que contenga muchos valores.

Hay tres formas de llamar a un método:

1. Utilizar el nombre de un método por sí solo para llamar a otro método de la misma clase;
2. Usar una variable que contiene una referencia a un objeto, seguida de un punto (.) y del nombre del método para llamar a un método (no **static**) del objeto al que se hace referencia;
3. Utilizar el nombre de la clase y un punto (.) para llamar a un método **static** de una clase.

Un método **static** sólo puede llamar directamente a otros métodos **static** de la misma clase (es decir, mediante el nombre del método por sí solo) y sólo puede manipular de manera directa variables **static** en la misma clase. Para acceder a los miembros no **static** de la clase, un método **static** debe usar una referencia a un objeto de esa clase. Recuerde que los métodos **static** se relacionan con una clase como un todo, mientras que los métodos no **static** se asocian con una instancia específica (objeto) de la clase y pueden manipular las variables de instancia de ese objeto.

Existen dos formas de regresar el control a la instrucción que llama a un método:

1. Si el método no devuelve un resultado, el control regresa cuando el flujo del programa llega a la llave derecha de terminación del método;
2. Si el método devuelve un resultado, la instrucción return expresión; evalúa la expresión y después devuelve el resultado al método que hizo la llamada.

El hecho de que un método regrese valores o no es opcional. Cuando un método no retorna valores, se deberá modificar la definición de sus parámetros y quedará como sigue:

```
void nombreMetodo ([lista de parámetros]){
    sentencia1;
    sentencia2;
    sentenciaN;
}
```

En donde se deberá utilizar la palabra reservada **void** que le indica al compilador y al método que su valor de retorno será "vacío". Dentro del cuerpo de sus sentencias se omite la sentencia **return** por no "regresar" valor alguno a donde fue llamado. Los parámetros de igual manera son opcionales, aunque los paréntesis van por obligación.

Cuando decimos que llamamos un método, estamos haciendo uso de las sentencias que lo conforman. De igual manera se dice que se invoca al método. Los métodos se invocan (llaman) por el nombre definido en su declaración y se deberá pasar la lista de argumentos entre paréntesis.

Por ejemplo, se definirá un método que imprima a pantalla un mensaje para el usuario.

```
public static void saludoUsuario(String nombre) {
    String cadena_Saludo = "Hola " + nombre;
    System.out.println(cadena_Saludo);
    System.out.println("Mensaje impreso desde un método estático");
}
```

Cuando ya se ha definido el método y sus sentencias son las necesarias y adecuadas, ahora si se hace una invocación a él, o mejor dicho, se invocan las sentencias que conforman el cuerpo del método. Se hace de la siguiente manera:

```
public static void main(String[] args) {
    String nombre = "Prof. Matías García";
    saludoUsuario(nombre);
}
```

Cuando los métodos declarados en JAVA regresan valores, quiere decir que se espera recibir alguna respuesta o resultado de alguna operación realizada por las sentencias que conforman su cuerpo.

Los parámetros de un método se pueden entender como los valores que éste recibe desde la parte del código donde es invocado. Los parámetros pueden ser de tipos primitivos (int, double, float, entre otros) u objetos. En la declaración del método se escribe qué parámetros recibirá dentro de los paréntesis separando cada uno de ellos por una coma, indicando el tipo de cada uno de ellos y el identificador, se entenderá también que no hay límites en el número de parámetros que se envían al método.

Cuando el método ya ha recibido (o consigue) los elementos necesarios para poder realizar sus funciones, retornará el resultado de aplicar esas funciones sobre los parámetros recibidos. Se deberá indicar en la declaración del método qué tipo de dato resultado retornará e indicarle en el cuerpo de sus sentencias la palabra reservada **return** y aquí le indicaremos qué regresará.

```
public double obtenerPotencia(double base, double potencia) {
    double resultado = Math.pow(base, potencia);
    return resultado;
}
```

Cuando se invoque (llame) el método que se quiera utilizar, deberemos pasarle los parámetros indicados (necesarios) en su declaración. Los parámetros se escriben a continuación del nombre del método que se quisiera utilizar, de igual manera entre paréntesis pero al hacer la invocación no se indica el tipo de dato de los parámetros, aunque las variables que enviemos forzosamente deberán coincidir con el tipo que pretendemos utilizar.

```
public static void main(String[] args) {
    double b = 3;
    double p = 2;
    double valor_resultante;

    valor_resultante = obtenerPotencia(b,p);
    System.out.println(b + " a la " + p + " = " + valor_resultante);
}
```

Recordar que en JAVA los parámetros de tipo primitivo (**int**, **long**, **double**, ...) siempre se pasan por valor. Las variables de tipo objeto y arrays se pasan por referencia (referencia a su dirección de memoria de alojamiento).

2.7.1 MÉTODOS CON UNA CANTIDAD VARIABLE DE ARGUMENTOS

Un tipo de parámetro que va precedido de tres puntos (**...**) en la lista de parámetros de un método, indica que el método recibe una cantidad variable de argumentos en la llamada. Puede haber un solo parámetro de este tipo dentro de la lista de parámetros del método que además debe ser el último en la lista de parámetros.

JAVA trata a la lista con una cantidad variable de parámetros como un arreglo cuyos elementos son del mismo tipo.

La sintaxis es:

```
[modificador] tipo_resultado nombreMetodo ([lista de parámetros], [<tipo> …
<nombre_parametro>])
```

donde la `[lista de parámetros]` puede ser vacía.

Ej. si quisiéramos calcular el promedio de una cantidad variable de notas, podríamos definir el método así:

```
public class PruebaCantVariableArg {
    public static double promedio(double... notas) {
          double sum = 0.0;
          for (double val : notas)
                sum += val;

          return sum / notas.length;
    }

    public static void main (String[] args){
          System.out.printf("El promedio entre 8 y 5 es %.2f\n", promedio(8,5));
          System.out.printf("El promedio entre 8, 5 y 7 es %.2f\n",
promedio(8,5,7));
          System.out.printf("El promedio entre 8, 5, 7, 2 y 9 es %.2f\n",
promedio(8,5,7,2,9));
    }
}
```

2.7.2 SOBRECARGA DE MÉTODOS

Pueden declararse métodos con el mismo nombre en la misma clase, siempre y cuando tengan distintos conjuntos de parámetros (que se determinan con base en el número, tipos y orden de los parámetros).

A esto se le conoce como sobrecarga de métodos. Cuando se hace una llamada a un método sobrecargado, el compilador de JAVA selecciona el método apropiado mediante un análisis del número, tipos y orden de los argumentos en la llamada. Por lo general, la sobrecarga de métodos se utiliza para crear varios métodos con el mismo nombre que realicen la misma tarea o tareas similares, pero con distintos tipos o números de argumentos.

Cuando dos métodos tienen la misma firma (número, tipos y orden de los parámetros) pero distintos

tipos de valores de retorno, genera un mensaje de error para indicar que el método ya está definido en la clase. Los métodos sobrecargados pueden tener tipos de valor de retorno distintos si tienen distintas listas de parámetros.

2.8 Inicialización de objetos mediante constructores

Cada clase que usted declare puede proporcionar un método especial llamado constructor, el cual puede utilizarse para inicializar un objeto de una clase al momento de crearlo. De hecho, JAVA requiere una llamada al constructor para cada objeto que se crea. La palabra clave **new** solicita memoria del sistema para almacenar un objeto, y después llama al constructor de la clase correspondiente para inicializar el objeto. La llamada se indica mediante el nombre de la clase, seguido de paréntesis. Un constructor debe tener el mismo nombre que la clase.

Toda clase debe tener cuando menos un constructor. Si no se proporcionan constructores en la declaración de una clase, el compilador crea un constructor predeterminado que no recibe argumentos cuando se le invoca. El constructor predeterminado inicializa las variables de instancia con los valores iniciales especificados en sus declaraciones, o con sus valores predeterminados (cero para los tipos primitivos numéricos, **false** para los valores **boolean** y **null** para las referencias).

Si su clase declara constructores, el compilador no creará uno predeterminado. En este caso, debe declarar un constructor sin argumentos si se requiere una inicialización predeterminada. Al igual que un constructor predeterminado, uno sin argumentos se invoca con paréntesis vacíos.

Al igual que un método, un constructor especifica en su lista de parámetros los datos que requiere para realizar su tarea. Cuando usted crea un nuevo objeto, estos datos se colocan en los paréntesis que van después del nombre de la clase.

Una importante diferencia entre los constructores y los métodos es que los constructores no pueden devolver valores, por lo cual no pueden especificar un tipo de valor de retorno (ni siquiera **void**).

Por lo general, los constructores se declaran como **public**.

También se puede crear una clase con varios constructores sobrecargados, que permiten a los objetos de esa clase inicializarse de distintas formas. Para sobrecargar los constructores, sólo hay que proporcionar varias declaraciones del constructor con distintas firmas.

Un constructor puede llamar a los métodos de la clase. Tenga en cuenta que tal vez las variables de instancia no estén aún inicializadas, ya que el constructor está en el proceso de inicializar el objeto. El uso de variables de instancia antes de inicializarlas en firma apropiada es un error lógico.

Si un programa intenta inicializar un objeto de una clase al pasar el número incorrecto de tipos de argumentos a su constructor, ocurre un error de compilación.

```
public class CuentaBancaria {
    // Constante
    public static final double LIMITE_NORMAL = 3000.00;
    // Variables de instancia
    private double saldo;
    private double limite;
    // Constructor sin parámetros
    public CuentaBancaria() {
          this.saldo = 0;
```

```
            this.limite = LIMITE_NORMAL;
      }
      // Constructor con parámetros
      public CuentaBancaria(double limit) {
            this.saldo = 0;
            this.limite = limit;
      }
      // Constructor con parámetros
      public CuentaBancaria(double saldoInicial, double limit) {
            this.saldo = saldoInicial;
            this.limite = limit;
      }
      // Método para depositar
      public void depositar(double importe) {
            this.saldo = this.saldo + importe;
      }

      public void mostrarDatosCuenta (){
            System.out.println("Saldo de cuenta " + this.saldo);
            System.out.println("Limite de cuenta " + this.limite);
      }

      public static void main(String[] args) {
            //Llama a constructor sin parámetros
            CuentaBancaria CuentaNormal1 = new CuentaBancaria();
            CuentaNormal1.mostrarDatosCuenta();
            //Llama a constructor con parámetro que modifica el limite
            CuentaBancaria CuentaEspecial1 = new CuentaBancaria(5000);
            CuentaEspecial1.mostrarDatosCuenta();
            //Llama a constructor con parámetros que modifican el saldo y el limite
            CuentaBancaria CuentaConSaldoInicial1 = new CuentaBancaria(2500, 8000);
            CuentaConSaldoInicial1.mostrarDatosCuenta();
      }

}
```

2.9 Los métodos establecer (set) y obtener (get)

Los campos **private** de una clase pueden manipularse sólo mediante los métodos de esa clase. Por lo tanto, un cliente de un objeto (es decir, cualquier clase que llame a los métodos del objeto) llama a los métodos **public** de la clase para manipular los campos **private** de un objeto de esa clase.

A menudo, las clases proporcionan métodos **public** para permitir a los clientes de la clase establecer (asignar valores a) u obtener (obtener los valores de) variables de instancia **private**. Los nombres de estos métodos no necesitan empezar con *set* o *get*, pero esta convención de nomenclatura es muy recomendada en JAVA, y es requerida para ciertos componentes de software especiales de JAVA, conocidos como JAVABeans, que pueden simplificar la programación en muchos entornos de desarrollo integrados (IDE).

Si una variable de instancia se declara como **public**, cualquier método que tenga una referencia a un objeto que contenga esta variable de instancia podrá leer o escribir en ella. Si una variable de instancia se declara como **private**, solo un método *get* **public** permitirá a otros métodos el acceso a la variable, pero el método *get* puede controlar la manera en que el cliente puede tener acceso a ella. Por ejemplo, un método *get* podría controlar el formato de los datos que devuelve y, por ende, proteger el código cliente de la representación actual de los datos. Un método *set* **public** puede (y debe) escudriñar con cuidado los intentos por modificar el valor de la variable, y lanzar una excepción si es necesario. Por ejemplo, un intento por establecer el día del mes en una fecha 37 sería rechazado, un intento por establecer el peso de una persona en un valor negativo sería negado, etc. Entonces, aunque los métodos *set* y *get* proporcionan acceso a los datos **private** , el acceso se restringe mediante la implementación de los métodos. Esto ayuda a promover la buena ingeniería de software.

Por lo general para cada atributo definido en una clase se definen un método *set* para darle un valor a ese atributo y un método *get* para obtener el valor del atributo a través de ese método.

Por convención el nombre de un método *set* para un atributo, está formado por la palabra set seguido del nombre del atributo con su primera letra en mayúscula.

Todo método *set* tiene un parámetro del mismo tipo que el atributo que quiere modificar y no devuelve ningún valor.

La convención para el nombre de un método *get* para un atributo, es la misma que para los métodos *set*. Pero los métodos *get* no tienen parámetros y devuelven un valor del tipo del atributo que definen.

```
public double getSaldo() {
    return saldo;
}

public void setSaldo(double saldo) {
    this.saldo = saldo;
}
```

2.10 Recolección de basura y el método finalize

Todo objeto utiliza recursos del sistema, como la memoria. Necesitamos una manera disciplinada de regresarlos al sistema cuando ya no se necesitan; de lo contrario, podrían ocurrir “fugas de recursos” que impidan que nuestro programa, o tal vez hasta otros programas, los utilicen. La máquina virtual de JAVA (JVM) realiza la recolección de basura en forma automática para reclamar la memoria ocupada por los objetos que ya no se usan. Cuando ya no hay más referencias a un objeto, éste se convierte en candidato para la recolección de basura. Por lo general, esto ocurre cuando la JVM ejecuta su recolector de basura. Por lo tanto, las fugas de memoria que son comunes en otros lenguajes como C y C++ (debido a que en esos lenguajes, la memoria no se reclama de manera automática) son menos probables en JAVA, pero algunas pueden ocurrir de todas formas, aunque con menos magnitud. Pueden ocurrir otros tipos de fugas de recursos. Por ejemplo, una aplicación podría abrir un archivo en disco para modificar el contenido. Si la aplicación no cierra el archivo, ningún a otra aplicación puede utilizarlo sino hasta que termine la que lo abrió.

El recolector de basura llama al método *finalize* para realizar las tareas de preparación para la terminación sobre un objeto, justo antes de que el recolector de basura reclame la memoria de ese objeto.

El método *finalize* no recibe parámetros y tiene el tipo de valor de retorno `void`. Un problema con el método *finalize* es que no se garantiza que el recolector de basura se ejecute en un tiempo especificado. De hecho, tal vez el recolector de basura nunca se ejecute antes de que termine un programa. Por ende, no queda claro si (o cuándo) se hará la llamada al método *finalize*. Por esta razón, la mayoría de los programadores deben evitar el uso del método *finalize*.

Una clase que utiliza recursos del sistema, como archivos en el disco, debe proporcionar un método que los programadores puedan llamar para liberar recursos cuando ya no se necesiten en un programa. Muchas clases de la API de JAVA proporcionan métodos *close* o *dispose* para este propósito.

2.11 CREACIÓN DE PAQUETES

Un paquete es un conjunto de clases (e interfaces), lógicamente relacionadas entre sí, agrupadas bajo un nombre (Ej. el paquete java.io agrupa las clases que permiten a un programa realizar la entrada y salida de información); incluso, un paquete puede contener a otros paquetes. Esto ayuda a organizar las clases en grupos para clasificar el acceso a las mismas cuando sean necesarias en la programación.

Las clases de bibliotecas preexistentes, como la API de JAVA, pueden importarse en un programa en JAVA. Cada clase en la API de JAVA pertenece a un paquete que contiene un grupo de clases relacionadas. Estos paquetes se definen una vez, pero se pueden importar en muchos programas. A medida que las aplicaciones se vuelven más complejas, los paquetes nos ayudan a administrar la complejidad de los componentes de una aplicación. Los paquetes también facilitan la reutilización de software, al permitir que los programas importen clases de otros paquetes, en vez de copiar las clases en cada uno de los programas que las utiliza. Otro beneficio de los paquetes es que proporcionan una convención para los nombres de clases únicos, lo cual ayuda a evitar los conflictos de nombres de clases.

Antes de poder importar una clase en varias aplicaciones, ésta debe colocarse en un paquete para que sea reutilizable.

Los pasos para crear una clase reutilizable son:

1. Declarar la clase `public`. De lo contrario, sólo la podrán usar otras clases en el mismo paquete.
2. Seleccione un nombre único para el paquete y agregue una declaración `package` al archivo de código fuente para la declaración de la clase reutilizable. Sólo puede haber una declaración `package` en cada archivo de código fuente de JAVA, y debe ir antes que todas las demás declaraciones e instrucciones en el archivo.

   Si no se proporciona una instrucción `package`, la clase se coloca en lo que se conoce como paquete predeterminado y sólo es accesible para las demás clases en el paquete predeterminado que se encuentran en el mismo directorio.
3. Compilar la clase de manera que se coloque en la estructura de directorio del paquete apropiada.
4. Importar la clase reutilizable en un programa para utilizarla.

El compilador utiliza un objeto especial, llamado cargador de clases, para localizar las clases que necesita.

El cargador de clases empieza buscando las clases estándar de JAVA que se incluyen con el JDK. Después busca los paquetes opcionales. JAVA cuenta con un mecanismo de extensión que permite agregar paquetes nuevos (opcionales), para fines de desarrollo y ejecución. Si la clase no se encuentra en las clases estándar de JAVA o en las clases de extensión, el cargador de clases busca en la ruta de clases, que contiene una lista de ubicaciones en la que se almacenan las clases. La ruta de clases consiste en una lista de directorios o archivos de ficheros, cada uno separado por un separador de directorio: un signo de punto y coma (;) en MS Windows o un signo de dos puntos (:) en UNIX/Linux/Mac OS X. Los archivos de ficheros son archivos individuales que contienen directorios de otros archivos, por lo general en formato comprimido. Por ejemplo, las clases estándar de JAVA que utilizamos en los programas están contenidas en el archivo de ficheros rt.jar, el cual se instala junto con el JDK. Los archivos de ficheros generalmente terminan con la extensión .jar o .zip. Los directorios y archivos de ficheros que se

especifican en la ruta de clases contienen las clases que deseamos poner a disponibilidad del compilador y la máquina virtual de JAVA.

De manera predeterminada, la ruta de clases consiste sólo del directorio actual. Sin embargo, la ruta de clases puede modificarse de la siguiente manera:

1. proporcionando la opción -classpath al compilador javac;
2. estableciendo la variable de entorno CLASSPATH (una variable especial que se define y el sistema operativo mantiene, de manera que las aplicaciones puedan buscar clases en las ubicaciones especificadas).

Todos los archivos .JAVA de un paquete del proyecto que hagan uso de clases contenidas en otros paquetes, deben contener la directiva import seguido del nombre del paquete, punto y la clase que necesita usar. Si son muchas clases del paquete puede utilizar el * para referirse a todas las clases del paquete

Un punto importante, es que la importación de paquetes y de clases, indica únicamente al compilador de JAVA donde buscar el código que necesita, sin aumentar el tamaño del programa.

2.12 Interfaz de Programación de Aplicaciones (API)

Para escribir programas en JAVA, se combinan los nuevos métodos y clases, desarrollados por el programador, con los métodos y clases predefinidos, que están disponibles en la Interfaz de Programación de Aplicaciones de JAVA (también conocida como la API de JAVA o biblioteca de clases de JAVA) y en diversas bibliotecas de clases. Por lo general, las clases relacionadas están agrupadas en paquetes, de manera que se pueden importar a los programas y reutilizarse. La API de JAVA proporciona una vasta colección de clases predefinidas que contienen métodos para realizar cálculos matemáticos comunes, manipulaciones de cadenas, manipulaciones de caracteres, operaciones de entrada/salida, operaciones de bases de datos, operaciones de red, procesamiento de archivos, comprobación de errores y muchas otras tareas útiles.

Para cada nueva clase de la API de JAVA que utilicemos, hay que indicar el paquete en el que se ubica.

Esta información nos ayuda a localizar las descripciones de cada paquete y clase en la documentación de la API de JAVA. Puede encontrar una versión basada en Web de esta documentación en

https://docs.oracle.com/javase/8/docs/api/

La versión 2 de JAVA, presenta un conjunto de cerca de sesenta paquetes, los cuales están dentro de del paquete general llamado java.*, entre los más importantes se encuentran:

- java.lang.-Tiene las clase con las que puede trabajar el programa principal.
- java.util.-Se encuentran clases especializadas como son la de los calendarios.
- java.io.-Proporciona un archivo independiente del dispositivo y servicios de Entrada/Salida.
- java.awt.-Contiene la mayoría de las clases dedicadas a los gráficos.

- java.net.-Son las clases que pueden trabajar con los programas de bajo nivel en Internet y WWW.
- java.applet.-Cuenta con una clase que puede trabajar con el lenguaje HTML, el cual se utiliza en los applets de JAVA.

2.12.1 Observaciones acerca del uso de las declaraciones import

Las clases System y String están en el paquete java.lang, que se importa de manera implícita en todo programa de JAVA, por lo que todos los programas pueden usar las clases de ese paquete sin tener que importarlas de manera explícita. La mayoría de las otras clases que utilizará en los programas de JAVA deben importarse de manera explícita.

Hay una relación especial entre las clases que se compilan en el mismo directorio en el disco. De manera predeterminada, se considera que dichas clases se encuentran en el mismo paquete; a éste se le conoce como el paquete predeterminado (default). Las clases en el mismo paquete se importan implícitamente en los archivos de código fuente de las otras clases en el mismo paquete. Por ende, no se requiere una declaración import cuando la clase en un paquete utiliza a otra en el mismo paquete.

2.13 UML

UML (Unified Modeling Language) es un lenguaje que permite modelar, especificar, construir y documentar los elementos que forman un sistema de software orientado a objetos. Un modelo es una simplificación de la realidad.

El modelado es esencial en la construcción de software para...

- Comunicar la estructura de un sistema complejo
- Especificar el comportamiento deseado del sistema
- Comprender mejor lo que estamos construyendo
- Descubrir oportunidades de simplificación y reutilización.

Un modelo proporciona “los planos” de un sistema y puede ser más o menos detallado, en función de los elementos que sean relevantes en cada momento. El modelo ha de capturar “lo esencial”.

Todo sistema puede describirse desde distintos puntos de vista:

- Modelos estructurales (organización del sistema)
- Modelos de comportamiento (dinámica del sistema)

Se ha convertido en el estándar de facto de la industria, debido a que ha sido concebido por los autores de los tres métodos más usados de orientación a objetos: Grady Booch, Ivar Jacobson y Jim Rumbaugh. Estos autores fueron contratados por la empresa Rational Software Co. para crear una notación unificada en la que basar la construcción de sus herramientas CASE. En el proceso de creación de UML han participado, no obstante, otras empresas de gran peso en la industria como Microsoft, Hewlett-Packard, Oracle o IBM, así como grupos de analistas y desarrolladores.

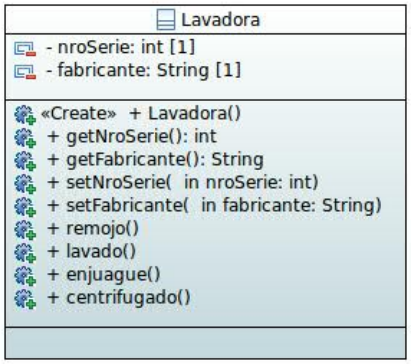
Modelar el sistema utilizando los diagramas de UML, significara en definitiva contar con documentos que plasman el trabajo de capturar la idea para la posterior evolución del proyecto. El cliente podrá entender el plan de trabajo de los especialistas y señalar cambios si no se capto correctamente alguna necesidad; o bien, indicar cambios sobre la marcha del proyecto. A su vez, los especialistas encargados del desarrollo generalmente trabajaran en equipo, por lo que cada uno de ellos podrá identificar su trabajo particular y el general a partir de los diagramas UML.

UML proporciona las herramientas para organizar un diseño solido y claro, que comprendan los especialistas involucrados en las distintas etapas de la evolución del proyecto, y por que no para documentar un anteproyecto que será entregado al cliente.

UML esta compuesto por diversos elementos gráficos que se combinan para conformar diagramas. Al ser UML un lenguaje, existen reglas para combinar dichos elementos. En conjunto, los diagramas UML brindan diversas perspectivas de un sistema, por ende el modelo. Ahora bien, el modelo UML describe lo que hará el sistema y no como será implementado.

Algunos de los diagramas de modelado UML mas utilizados:

Diagrama de clases



Un diagrama de clases representa en un esquema gráfico, las clases u objetos intervinientes y como se relacionan en su escenario, sistema o entorno. Con estos diagramas, se logra diseñar el sistema a ser desarrollado en un lenguaje de programación, generalmente orientado a objetos.

Estos diagramas los incorporan algunos entornos de desarrollo, tal es el caso de Eclipse con el plugin Papyrus o Netbeans con su respectivo plugin easyUML. Es un buen hábito generar proyectos UML con sus respectivos diagramas de clases para luego automáticamente obtener código fuente que nos colabore en el desarrollo del sistema o software.

Diagramas de casos de uso

Describen las acciones de un sistema desde el punto de vista del usuario. Si la finalidad es crear un sistema que pueda ser usado por la gente en general, es importante este diagrama, ya que permite a los analistas de sistemas y desarrolladores (programadores) obtener los requerimientos desde el punto de vista del usuario.

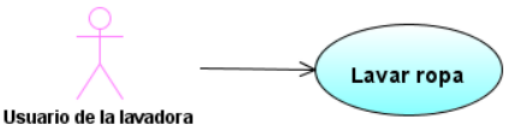


Diagrama de estados

Muestra las transiciones de un objeto en sus cambios de estados. Una lavadora puede estar en las fases de remojo, lavado, enjuague, centrifugado o apagada. Un elevador se puede mover hacia abajo, hacia arriba o estar en esta de reposo.

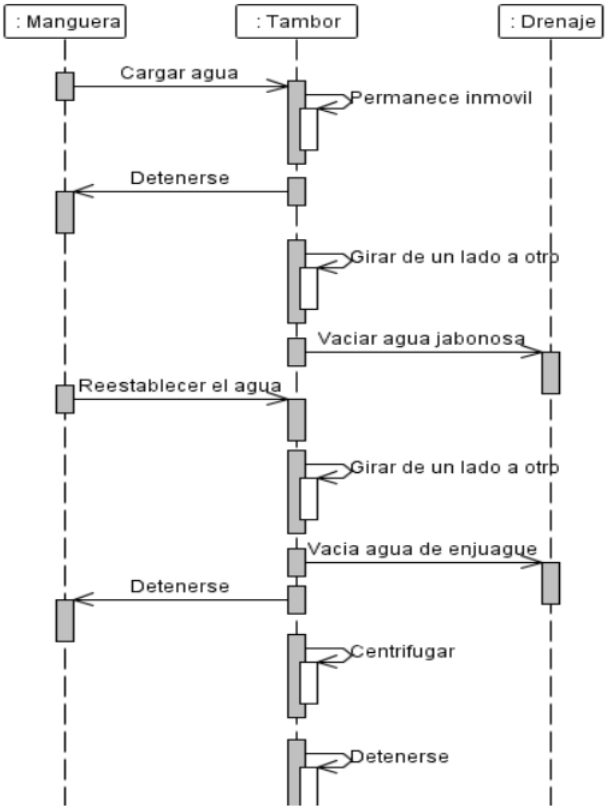


Diagrama de secuencias

Representan información dinámica ya que los objetos interactúan entre si mientras el tiempo transcurre. En definitiva, los diagramas de secuencias, visualizan la mecánica de interacciones entre objetos con base en tiempos. En el Ej. de la lavadora, encontramos los componentes manguera, tambor y drenaje como objetos que interactúan mientras transcurre el tiempo de funcionamiento.

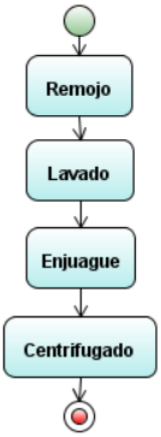
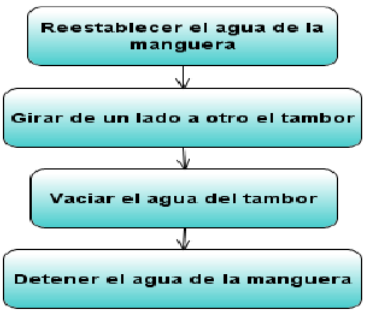


Diagrama de actividades



En un caso de uso como en el comportamiento de objetos en un sistema, siempre hay actividades que generalmente son secuenciales. Sin importar el tiempo, podemos reflejar en el diagrama de actividades, la secuencia de acciones que desarrollan los objetos.

Representación gráfica de una clase (notación UML)

Una clase se representa con un rectángulo dividido en tres partes:

- El nombre de la clase (identifica la clase de forma unívoca)
- Sus atributos (datos asociados a los objetos de la clase)
- Sus métodos (comportamiento de los objetos de esa clase)

Notación UML:

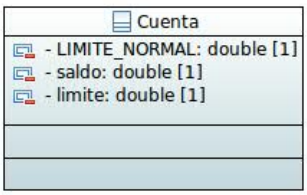
Atributos [visibilidad] nombre [multiplicidad] [: tipo [= valor_por_defecto]]

Métodos [visibilidad] nombre ([[in|out] parámetro : tipo [, ...]])[:tipo_devuelto]

- Los corchetes indican partes opcionales.
- Visibilidad: privada (-), protegida (#) o pública (+)
- Multiplicidad entre corchetes (p.ej. [2], [0..2], [*], [3..*])
- Parámetros de entrada (in) o de salida (out).

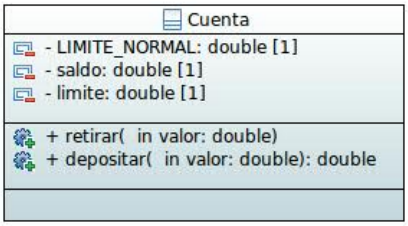
Declaración en Java de una clase con sus atributos:

```
public class Cuenta {
    private static double LIMITE_NORMAL = 5000;
    private double saldo;
    private double limite;
}
```



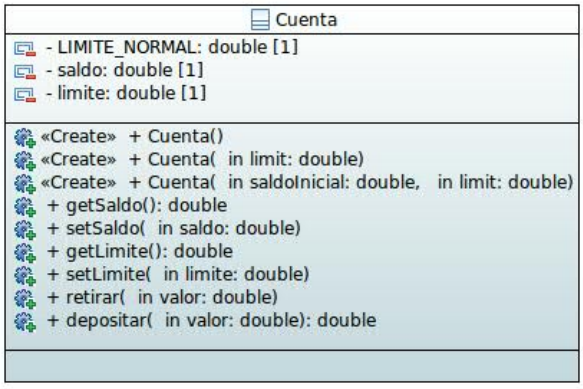
Representación de las operaciones de una clase:

```
public class Cuenta {
    private static double LIMITE_NORMAL = 5000;
    private double saldo;
    private double limite;

    public void retirar (double valor){
          this.saldo = this.saldo - valor;
    }
     public double depositar (double valor){
          double nuevoSaldo;
          nuevoSaldo = this.saldo + valor;
          return nuevoSaldo;
    }
}
```



Representación completa de una clase:

```
public class Cuenta {
    private static double LIMITE_NORMAL = 5000;
    private double saldo;
    private double limite;

    public Cuenta(){
          this.saldo = 0;
          this.limite = LIMITE_NORMAL;
    }
    public Cuenta(double limit) {
          this.saldo = 0;
          this.limite = limit;
    }
    public Cuenta(double saldoInicial, double
limit) {
          this.saldo = saldoInicial;
          this.limite = limit;
    }
    public double getSaldo() {
          return saldo;
    }
    public void setSaldo(double saldo) {
          this.saldo = saldo;
    }
    public double getLimite() {
          return limite;
    }
    public void setLimite(double limite) {
          this.limite = limite;
```



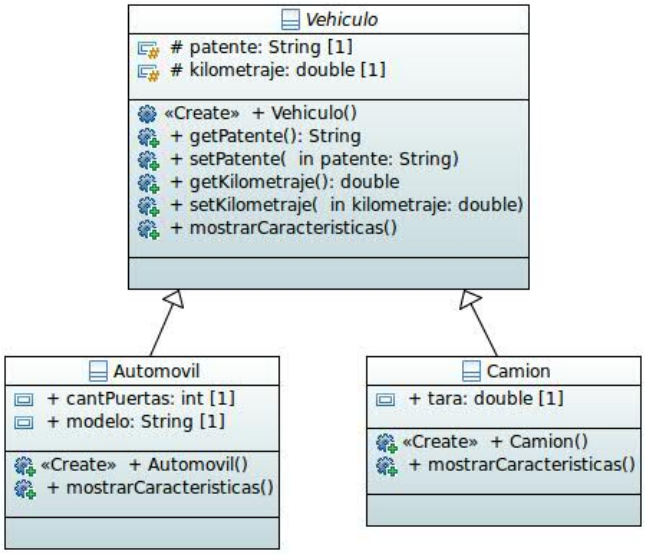
```
	}
	public void retirar (double valor){
		this.saldo = this.saldo - valor;
	}
	public double depositar (double valor){
		double nuevoSaldo;
		nuevoSaldo = this.saldo + valor;
		return nuevoSaldo;
	}
}
```

Relaciones entre clases

Una relación representa el detalle del vínculo entre dos clases, destacando el tipo (cual es la relación), la cardinalidad o multiplicidad (cantidad de objetos de una y otra clase) y la navegabilidad (que objeto puede observar a otro). Ante un diseño orientado a objetos, es importante conocer la diversidad de relaciones que se pueden producir, necesitar o establecer entre clases.

Las relaciones existentes entre las distintas clases nos indican cómo se comunican los objetos de esas clases entre sí. Los mensajes “navegan” por las relaciones existentes entre las distintas clases.

Relación de generalización: se basa en los elementos comunes encontrados en dos o mas clases que permiten reunidos ser generalizados hacia una clases superior (herencia). Con este concepto, al ser instanciada una clase derivada, se heredan propiedades y métodos de la clase superior. Las clases superiores pueden ser abstractas, con lo que podremos aprovechar el concepto de métodos polimórficos.



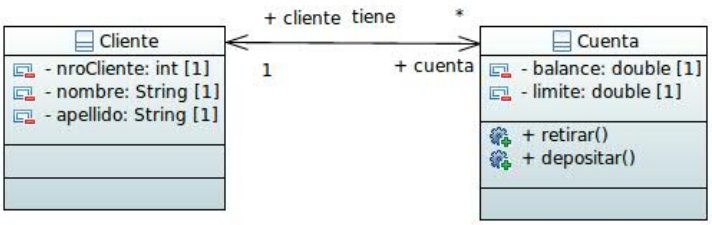
La Subclase además de poseer sus propios métodos y atributos, poseerá las características y atributos visibles de la Super Clase (public y protected).

```
public abstract class Vehiculo {
	protected String patente;
	protected double kilometraje;

	public Vehiculo() {
		//sentencias constructor
	}

	public String getPatente() {
		return patente;
	}

	public void setPatente(String patente) {
		this.patente = patente;
	}

	public double getKilometraje() {
		return kilometraje;
```

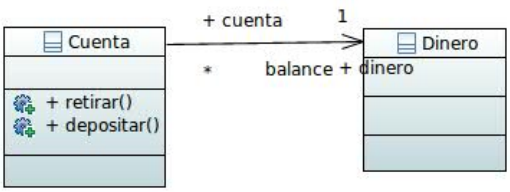
```
    }
    public void setKilometraje(double kilometraje) {
        this.kilometraje = kilometraje;
    }
    public void mostrarCaracteristicas (){
        //sentencias método
    }
};
public class Automovil extends Vehiculo {
    public int cantPuertas;
    public String modelo;
    public Automovil() {
        //sentencias constructor Automovil
    }
    public void mostrarCaracteristicas() {
        //sentencias sobrecarga de método
    }
};
public class Camion extends Vehiculo {
    public double tara;
    public Camion() {
        //sentencias constructor Camion
    }
    public void mostrarCaracteristicas() {
        //sentencias sobrecarga de método
    }
};
```

Relación de asociación: es una relación estructural que describe una conexión entre objetos. Dos o más clases pueden estar asociadas de diferentes modos

-Relación de asociación agregación: si una clase posee una propiedad de otra clase y al ser instanciada recibe una copia de dicho objeto como parámetro, decimos que lo agrega a la clase. Con esto podemos expresar que el objeto agregado persiste si se encuentra el fin de ámbito del objeto que lo agrego.



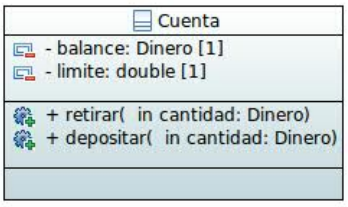
Aunque las asociaciones suelen ser bidireccionales (se pueden recorrer en ambos sentidos), en ocasiones es deseable hacerlas unidireccionales (restringir su navegación en un único sentido).



Gráficamente, cuando la asociación es unidireccional, la línea termina en una punta de flecha que indica el sentido de la asociación

El ultimo ejemplo se interpretaría de la siguiente forma, donde el atributo privado “balance” recibe como valor de inicialización un objeto de tipo “Dinero”.

```
public class Cuenta {
    private Dinero balance;
    private double limite;
    public void retirar (Dinero cantidad){
        balance -= cantidad;
    }
    public void depositar (Dinero cantidad){
```



```
            balance += cantidad;
        }
    }
```

La multiplicidad (o cardinalidad) de una asociación determina cuántos objetos de cada tipo intervienen en la relación.

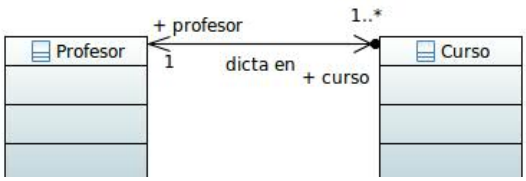
El número de instancias de una clase que se relacionan con UNA instancia de la otra clase.

- Cada asociación tiene dos multiplicidades (una para cada extremo de la relación).
- Para especificar la multiplicidad de una asociación hay que indicar la multiplicidad mínima y la multiplicidad máxima (mínima..máxima)

Multiplicidad	Significado
1	Uno y sólo uno
0..1	Cero o uno
N..MM	Desde N hasta M
*	Cero o varios
0..*	Cero o varios
1..*	Al menos uno o varios

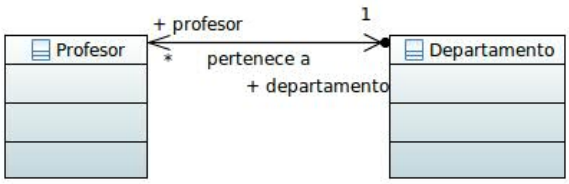
Un curso es dictado por un único profesor.

Un profesor puede dictar uno o varios cursos.



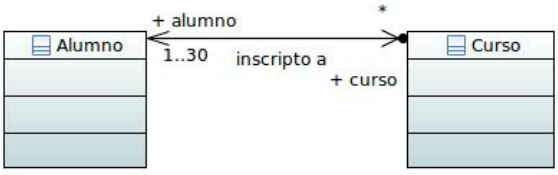
Todo profesor pertenece a un departamento.

Un departamento tiene varios profesores.



Un alumno puede estar inscripto a ningún o varios cursos.

Un curso puede tener entre 1 y 30 alumnos inscriptos.



-**Relación de asociación agregación:** La agregación es un tipo de asociación que indica que una clase es parte de otra clase (composición débil). Los componentes pueden ser compartidos por varios compuestos (de la misma asociación de agregación o de varias asociaciones de agregación distintas). La destrucción del compuesto no conlleva la destrucción de los componentes. Habitualmente se da con mayor frecuencia que la composición.

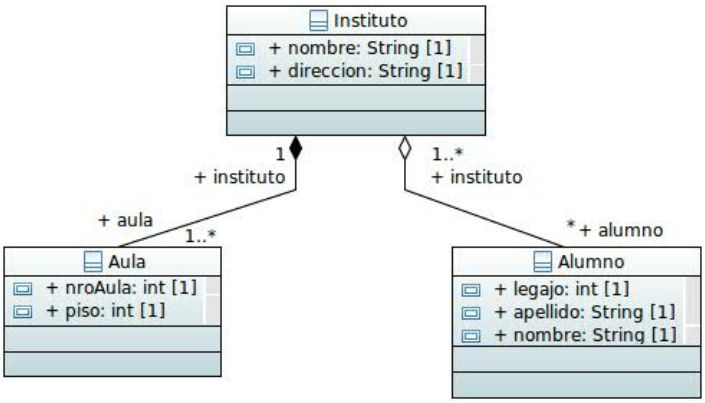
La agregación se representa en UML mediante un diamante de color blanco colocado en el extremo en el que está la clase que representa el “todo”.

-**Relación de asociación composición:** La Composición es una forma fuerte de composición donde la vida de la clase contenida debe coincidir con la vida de la clase contenedor. Los componentes constituyen una parte del objeto compuesto. De esta forma, los componentes no pueden ser compartidos por varios objetos compuestos. La supresión del objeto compuesto conlleva la supresión de los componentes.

El símbolo de composición es un diamante de color negro colocado en el extremo en el que está la clase que representa el "todo" (Compuesto).

Un instituto esta compuesto por una o varias aulas. Un aula es parte de un instituto y si este desapareciera también lo haría el aula.

Un instituto es concurrido por alumnos. Un alumno concurre a uno o varios institutos, para ser alumno, y no dejara de existir por mas que el instituto cierre.



Característica	Agregación	Composición
Varias asociaciones comparten los componentes	Si	No
Destrucción de los componentes al destruir el compuesto	No	Si
Cardinalidad a nivel de compuesto	Cualquiera	0..1 o 1
Representación	Rombo transparente	Rombo negro

BIBLIOGRAFÍA

- Berzal Galiano, Fernando, "Apuntes de programación orientada a objetos en JAVA: Fundamentos de programación y principios de diseño" (2006)
- Bonaparte, Ubaldo José, “Proyectos UML Diagramas de clases y aplicaciones JAVA en NetBeans” (edUTecNe 2012)
- Ceballos, Fco. Javier, “JAVA 2 Curso de programación” 4ta Ed. (Ra-Ma 2010)
- Deitel, Paul y Deitel, Harvey, “JAVA Cómo programar” 9na Ed. (Pearson 2012)
- Eckel, Bruce, “Piensa en JAVA” 4ta Ed. (Prentice-Hall 2007)
- Kuhn, Mónica, “Apuntes de Programación II” INSPT/UTN (2014)
- Otero, Abraham, “Tutorial básico de JAVA” 3ra Ed. (JAVAhispano.org 2007)
- Pérez, Gustavo Guillermo, “Aprendiendo JAVA y Programación Orientada a Objetos” (2008)
- Sánchez, Jorge, “JAVA 2” (2004)

LICENCIA

Matías E. García

Prof. & Tec. en Informática Aplicada
www.profmatiasgarcia.com.ar
info@profmatiasgarcia.com.ar