PROGRAMACIÓN ORIENTADA A OBJETOS

1º. Conceptos.

a) Encapsulamiento

Al empaquetamiento de las variables de un objeto con la protección de sus métodos se le llama *encapsulamiento*. Típicamente, el encapsulamiento es utilizado para esconder detalles de la puesta en práctica no importantes de otros objetos. Entonces, los detalles de la puesta en práctica pueden cambiar en cualquier tiempo sin afectar otras partes del programa.

El encapsulamiento de variables y métodos en un componente de software ordenado es, todavía, una simple idea poderosa que provee 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. Así mismo, un objeto puede ser transferido alrededor del sistema sin alterar su estado y conducta.
- *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 tiempo sin afectar a los otros objetos que dependan de ello.

Los objetos proveen el beneficio 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. La realidad es diferente: 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.

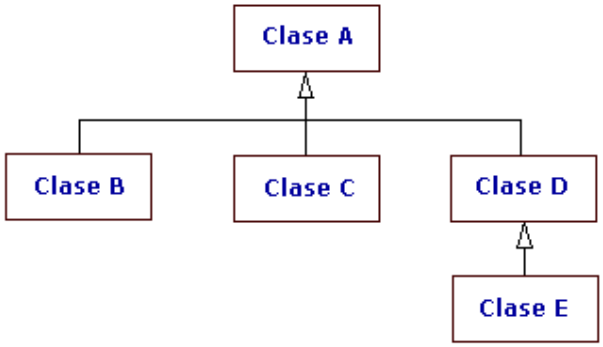
La herencia está fuertemente ligada a la reutilización del código en la OOP. Esto es, el código de cualquiera de las clases puede ser utilizado sin más que crear una clase derivada de ella, o bien una subclase.
Hay dos tipos de herencia: Herencia Simple y Herencia Múltiple. La primera indica que se pueden 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 sólo 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 OOP 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 sólo 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 OOP es el polimorfismo. Un objeto solamente tiene una forma (la que se le asigna cuando se construye ese objeto) pero la referencia a 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 clase como la definición de conceptos demasiado 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, sobreescribir los métodos que heredan por implementaciones especializadas para esos métodos. De igual manera, no hay limitación a un sólo 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 descienda 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 gran parte de la clase no está definida ni implementada. Otros programadores concluirán esos detalles con subclases especializadas.

2º. 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 OOP 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.

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.

3º. Sobrecarga de métodos y de constructores

La firma de un método es la combinación del tipo de dato que regresa, su nombre y su lista de argumentos.

La sobrecarga de métodos es la creación de varios métodos con el mismo nombre pero con diferentes firmas y definiciones. Java utiliza el número y tipo de argumentos

para seleccionar cuál definición de método ejecutar.

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.

Ej:

```
/* Métodos sobrecargados */

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:

```
class Usuario4{
     String nombre;
     int edad;
     String direccion;

 //El constructor está sobrecargado

     Usuario4(){
        nombre = null;
        edad = 0;
        direccion = null;
     }



     Usuario4(String nombre, int edad, String direccion){
        this.nombre = nombre;
        this.edad = edad;
        this.direccion = direccion;
     }

     Usuario4(Usuario4 usr){
        nombre = usr.getNombre();
        edad = usr.getEdad();
        direccion = usr.getDireccion();
     }
```

```
      void setNombre(String n){
         nombre = n;
      }

      String getNombre(){
         return nombre;
      }

 //El metodo setEdad() está sobrecargado

      void setEdad(int e){
         edad = e;
      }

      void setEdad(double e){
         edad = (int)e;
      }

      int getEdad(){
         return edad;
      }

      void setDireccion(String d){
         direccion = d;
      }

      String getDireccion(){
         return direccion;
      }
}

class ProgUsuario4{
   void imprimeUsuario(Usuario4 usr){
     System.out.println("\nNombre: " + usr.nombre);
     System.out.println("Edad: " + usr.getEdad());
     System.out.println("Direccion: " + usr.getDireccion());
   }

   public static void main(String args[]){
      ProgUsuario4 prog = new ProgUsuario4();
      Usuario4 usr1,usr2;


      usr1 = new Usuario4();
      prog.imprimeUsuario(usr1);

      usr2 = new Usuario4("Eduardo",24,"Mi direccion");
      prog.imprimeUsuario(usr2);

      usr1 = new Usuario4(usr2);

      usr1.setEdad(50);
      usr2.setEdad(30.45);

      prog.imprimeUsuario(usr1);
      prog.imprimeUsuario(usr2);
   }
}
```

4º. 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) 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(){ ... }
}
```

5º. 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 pero al menos uno de los métodos debe ser declarado abstracto.

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 si 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:

```
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.

6º. Interfaces

Una interface es una variante de una clase abstracta con la condición de que todos sus métodos deben ser asbtractos. Si la interface va a tener atributos, éstos 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 sólo algunos de los métodos de la interface pero esa clase debe ser una clase abstracta.

7º. 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:

	public	**protected**	**(sin modificador)**	**private**
Clase	SI	SI	SI	SI
Subclase en el mismo paquete	SI	SI	SI	NO
No-Subclase en el mismo paquete	SI	SI	SI	NO
Subclase en diferente paquete	SI	SI/NO **(*)**	NO	NO
No-Subclase en diferente paquete (Universo)	SI	NO	NO	NO

(*) Los miembros (variables y metodos) de clase (static) si son visibles. Los miembros de instancia no son visibles.

Como se observa de la tabla anterior, una clase se ve a ella misma todo tipo de variables y métodos (desde los public hasta los private); las demas clases del mismo paquete (ya sean subclases o no) tienen acceso a los miembros desde los public hasta los sin-modificador. Las subclases de otros paquetes pueden ver los miembros public y a los miembros protected, éstos últimos siempre que sean static ya de no ser así no serán visibles en la subclase (Esto se explica en la siguiente página). El resto del universo de clases (que no sean ni del mismo paquete ni subclases) pueden ver sólo los miembros public.