



# UNIDAD DE TRABAJO 6

UTILIZACIÓN AVANZADA DE CLASES

## Contenido

|  |    |
|--|----|
| Qué vimos en la unidad anterior.....   | 3  |
| Encapsulamiento .....                  | 3  |
| Polimorfismo .....                     | 3  |
| Herencia .....                         | 4  |
| Relaciones entre clases .....          | 5  |
| Qué son las asociaciones.....          | 5  |
| UML .....                              | 5  |
| Tipos de asociaciones.....             | 7  |
| Agregación.....                        | 7  |
| Composición .....                      | 7  |
| Implementación .....                   | 8  |
| La generalización (herencia) .....     | 8  |
| Generalización y especialización ..... | 9  |
| La herencia .....                      | 10 |
| This .....                             | 11 |
| Super .....                            | 11 |
| Tipos de herencia .....                | 12 |
| Implementación de la herencia.....     | 12 |
| Clases abstractas .....                | 15 |
| El polimorfismo .....                  | 16 |
| Interfaces.....                        | 19 |
| Definición de interfaz .....           | 20 |
| Implementación de una interfaz .....   | 20 |
| Interfaces y polimorfismo .....        | 22 |
| Interfaces en el Java SE .....         | 22 |

## Qué vimos en la unidad anterior

En la unidad de trabajo anterior se expuso el concepto de Clase y Objeto, indicando sus componentes; así como, la enumeración de las propiedades de la Programación Orientada a Objetos: encapsulamiento, polimorfismo y herencia.

En esta unidad de trabajo, se profundizará más sobre estas propiedades. Recordemos qué significaban estos conceptos

### Encapsulamiento

Una de las propiedades fundamentales de la POO, que nos define que los objetos están encapsulados. Esto quiere decir, que los atributos de los objetos no deben ser accesibles desde el exterior. Para ello, se utilizarán métodos que nos permitan manipularlos.

En general, se suelen definir los atributos como privados y se accederá a ellos con los métodos get/set.

El acceso a los miembros de un objeto es: nombre\_objeto.nombre\_método(parámetros)

### Polimorfismo

Cada método de una clase puede tener varias definiciones distintas. Es decir, para una misma acción podemos tener comportamientos diferentes porque existen diferentes implementaciones

El polimorfismo admite sobrecargar los métodos. Esto significa crear distintas variantes del mismo método.

Ejemplo:

```
class Matemáticas{
    public double suma(double x, double y) {
        return x+y;
    }
    public double suma(double x, double y, double z){
        return x+y+z;
    }
    public double suma(double[] array){
        double total =0;
        for(int i=0; i<array.length;i++){
            total+=array[i];
        }
        return total;
    }
}
```

La clase matemáticas posee tres versiones del método suma: una versión que suma dos números double, otra que suma tres y la última que suma todos los miembros de un array de doubles. Desde el código se puede utilizar cualquiera de las tres versiones según convenga.

Por tanto, podemos implementar métodos sobrecargados en función de las operaciones que deban realizar los objetos de la clase. Uno de los métodos que suele estar sobrecargado es el método constructor, que nos permitirá crear objetos con diferente número de propiedades.

## Herencia

Una clase puede ser subclase o clase hija de otra clase, de tal manera que hereda sus propiedades y comportamientos. Este principio, es fundamental ya que en aplicaciones más complejas deberemos diseñar una jerarquía de clases.

**Permite crear nuevas clases que heredan características presentas en clases anteriores.** Esto facilita enormemente el trabajo porque ha permitido crear clases estándar para todos los programadores y a partir de ellas crear nuestras propias clases personales. Esto es más cómodo que tener que crear nuestras clases desde cero.

Para que una clase herede las características de otra hay que utilizar la palabra clave **extends** tras el nombre de la clase. A esta palabra le sigue el nombre de la clase cuyas características se heredarán. Sólo se puede tener herencia de una clase (a la clase de la que se hereda se la llama superclase y a la clase heredada se la llama subclase).

Los especificadores determinan el alcance de la visibilidad del elemento al que se refieren. Referidos por ejemplo a un método, pueden hacer que el método sea visible sólo para la clase que lo utiliza (private), para éstas y las heredadas (protected), para todas las clases del mismo paquete (friendly) o para cualquier clase del tipo que sea (public).

| zona                                    | private<br>(privado) | sin<br>modificador<br>(friendly) | protected<br>(protegido) | public<br>(público) |
|---|----------------------|----------------------------------|--------------------------|---------------------|
| Misma clase                             | X                    | X                                | X                        | X                   |
| Subclase en el mismo paquete            |                      | X                                | X                        | X                   |
| Clase (no subclase) en el mismo paquete |                      | X                                |                          | X                   |
| Subclase en otro paquete                |                      |                                  | X                        | X                   |
| No subclase en otro paquete             |                      |                                  |                          | X                   |

Nota: En los diagramas de clases UML (la notación más popular para representar clases), las propiedades y métodos privados anteponen al nombre el signo -, los públicos el signo +, los protegidos el signo # y los friendly no colocan ningún signo.

Ejemplo:

```
class Persona {
    public String nombre; //Se puede acceder desde
    cualquier clase
    private int contraseña; //Sólo se puede acceder desde
    la
        //clase Persona
    protected String dirección; //Acceden a esta propiedad
        //esta clase y sus descendientes
```

## Relaciones entre clases

Se puede entender que el diseño de una aplicación es prácticamente el diseño de un conjunto de clases, ya que una aplicación es un conjunto de objetos que se relacionan. Por ello, en el diagrama de clases se deben indicar la relación que hay entre las clases.

### Qué son las asociaciones

Como los objetos no existen aisladamente, es muy importante estudiar las relaciones que existen entre sí. Aquí estudiamos las relaciones básicas. La decisión para establecer una u otra relación entre dos objetos no es una tarea de programación, sino de diseño orientado a objetos y depende básicamente de la experiencia del diseñador

**Las asociaciones son relaciones entre clases.** Es decir, marcan una comunicación o colaboración entre clases. Dos clases tienen una asociación si:







- ◆ Un objeto de una clase envía un mensaje a un objeto de la otra clase. Enviar un mensaje es utilizar alguno de sus métodos o propiedades para que el objeto realice una determinada labor.
- ◆ Una clase tiene propiedades cuyos valores son objetos o colecciones de objetos de otra clase
- ◆ Un objeto de una clase recibe como parámetros de un método objetos de otra clase.

## UML

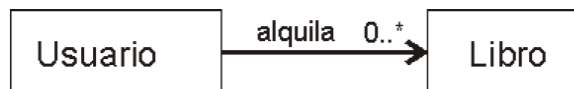
UML es un lenguaje gráfico para visualizar, especificarlo, construir y documentar un sistema. En el módulo de Entornos de Desarrollo aprenderás más sobre este lenguaje. Se utiliza mucho en la Programación Orientada a Objetos para modelar las clases, por este motivo se añade este apartado para aclarar algunos conceptos o representaciones que pueden aparecer más adelante.

Se utiliza UML para representar las relaciones entre clases. En la siguiente figura se muestra un glosario de notación UML para elaborar diagramas de clases.

Glosario de notaciones para diagramas de clases

| Símbolo   | Significado                                |
|---|--|
| <div> <div>Clases</div> <div>Atributos</div> <div>Métodos</div> </div>            | Clase                                      |
| <div> <div>&lt;&lt;interface&gt;&gt;</div> <div>operaciones</div> </div>          | interfaz                                   |
|  | Generalización                             |
|  | Implementación de interfaz                 |
|  | Asociación<br>Asociación con multiplicidad |
|  | Asociación con navegabilidad               |
|  | Agregación                                 |
|  | Composición                                |

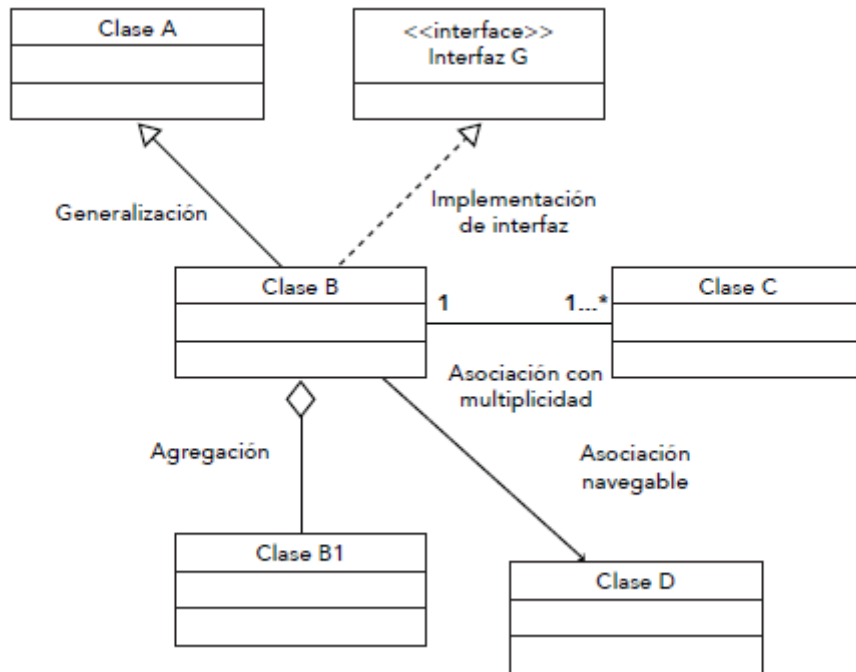
En UML las asociaciones se representan con una línea entre las dos clases relacionadas, encima de la cual se indica el nombre de la asociación y una flecha para indicar el sentido de la asociación. Ejemplo:



Como se observa en el ejemplo la dirección de la flecha es la que indica que es el usuario el que alquila los libros. Los números indican que cada usuario puede alquilar de cero a más (el asterisco significa muchos) libros. Esos números se denominan cardinalidad, e indican con cuántos objetos de la clase se pueden relacionar cada objeto de la clase que está en la base de la flecha. Puede ser:

- **0..1.** Significa que se relaciona con uno o ningún objeto de la otra clase.
- **0..\*.** Se relaciona con cero, uno o más objetos
- **1..\*.** Se relaciona al menos con uno, pero se puede relacionar con más
- **un número concreto.** Se puede indicar un número concreto (como 3 por ejemplo) para indicar que se relaciona exactamente con ese número de objetos, ni menos, ni más.

### Ejemplo de construcción de un diagrama de clases



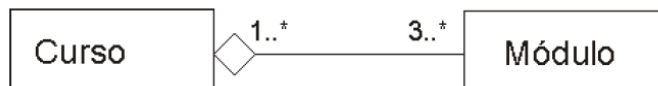
### Tipos de asociaciones

Distinguimos dos tipos de asociaciones: la agregación y la composición. Definen asociaciones del tipo *es parte de* o *se compone de*.

#### Agregación

Indica que un elemento es parte de otro. Indica una relación en definitiva de composición.

Ejemplo: la clase **Curso** tendría una relación de composición con la clase **Módulo**. Cada curso se compone de tres o más módulos. Cada módulo se relaciona con uno o más cursos.



#### Composición

La composición indica una agregación fuerte, de hecho significa que una clase consta de objetos de otra clase para funcionar. La diferencia es que cada objeto que compone el objeto grande no puede ser parte de otro objeto, es decir pertenece de forma única a uno.

La existencia del objeto al otro lado del diamante está supeditada al objeto principal y esa es la diferencia con la agregación.

Ejemplo: en este caso se refleja que un edificio consta de pisos. De hecho con ello lo que se indica es que un piso sólo puede estar en un edificio. La existencia del piso está ligada a la del edificio.



### Implementación

La implementación en Java de clases con relaciones de agregación y composición es similar. Pero hay un matiz importante. Puesto que en la composición, los objetos que se usan para componer el objeto mayor tienen una existencia ligada al mismo, se deben **crear** dentro del objeto grande.

Por ejemplo (composición):

```
public class Edificio {  
    private Piso piso[];  
    public Edificio(.....){  
        piso=new Piso[x]; //composición  
        .....  
    }
```

En la composición (como se observa en el ejemplo), la existencia del piso está ligada al edificio por eso los pisos del edificio se deben de crear **dentro** de la clase **Edificio** y así cuando un objeto **Edificio** desaparezca, desaparecerán los pisos del mismo.

Eso no debe ocurrir si la relación es de agregación

En el caso de los cursos y los módulos, como los módulos no tienen esa dependencia de existencia según el diagrama, seguirán existiendo cuando el módulo desaparezca, por eso se deben declarar fuera de la clase cursos. Es decir, no habrá **new** para crear módulos en el constructor. Sería algo parecido a esto

```
public class Cursos {  
    private Módulo módulos[];  
    public Cursos(....., Módulo m[]){  
        módulos=m; //agregación  
        .....  
    }
```

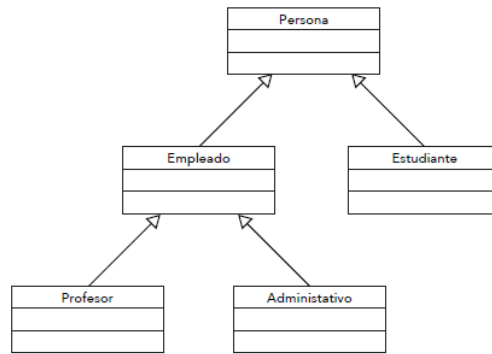
### La generalización (herencia)

La generalización es una relación entre clases en las que hay una clase padre, llamada superclase, y una o más clases hijas especializadas, a las que se les denomina *subclases*. La herencia es el mecanismo mediante el cual se implementa la relación de generalización. En la práctica, cuando se codifica un sistema, se habla de herencia en lugar de generalización.

Cuando hay herencia, la clase hija o subclase adquiere los atributos y métodos de la clase padre.

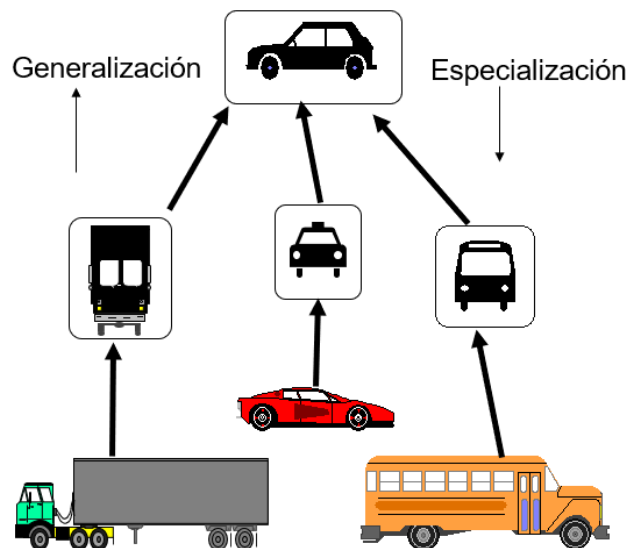
Ejemplo:





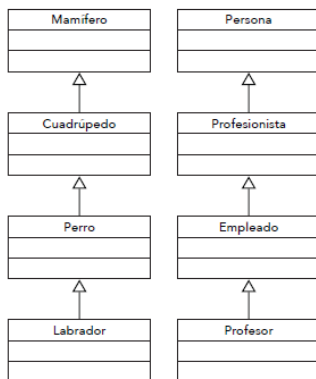
Se emplea habitualmente los términos madre para referirnos a una superclase e hija para una subclase

### Generalización y especialización



La generalización y la especialización son dos perspectivas diferentes de la misma relación de herencia.

Con la *generalización* se buscan clases que sean de nivel superior a alguna clase en particular.  
Ejemplo:

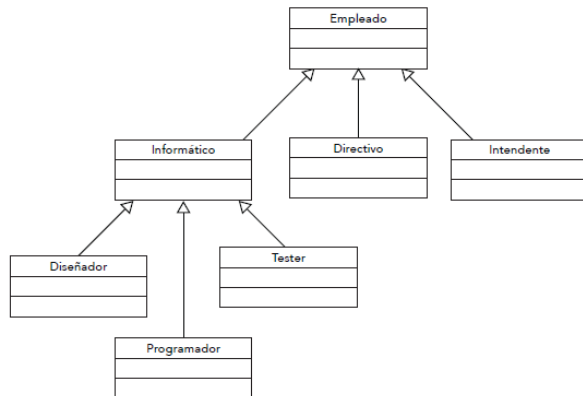


## MÓDULO PROGRAMACIÓN

Con la *generalización* se buscan propiedades comunes entre varios objetos similares que puedan agruparse para formar una nueva clase genérica. En el ejemplo, se piensa en que un *Labrador* tiene propiedades comunes con otros "objetos", como pueden ser un *Boxer* o un *Dálmata*, que se pueden agrupar en una superclase llamada *Perro*.

A la contraparte de la *generalización* se le llama *especialización* y consiste en encontrar subclases de nuestra clase actual, en las cuales se detallan las propiedades particulares de cada una de ellas. Una subclase representa la *especialización* de la clase superior (superclase).

### Ejemplo especialización

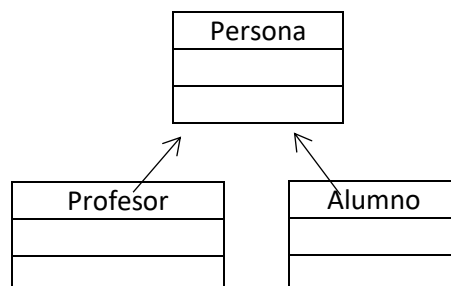


Todos los *Empleados* tienen atributos comunes, por ejemplo, su nombre, dirección, número de empleado, etcétera, y métodos comunes como `calcularQuincena()`, `calcularAntigüedad()`, etc. Sin embargo, hay cosas que caracterizan sólo a los informáticos, por ejemplo, las técnicas y herramientas de software que manejan. Así, se va construyendo una relación jerárquica de herencia entre los objetos, buscando todas las subclases que podría tener alguna clase.

### La herencia

La herencia facilita que un hijo se comporte según sus características específicas, sin tener que codificar nuevamente todos los métodos que ya contiene el padre.

Por ejemplo, la clase padre *Persona* puede tener dos hijas: *Alumno* y *Profesor*. En este caso, *Persona* es la superclase de las clases *Alumno* y *Profesor*, las cuales se dice que son subclases de *Persona*. Con el mecanismo de herencia, las clases hijas heredan el mismo código que contiene su clase padre, es decir, tienen los mismos métodos y atributos del padre.



De esta manera, implementamos la *generalización*, ya que las clases hijas no tienen que repetir ese código. Además, se agregan métodos y atributos particulares para cada subclase y así implementamos la *especialización*, ya que este código no pertenece a la clase padre.

En Java solo es posible extender de una clase (herencia simple).

### This

La referencia **this** se usa para referirse al propio objeto y se usa explícitamente para referirse tanto a las variables de instancia como a los métodos de un objeto.

Dentro de los métodos no estáticos, podemos utilizar el identificador especial **this** para referirnos a la instancia que está recibiendo el mensaje.

Algunos usos habituales son:

- Evitar conflictos de nombres con los argumentos de métodos.

```
class Persona {  
    String nombre;  
  
    public void setNombre(String nombre) {  
        this.nombre = nombre;  
    }  
    ...  
}
```

- Pasar una referencia de ese objeto a otro método.

```
Button incrementar = new Button("Incrementar");  
incrementar.addActionListener(this);
```

### Super

Super se usa para referirse a métodos de la clase padre.

```
import MiClase;  
  
public class MiNuevaClase extends MiClase {  
    public void suma_a_i( int j ) {  
        i = i + ( j/2 );  
    }  
  
    public void suma_a_i_padre( int j ) {  
        super.suma_a_i( j )  
    }  
}
```

Cuando extendemos una clase, los constructores de la subclase deben invocar algún constructor de la superclase como primera sentencia.

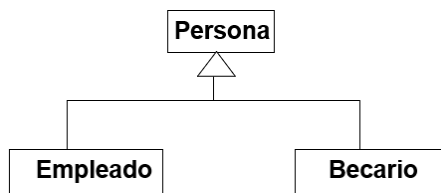
## MÓDULO PROGRAMACIÓN

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

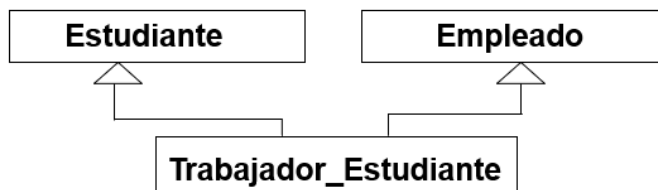
- Si no se invoca explícitamente un constructor de la superclase con `super`, debe existir un constructor sin parámetros en la superclase o no debe haber ningún constructor (el compilador genera uno por defecto sin parámetros).

### Tipos de herencia

- Simple: una clase hereda de una única super-clase.



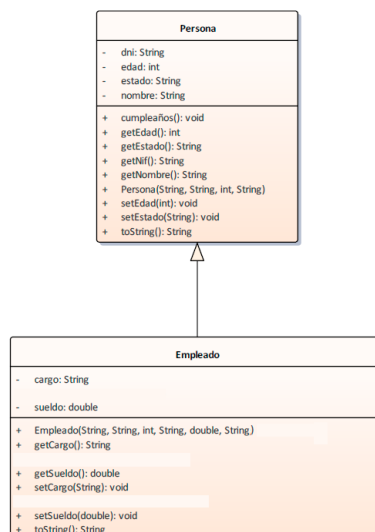
- Múltiple: una clase hereda de varias super-clases.



### Implementación de la herencia

Para implementar la herencia se utiliza la palabra reservada *extends*

A continuación, se muestra un ejemplo de herencia para el siguiente diagrama de clases:



## MÓDULO PROGRAMACIÓN

```
/**
 * Clase Persona
 * Modela objetos persona
 */
package herencia;
public class Persona {

    private String dni;
    private String nombre;
    private int edad;
    private String estado;

    /**
     * Método constructor. Crea personas con valores iniciales de
     todos sus atributos
     * @param dni Indica el dni de la persona que se está creando
     * @param nombre Indica el nombre de la persona que se está
     creando
     * @param edad Indica la edad de la persona que se está creando
     * @param estado Indica el estado civil de la persona que se está
     creando
     */
    public Persona(String dni, String nombre, int edad, String estado)
    {
        this.dni = dni;
        this.nombre = nombre;
        this.edad = edad;
        this.estado = estado;
    }

    /**
     * Método que cumple años. Se incrementa en uno la edad de la
     persona
     */
    public void cumpleaños() {
        edad++;
    }

    /**
     * Método que devuelve el DNI de la persona
     * @return dni de la persona
     */
    public String getDni() {
        return dni;
    }

    /**
     * Método que devuelve el Nombre de la persona
     * @return nombre de la persona
     */
    public String getNombre() {
        return nombre;
    }

    /**
     * Método que devuelve el edad de la persona
     * @return edad de la persona
     */
    public int getEdad() {
        return edad;
    }
}
```

```
/**
 * Método que modifica la edad de la persona
 * @param edad indica la nueva edad de la persona
 */
public void setEdad(int edad) {
    this.edad = edad;
}

/**
 * Método que devuelve el estado civil de la persona
 * @return estado civil de la persona
 */
public String getEstado() {
    return estado;
}

/**
 * Método que modifica el estado civil de la persona
 * @param estado indica el nuevo estado civil
 */
public void setEstado(String estado) {
    this.estado = estado;
}

/**
 * Método que muestra la información de las personas
 */
public String toString() {
    return "Persona{" + "dni=" + dni + ", nombre=" + nombre + ",
edad=" + edad + ", estado=" + estado + '}';
}

}

/**
 * Clase Empleado subclase de la Clase Persona
 * Modela objetos persona que trabajan
 */
package herencia;
public class Empleado extends Persona {

    private double sueldo;
    private String cargo;

    public Empleado(String dni, String nombre, int edad, String
estado,
        double sueldo, String cargo) {
        super(dni, nombre, edad, estado);
        this.sueldo = sueldo;
        this.cargo = cargo;
    }

    public String getCargo() {
        return cargo;
    }
}
```

```
public void setCargo(String cargo) {
    this.cargo = cargo;
}

public double getSueldo() {
    return sueldo;
}

public void setSueldo(double sueldo) {
    this.sueldo = sueldo;
}

public String toString() {
    return super.toString() + " # Empleado{" + "sueldo=" + sueldo
+ ", cargo=" + cargo + '}';
}
}

/*
 * Ejemplo de herencia
 * En este programa se crean dos objetos persona y empleado
 * y se trabajará con ellos para aprender la herencia
 */
package herencia;

public class Example_Herencia {
    public static void main(String[] args) {
        Persona personal = new Persona ("53022455-D", "Abel Marín",
43, "Casado");
        Empleado empleado1 = new Empleado ("5389281-S", "Rodrigo
Ruiz", 53, "Soltero", 1250.9, "Administrativo");
        System.out.println (personal.toString());
        System.out.println (empleado1.toString());

    }
}
```

### Clases abstractas

En el modelado orientado a objetos a veces sirve introducir clases de cierto nivel que incluso pueden no existir en la realidad, pero que su concepto es útil para organizar mejor la estructura de un programa. Estas clases se conocen como **clases abstractas**.

No se pueden instanciar objetos de una clase abstracta, sin embargo, si se pueden instanciar objetos de las subclases que heredan de una clase abstracta, siempre y cuando no se definan como abstractas

Cuando se organizan las clases en una jerarquía lo normal es que las clases que representan los conceptos más abstractos ocupen un lugar más alto en la misma.

Los lenguajes orientados a objetos dan la posibilidad de declarar clases que definen como se utiliza pero sin tener que implementar los métodos que posee.

En Java se realiza mediante clases abstractas que poseen métodos abstractos que implementarán las clases hijas. En el siguiente apartado se muestra un ejemplo de clase abstracta.

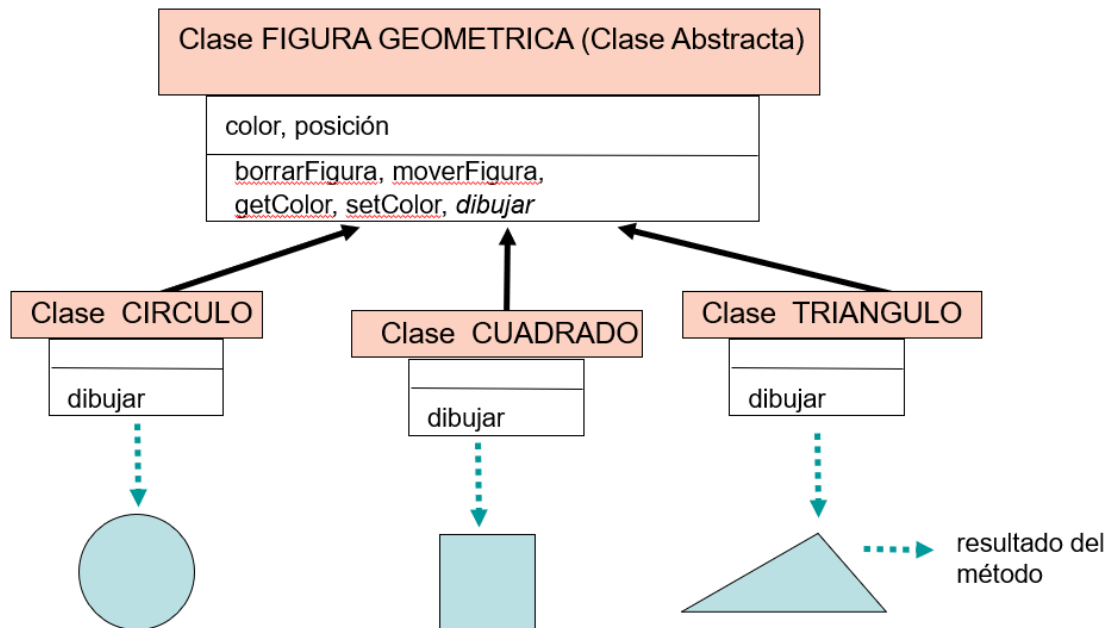
## El polimorfismo

Es la propiedad de la POO que implica que un mismo método o comportamiento puede tener distintas implementaciones. Es decir, se refiere a la capacidad de los objetos para responder de distinta forma a la misma operación.

Tenemos varias opciones para implementar el polimorfismo:

- Con la sobrecarga de métodos. Esto es cuando tenemos un mismo método pero recibe diferente tipo o número de parámetros de entrada. Esto sucede muy a menudo con el método constructor.
- Con la redefinición de métodos heredados. De tal manera, que el método de la subclase llama a su método de su súper clase y añade su propio comportamiento
- Con la implementación de métodos abstractos. Estos son comportamientos para los cuales no tenemos implementación en la clase base (la cual se debe definir como abstracta) pero sí en las subclases. Por ejemplo, en el siguiente ejemplo tenemos los métodos dibujar y área

Ejemplo de Figuras Geométricas





## MÓDULO PROGRAMACIÓN

```
public class Punto {
    private int x;
    private int y;

    public Punto(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public void setX(int x) {
        this.x = x;
    }

    public int getY() {
        return y;
    }

    public void setY(int y) {
        this.y = y;
    }

    @Override
    public String toString() {
        return " x: " + this.x + " y: " + this.y;
    }
}

public abstract class FiguraGeometrica {

    private Punto posicion;
    private String color;

    public FiguraGeometrica(Punto posicion, String color) {
        this.posicion = posicion;
        this.color = color;
    }

    public void mover(Punto nuevaPos) {
        posicion = nuevaPos;
        dibujar();
    }

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }

    public Punto getPosicion() {
        return posicion;
    }

    //métodos abstractos
}
```

## MÓDULO PROGRAMACIÓN

```
    public abstract void dibujar();
    public abstract double area();
}
public final class Circulo extends FiguraGeometrica {

    private double radio;

    public Circulo(Punto posicion, String color, double radio) {
        super(posicion, color);
        this.radio = radio;
    }

    public double getRadio() {
        return radio;
    }

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

    @Override
    public double area() {
        return 3.14 * Math.pow(radio, 2.0);
    }

    @Override
    public void dibujar() {
        System.out.println("Soy un círculo de color: " +
super.getColor() + " con posición:" + super.getPosicion().toString());
    }
}

public final class Cuadrado extends FiguraGeometrica {

    private double lado;

    public Cuadrado(Punto posicion, String color, double lado) {
        super(posicion, color);
        this.lado = lado;
    }

    public double getLado() {
        return lado;
    }

    public void setLado(double lado) {
        this.lado = lado;
    }

    @Override
    public double area() {
        return lado * lado;
    }

    @Override
    public void dibujar() {
        System.out.println("Soy un cuadrado de color: " +
super.getColor() + " con posición:" + super.getPosicion().toString());
    }
}
```

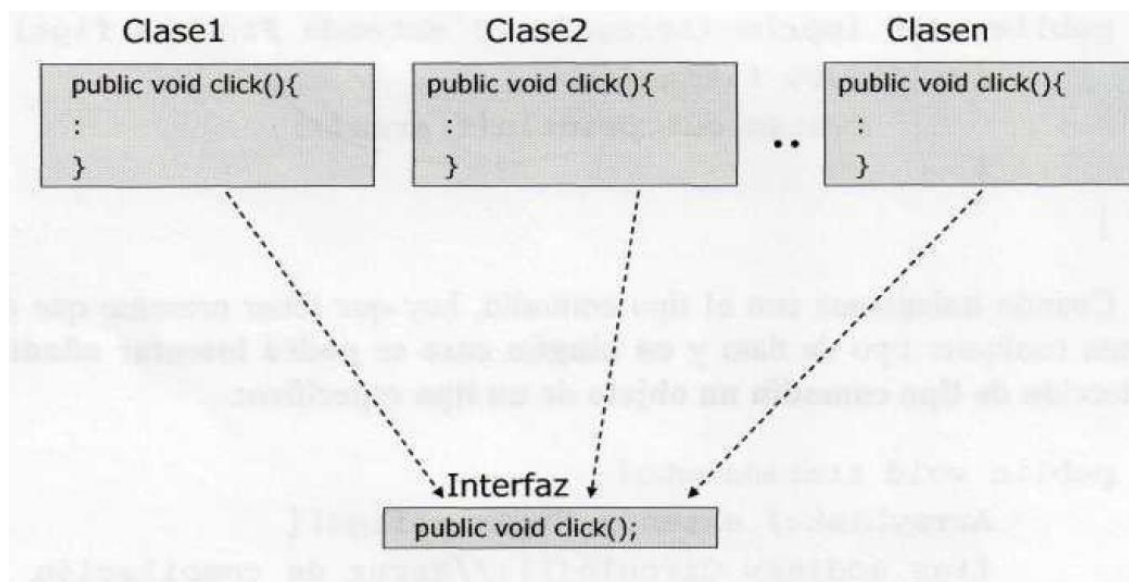
## Interfaces

Estrictamente hablando, una interfaz es un conjunto de métodos abstractos y de constantes públicas definidos en un archivo .java. Una interfaz es similar a una clase abstracta llevada al límite, en la que todos sus métodos son abstractos.

La finalidad de una interfaz es la de definir el formato que deben de tener determinados métodos que han de implementar ciertas clases.

Por ejemplo, para gestionar eventos en una aplicación basada en entorno gráfico, las clases donde se capturan estos eventos deben codificar una serie de métodos que se ejecutarán al producirse estos eventos. Cada tipo de evento tendrá su propio método de respuesta, cuyo formato estará definido en una interfaz. Así, aquellas clases que deseen responder a un determinado evento, deberán implementar el método de respuesta de acuerdo al formato definido en la interfaz.

Hay que insistir en el hecho de que una interfaz no establece lo que un método tiene que hacer y cómo hacerlo, sino el formato (nombre, parámetros y tipo de devolución) que éste debe tener.



### Definición de interfaz

Una interfaz se define mediante la palabra **interface**, utilizando la siguiente sintaxis:

```
[public] interface Nombre_interfaz{  
  
    //Constantes y métodos abstractos  
  
    tipo metodo1 (argumentos);  
    tipo metodo2 (argumentos);  
    .  
    .  
    .  
}
```

Por ejemplo,

```
public interface Operaciones{  
    void rotar();  
    String serializar();  
}
```

### Implementación de una interfaz

Como ya se ha dicho antes, el objetivo de las interfaces es proporcionar un formato común de métodos para las clases. Para forzar a que una clase defina el código para los métodos declarados en una determinada interfaz, la clase deberá *implementar* la interfaz.

En la definición de una clase, se utiliza la palabra **implements** para indicar qué interfaz se ha de implementar:

```
public class MiClase implements MiInterfaz{  
  
  
}
```

Por ejemplo,

```
public class Triangulo implements Operaciones  
{  
    public void rotar () {  
        //implementación del método  
    }  
    public String Serializar ( ) {  
        //implementación de método  
    }  
}
```

Sobre la implementación de interfaces, se ha de tener en cuenta lo siguiente:

- Al igual que sucede al heredar una clase abstracta, **cuando una clase implementa una interfaz, está obligada a definir el código (implementar) de todos los métodos existentes en la misma.** De no ser así, la clase deberá ser declarada como abstracta.
- **Una clase puede implementar más de una interfaz**, en cuyo caso, deberá implementar los métodos existentes en todas las interfaces. El formato utilizado en la definición de la clase será:

```
public class Míclase implements Interfaz1, Interfaz2,...{  
  
}
```

- **Una clase puede heredar otra clase e implementar al mismo tiempo una o varias interfaces.** En este sentido, las interfaces proporcionan una gran flexibilidad respecto a las clases abstractas a la hora de "forzar" a una clase a implementar ciertos métodos, ya que el implementar una interfaz no impide que la clase pueda heredar las características y capacidades de otras clases.

Por ejemplo, si la clase Triángulo quisiera tener los métodos *rotar()* y *serializar()*, podría implementar la interfaz Operaciones y seguir heredando la clase Figura.

Al mismo tiempo, el hecho de aislar esos métodos en la interfaz y no incluirlos en la clase Figura, permite que otras clases que no sean figuras puedan implementar esos métodos sin necesidad de heredar a ésta, mientras que el resto de clases que heredan Figura (Rectángulo, Círculo, etc.) y no desean disponer la capacidad de rotar y señalar, no se verán en la obligación de proporcionar dichos métodos.

La sintaxis utilizada para heredar una clase e implementar interfaces es:

```
public class Míclase extends Superclase implements Interfaz1, Interfaz2 {  
  
}
```

- **Una interfaz puede heredar otras interfaces.** No se trata realmente de un caso de herencia como tal, pues lo único que "adquiere" la subinterfaz es el conjunto de métodos abstractos existentes en la superinterfaz. La sintaxis utilizada es la siguiente:

```
public interface MilInterfaz extends Interfaz1, Interfaz2 {  
  
}
```

### Interfaces y polimorfismo

Como ya ocurriera con las clases abstractas, el principal objetivo que persiguen las interfaces con la definición de un formato común de métodos es el polimorfismo.

Una **variable de tipo interfaz puede almacenar cualquier objeto de las clases que la implementan**, pudiendo utilizar esta variable para invocar a los métodos del objeto que han sido declarados en la interfaz e implementados en la clase:

```
Operaciones op = new Triangulo();  
op.rotar();  
  
op.serializar();
```

Esta capacidad, unida a su flexibilidad, hacen de las interfaces una estructura de programación tremendamente útil.

### Interfaces en el Java SE

Además de clases, los paquetes de Java estándar incluyen numerosas interfaces, algunas de ellas son implementadas por las propias clases del Java SE y otras están diseñadas para ser implementadas en las aplicaciones.

Como muestra, comentamos algunas de las más importantes:

- **java.lang Runnable.** Contiene un método para ser implementado por aquellas aplicaciones que van a funcionar en modo multitarea.
- **java.util Enumeración.** La utilizamos en el apartado dedicado a las colecciones. Proporciona métodos que son implementados por objetos utilizados para recorrer colecciones.
- **java.awt.event WindowListener.** Proporciona métodos que deben ser implementados por las clases que van a gestionar los eventos (clases manejadoras) producidos en la ventana, dentro de una aplicación basada en entorno gráfico. Además de esta interfaz, hay otras muchas más para la gestión de otros tipos de eventos en los diversos controles gráficos Java.
- **java.sql Connection.** Interfaz implementada por los objetos utilizados para manejar conexiones con bases de datos. Además de ésta, el paquete java.sql contiene otras interfaces relacionadas con el envío de consultas SQL y la manipulación de resultados, como es el caso de Statement o ResultSet.
- **java.io Serializable.** Esta interfaz no contiene ningún método que deba ser definido por las clases que la implementan, sin embargo, la JVM requiere que dicha interfaz deba ser implementada por aquellas clases cuyos objetos tengan que ser transferidos a algún dispositivo de almacenamiento, como por ejemplo un archivo de disco.

### Ejemplos:

Ejemplo de interfaz que contiene solo **constantes**:

```
public interface DatosCentroDeEstudios {  
  
    byte NumeroDePisos = 5;  
    byte NumeroDeAulas = 25;  
    byte NumeroDeDespachos = 10;  
    // resto de datos constantes  
}
```

Ejemplo de interfaz que contiene solo **métodos**:

```
public interface CalculosCentroDeEstudios {  
  
    short NumeroDeAprobados(float[] Notas);  
    short NumeroDeSuspendidos(float[] Notas);  
    float NotaMedia(float[] Notas);  
    float Varianza(float[] Notas);  
    // resto de metodos  
}
```

Ejemplo de interfaz con **constantes y métodos**:

```
public interface CentroDeEstudios {  
  
    byte NumeroDePisos = 5;  
    byte NumeroDeAulas = 25;  
    byte NumeroDeDespachos = 10;  
    // resto de datos constantes  
    short NumeroDeAprobados(float[] Notas);  
    short NumeroDeSuspendidos(float[] Notas);  
    float NotaMedia(float[] Notas);  
    float Varianza(float[] Notas);  
    // resto de metodos  
}
```

También podemos definir un interfaz basado en otro. Utilizamos la palabra reservada ***extends***, tal y como hacemos para derivar subclases:

```
public interface CentroDeEstudios extends
DatosCentroDeEstudios {

    short NumeroDeAprobados(float[] Notas);
    short NumeroDeSuspendidos(float[] Notas);
    float NotaMedia(float[] Notas);
    float Varianza(float[] Notas);

    // resto de metodos

}
```

Al igual que hablamos de superclases y subclases, podemos emplear la terminología superinterfaces y subinterfaces. A diferencia de las clases, un interfaz puede extender simultáneamente a varios superinterfaces, lo que supone una aproximación a la posibilidad de realizar herencia múltiple:

```
public interface CentroDeEstudios extends DatosCentroDeEstudios,
CalculosCentroDeEstudios {

    // otros posibles métodos y constantes

}
```

Para poder utilizar los miembros de un interfaz es necesario implementarlo en una clase; se emplea la palabra reservada ***implements*** para indicarlo:

```
public class CCentroDeEstudios implements CentroDeEstudios {

    public short NumeroDeAprobados(float[] Notas) {
        short NumAprobados = 0;
        for (int i=0; i<Notas.length; i++)
            if (Notas[i]>=5.0f)
                NumAprobados++;
        return NumAprobados;
    }

    public short NumeroDeSuspendidos(float[] Notas) {
        short NumSuspendidos = 0;
    }
}
```



## MÓDULO PROGRAMACIÓN

```
        for (int i=0; i<Notas.length; i++)
            if (Notas[i]<5.0f)
                NumSuspendidos++;
        return NumSuspendidos;
    }

    public float NotaMedia(float[] Notas) {
        float Suma = 0;
        for (int i=0; i<Notas.length; i++)
            Suma = Suma + Notas[i];
        return Suma/(float)Notas.length;
    }

    public float Varianza(float[] Notas) {
        float Media = NotaMedia(Notas);
        float Suma = 0;
        for (int i=0; i<Notas.length; i++)
            Suma = Suma + Math.abs(Media-Notas[i]);
        return Suma/(float)Notas.length;
    }
}

public class ItfPrueba {
    public static void main(String[] args){
        float[] Notas={3,6,8,10,2,5,7,9,10};

        CCentroDeEstudios MiAula = new CcentroDeEstudios();

        System.out.println("Aprobados: "+MiAula
            .NumeroDeAprobados(Notas));

        System.out.println("Suspendidos: "+MiAula
            .NumeroDeSuspendidos(Notas));

        System.out.println("Media: "+MiAula .NotaMedia(Notas));
        System.out.println("Varianza: "+MiAula .Varianza(Notas));
    }
}
```