**TEMA 6: ORIENTACIÓN A OBJETOS AVANZADO**

**1. Conceptos.**

**a) Encapsulamiento**

Al empaquetamiento de las variables de un objeto con la protección de sus métodos se le llama ***encapsulamiento***. Encapsulamiento es el ocultamiento del estado, es decir, de los datos miembro de un objeto, de manera que solo se pueda cambiar mediante las operaciones definidas para ese objeto. El encapsulamiento protege a los datos asociados de un objeto contra su modificación por quien no tenga derecho a acceder a ellos, eliminando efectos secundarios e interacciones.

De esta forma, el usuario de la clase puede obviar la implementación de los métodos y propiedades para concentrarse sólo en cómo usarlos. Por otro lado se evita que el usuario pueda cambiar su estado de maneras imprevistas e incontroladas.

El encapsulamiento de variables y métodos en un componente de software provee de dos principales ***beneficios*** a los desarrolladores de software:

* **Modularidad**. Esto es, el código fuente de un objeto puede ser escrito, así como darle mantenimiento, independientemente del código fuente de otros objetos.

La *modularidad* es la capacidad que tiene un [sistema](http://es.wikipedia.org/wiki/Sistema) de ser estudiado, visto o entendido como la unión de varias partes que interactúan entre sí y que trabajan para alcanzar un objetivo común, realizando cada una de ellas una tarea necesaria para la consecución de dicho objetivo. Cada una de esas partes en que se encuentre dividido el sistema recibe el nombre de [**módulo**](http://es.wikipedia.org/wiki/M%C3%B3dulo_(programaci%C3%B3n)).

* **Ocultamiento de la información.** Es decir, un objeto tiene una "interfaz publica" que otros objetos pueden utilizar para comunicarse con él. Pero el objeto puede mantener información y métodos privados que pueden ser cambiados en cualquier momento sin afectar a los otros objetos que dependan de ello.

Los **objetos** tienen la ventaja de la modularidad y el ocultamiento de la información. Las **clases** proveen el beneficio de la **reutilización**. Los programadores de software utilizan la misma clase, y por lo tanto el mismo código, una y otra vez para crear muchos objetos.

En las implantaciones orientadas a objetos se percibe un objeto como un paquete de datos y procedimientos que se pueden llevar a cabo con estos datos. Esto encapsula los datos y los procedimientos.

Los **atributos** se relacionan al objeto o instancia y los métodos a la clase. ¿Por qué se hace así? Los atributos son variables comunes en cada objeto de una clase y cada uno de ellos puede tener un valor asociado, para cada variable, diferente al que tienen para esa misma variable los demás objetos.

Los **métodos**, por su parte, pertenecen a la clase y no se almacenan en cada objeto, puesto que sería un desperdicio almacenar el mismo procedimiento varias veces y ello va contra el principio de reutilización de código.

**b) Herencia**

La *herencia* es un mecanismo que permite la definición de una clase a partir de la definición de otra ya existente. La herencia permite compartir automáticamente métodos y datos entre clases, subclases y objetos.

**La herencia está fuertemente ligada a la reutilización del código en la POO.**

En Java, como en otros lenguajes de programación orientados a objetos, las clases pueden derivar desde otras clases. La clase derivada (la clase que proviene de otra clase) se llama **subclase**. La clase de la que está derivada se denomina **superclase**.

De hecho en Java, todas las clases deben derivar de alguna clase. Lo que nos lleva a la cuestión ¿Dónde empieza todo esto? La clase más alta, la clase de la que todas las demás descienden, es la clase **Object**, definida en java.lang. Object es la raíz de la herencia de todas las clases.

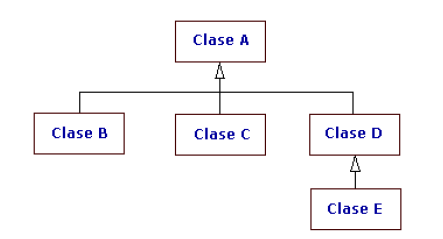
Hay dos **tipos de herencia**: Herencia Simple y Herencia Múltiple. La primera indica que se pueden definir nuevas clases solamente a partir de una clase inicial mientras que la segunda indica que se pueden definir nuevas clases a partir de dos o más clases iniciales. **Java solo permite herencia simple**.

**c) Superclase y Subclases**

El concepto de herencia conduce a una estructura jerárquica de clases o estructura de árbol, lo cual significa que en la POO todas las relaciones entre clases deben ajustarse a dicha estructura.

**En esta estructura jerárquica, cada clase tiene sólo una clase padre**. La clase padre de cualquier clase es conocida como su **superclase**. La clase hija de una superclase es llamada una **subclase**.

* Una superclase puede tener cualquier número de subclases.
* Una subclase puede tener solo una superclase.



* A es la superclase de B, C y D.
* D es la superclase de E.
* B, C y D son subclases de A.
* E es una subclase de D.

**d) Polimorfismo**

Otro concepto de la POO es el polimorfismo. Un objeto solamente tiene una forma (la que se le asigna cuando se construye ese objeto) pero la referencia al objeto es polimórfica porque puede referirse a objetos de diferentes clases (es decir, la referencia toma múltiples formas). Para que esto sea posible **debe haber una relación de herencia entre esas clases**. Por ejemplo, considerando la figura anterior de herencia se tiene que:

* Una referencia a un objeto de la clase B también puede ser una referencia a un objeto de la clase A.
* Una referencia a un objeto de la clase C también puede ser una referencia a un objeto de la clase A.
* Una referencia a un objeto de la clase D también puede ser una referencia a un objeto de la clase A.
* Una referencia a un objeto de la clase E también puede ser una referencia a un objeto de la clase D.
* Una referencia a un objeto de la clase E también puede ser una referencia a un objeto de la clase A.

**e) Abstracción**

Volviendo a la figura anterior de la relación de herencia entre clases, se puede pensar en una jerarquía de clases como la definición de conceptos **más abstractos en lo alto de la jerarquía** y esas ideas se convierten en algo más concreto conforme se desciende por la cadena de la superclase.

Sin embargo, las clases hijas no están limitadas al estado y conducta provistos por sus superclases; **pueden agregar variables y métodos** además de los que ya heredan de sus clases padres. Las clases hijas pueden, también, **sobrescribir los métodos** que heredan por implementaciones especializadas para esos métodos. De igual manera, no hay limitación a un solo nivel de herencia por lo que se tiene un árbol de herencia en el que se puede heredar varios niveles hacia abajo y mientras más niveles se desciende en una clase, más especializada será su conducta.

La herencia presenta los siguientes **beneficios**:

Las subclases proveen conductas especializadas sobre la base de elementos comunes provistos por la superclase. A través del uso de herencia, los programadores pueden *reutilizar el código de la superclase* muchas veces.

Los programadores pueden implementar superclases llamadas **clases abstractas** que definen conductas "genéricas". Las superclases abstractas definen, y pueden implementar parcialmente la conducta, pero parte de la clase no está definida ni implementada. Otros programadores concluirán esos detalles con subclases especializadas.

**2. Heredando clases en Java**

El concepto de herencia conduce a una estructura jerárquica de clases o estructura de árbol, lo cual significa que en la POO todas las relaciones entre clases deben ajustarse a dicha estructura.

En esta estructura jerárquica, cada clase tiene solo una clase padre. La clase padre de cualquier clase es conocida como su superclase. La clase hija de una superclase es llamada una subclase.

De manera automática, una subclase hereda los atributos y métodos de su superclase.

Además, una subclase puede agregar nueva funcionalidad (atributos y métodos) que la

superclase no tenía.

Los constructores no son heredados por las subclases.

Para crear una subclase, se incluye la palabra clave **extends** en la declaración de la clase.

|  |
| --- |
| class nombreSubclase **extends** nombreSuperclase {  ................  } |

En Java, la clase padre de todas las clases es la clase **Object** y cuando una clase no tiene una superclase explícita, su superclase es Object.

**3. Sobrecarga de métodos y de constructores**

Cada método tiene una **"firma"** por así decirlo, que son su nombre, el tipo y número de sus parámetros. Existe una característica para tener dos métodos (o constructores) con el mismo nombre. Esta característica se denomina **sobrecarga de métodos**.

La **cabecera de un método** es la combinación del tipo de dato que devuelve, su nombre y su lista de argumentos.

Java diferencia los métodos sobrecargados con base en el número y tipo de argumentos que tiene el método y no por el tipo que devuelve.

También existe la **sobrecarga de constructores**. Cuando en una clase existen varios constructores, se dice que hay sobrecarga de constructores.

Veamos un ejemplo en el caso de una clase llamada Publicacion y cómo los constructores nos dan un ejemplo de sobrecarga de métodos:

****

El compilador resolverá qué constructor debe ejecutar en cada momento en función del número de parámetros y su tipo. Si se llama al constructor sin parámetros se ejecutará el primer constructor y en caso de hacerlo con dos parámetros String, se ejecutará el segundo.

|  |  |  |
| --- | --- | --- |
|  | |  | | --- | | **Nota:** El concepto de **sobrecarga de métodos** se puede aplicar siempre que los parámetros sean diferentes, bien por su tipo, bien porque el número de parámetros de un método u otro es diferente. Hay que tener cuidado con los tipos: int, byte y short ya que aunque son tipos diferentes, si hacemos la llamada a un método con un número entero no sabría a cuál de los métodos llamar, ya que un entero puede ser considerado de las tres formas. Con double y float no pasa, porque acordaos que hemos de forzar a que Java entienda un decimal como float, por defecto lo entiende como double. | |

Otros ejemplos:

int calculaSuma(int x, int y, int z){

...

}

int calculaSuma(double x, double y, double z){

...

}

/\* **Error**: estos métodos no están sobrecargados \*/

int calculaSuma(int x, int y, int z){

...

}

double calculaSuma(int x, int y, int z){

...

}

**Ej**:

package usuario;

//class Usuario{ tiene que ser pública para que sea visible

public class Usuario{

String nombre;

int edad;

String direccion;

//El constructor está sobrecargado

public Usuario(){

nombre = null;

edad = 0;

direccion = null;

}

public Usuario(String nombre, int edad, String direccion){

this.nombre = nombre;

this.edad = edad;

this.direccion = direccion;

}

public Usuario(Usuario usr){

nombre = usr.getNombre();

edad = usr.getEdad();

direccion = usr.getDireccion();

}

public void setNombre(String n){

nombre = n;

}

public String getNombre(){

return nombre;

}

//El metodo setEdad() está sobrecargado

public void setEdad(int e){

edad = e;

}

public void setEdad(double e){

edad = (int)e;

}

public int getEdad(){

return edad;

}

public void setDireccion(String d){

direccion = d;

}

public String getDireccion(){

return direccion;

}

// Añadir un método para imprimir un usuario para eliminarlo del main

}

package prueba;

import usuario.Usuario;

public class Main{

void imprimeUsuario(Usuario usr){

System.out.println("\nNombre: " + usr.getNombre());

System.out.println("Edad: " + usr.getEdad());

System.out.println("Direccion: " + usr.getDireccion());

}

public static void main(String args[]){

Main prog = new Main();

//La otra opción sería que imprimeUsuario fuera static

Usuario usr1, usr2;

usr1 = new Usuario();

prog.imprimeUsuario(usr1);

usr2 = new Usuario ("Eduardo",24,"Mi direccion");

prog.imprimeUsuario(usr2);

usr1 = new Usuario(usr2);

usr1.setEdad(50);

usr2.setEdad(30.45);

prog.imprimeUsuario(usr1);

prog.imprimeUsuario(usr2);

}

}

**4. Sobreescritura de métodos**

Una subclase hereda todos los métodos de su superclase que son accesibles a dicha subclase a menos que la subclase **sobreescriba los métodos**.

Una **subclase sobreescribe un método de su superclase**, cuando define un método con las mismas características (nombre, número y tipo de argumentos y retorno) que el método de la superclase.

Las subclases emplean la sobreescritura de métodos la mayoría de las veces para agregar o modificar la funcionalidad del método heredado de la clase padre.

Ej:

class ClaseA{

void miMetodo(int var1, int var2){ ... }

String miOtroMetodo(){ ... }

}

class ClaseB extends ClaseA{

//Estos métodos sobreescriben a los métodos de la clase padre

void miMetodo(int var1 ,int var2){ ... }

String miOtroMetodo(){ ... }

}

**5. Clases abstractas**

Una clase que declara la existencia de métodos, pero no la implementación de dichos métodos, se considera una clase abstracta.

Una clase abstracta puede contener métodos no-abstractos.

Para declarar una clase o un método como abstractos, se utiliza la palabra reservada **abstract**.

abstract class Drawing{

abstract void miMetodo(int var1, int var2);

String miOtroMetodo(){ ... }

}

Una clase abstracta no se puede instanciar pero sí se puede heredar y las clases hijas serán las encargadas de agregar la funcionalidad a los métodos abstractos. Si no lo hacen así, las clases hijas deben ser también abstractas.

Ej:

Public abstract class FiguraGeometrica {

. . .

abstract double Area();

. . .

}

class Circulo extends FiguraGeometrica {

double radio;

double Area() {

return 3.14\*radio\*radio;

}

}

Se pueden crear referencias a clases abstractas como cualquier otra. No hay ningún problema en poner:

FiguraGeometrica figura;

Sin embargo una clase abstracta no se puede instanciar, es decir, no se pueden crear objetos de una clase abstracta. El compilador producirá un error si se intenta:

FiguraGeometrica figura = new FiguraGeometrica();

Esto es coherente dado que una clase abstracta no tiene completa su implementación y encaja bien con la idea de que algo abstracto no puede materializarse.

Sin embargo, utilizando el up-casting\* se puede escribir:

FiguraGeometrica figura = new Circulo(. . .);

figura.Area();

(\*) Es la operación en que un objeto de una clase derivada se asigna a una referencia cuyo tipo es alguna de las superclases. Cuando se realiza este tipo de operaciones, hay que tener cuidado porque la referencia es a los miembros de la clase hija, aunque la referencia apunte a un objeto de este tipo.

**6. Interfaces**

Una interface es una variante de una clase abstracta con la condición de que todos sus métodos deben ser abstractos (lo serán aunque no se especifique). Si la interface va a tener atributos, estos deben llevar las palabras reservadas **static final** y con un valor inicial ya que funcionan como constantes.

Ej:

interface Nomina{

public static final String EMPRESA = "Patito, S. A.";

public void detalleDeEmpleado(Nomina obj);

}

Una clase implementa una o más interfaces (separadas con comas ",") con la palabra reservada implements. Con el uso de interfaces se puede "simular" la herencia múltiple que Java no soporta.

class Empleado implements Nomina{

...

}

La clase que implementa una interface tiene dos opciones:

1) Implementar todos los métodos de la interface.

2) Implementar solo algunos de los métodos de la interface pero esa clase debe ser una clase abstracta.

***(Ver ampliación Interfaces)***

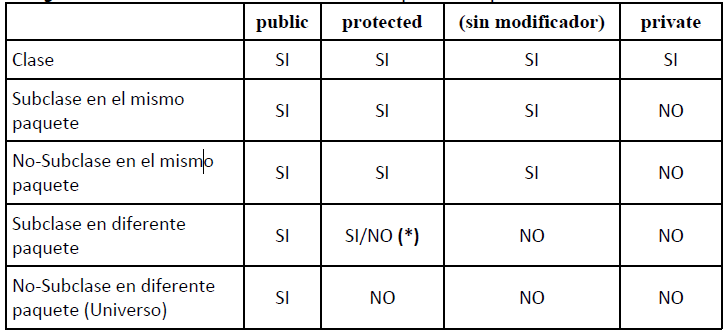
**7. Control de acceso a miembros de una clase**

Los modificadores más importantes son los que permiten controlar la visibilidad y acceso a los métodos y variables que están dentro de una clase.

Uno de los beneficios de las clases, es que pueden proteger a sus variables y métodos (tanto de instancia como de clase) del acceso desde otras clases.

Java soporta cuatro niveles de acceso a variables y métodos. En orden, del más público al menos público son: público (public), protegido (protected), sin modificador (también conocido como package) y privado (private).

La siguiente tabla muestra el **nivel de acceso permitido por cada modificador**:



**(\*)** Los miembros (variables y métodos) de clase (static) sí son visibles. Los miembros de instancia no son visibles.

Como se observa en la tabla anterior:

* Una clase puede ver/usar dentro de ella misma todo tipo de variables y métodos (desde los public hasta los private);
* Las demás clases del mismo paquete (ya sean subclases o no) tienen acceso a los miembros public, protected y sin modificador.
* Las subclases de otros paquetes pueden ver los miembros public y a los miembros protected, estos últimos siempre que sean static, ya que de no ser así no serán visibles en la subclase.
* El resto del universo de clases (que no sean ni del mismo paquete ni subclases) pueden ver solo los miembros public.