UTILIZACIÓN AVANZADA DE CLASES

Las clases pueden tener entre sí relaciones de:

- **Asociación.**
- **Herencia.**
- **Dependencia**

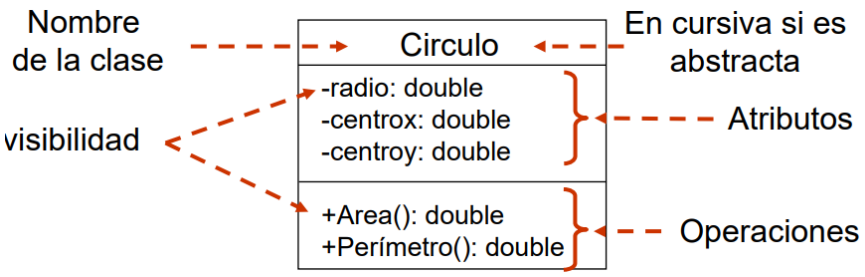
En esta unidad se profundiza en ello, mostrando cómo es posible organizar clases dentro de un programa para desarrollar estas relaciones. Se presta especial atención a la relación de herencia, esencial en todo lenguaje de programación orientado a objetos, mediante la cual es posible establecer jerarquías entre clases, haciendo que un conjunto de estas herede características de una clase base o padre.

En este contexto, se presentan los conceptos de polimorfismo, sobreescritura de métodos y diseño de clases abstractas e interfaces.

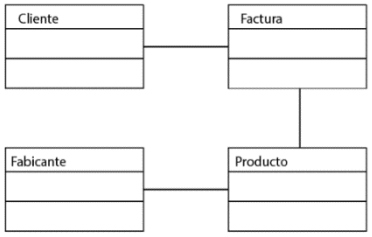
0. REPRESENTACIÓN DE UNA CLASE.

Como hemos visto en las unidades anteriores una clase proporciona una plantilla o molde sobre la cual instanciar diferentes objetos o entidades.

Una clase se conforma de: atributos y métodos. Para su representación podemos utilizar un diagrama UML reflejando la siguiente estructura:

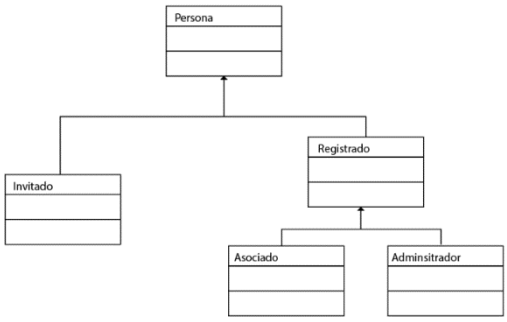


1. RELACIONES ENTRE CLASES



1.1. RELACIÓN DE ASOCIACIÓN.

Es una relación entre varias clases, incluso puede estar relacionada consigo misma. Los objetos en muchas ocasiones están relacionados por algunos atributos, como puede ser un cliente a un pedido y/o a una factura



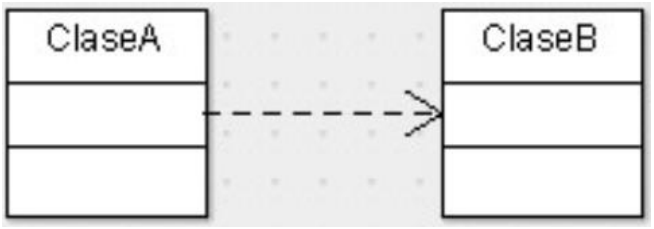
1.2. RELACIÓN DE GENERALIZACIÓN (HERENCIA)

Es la relación de clases donde una entidad más general se divide en otras más específicas. Son las relaciones de herencia, donde el elemento padre posee los atributos y métodos que heredan todos sus descendientes, mientras que los hijos poseen atributos únicos respecto al padre

1.3. RELACIÓN DE DEPENDENCIA (Relación de uso).

Relación entre dos clases en las que un elemento puede afectar el estado de otro. Esta es la relación menos importante de las comentadas. Simplemente refleja que la implementación de una clase depende del estado de la otra

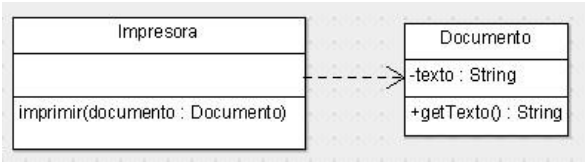
- La ClaseA usa a la ClaseB.
- La ClaseA depende de la ClaseB.
- Dada la dependencia, todo cambio en la ClaseB podrá afectar a la ClaseA.
- La ClaseA conoce la existencia de la ClaseB pero la ClaseB desconoce que existe la ClaseA.



En la práctica este tipo de relación se interpreta como que la *ClaseA* hace uso de la *ClaseB*, y lo más normal recibiéndola recibiéndola como parámetro de entrada en uno de sus métodos.

Supongamos lo siguiente:

- Tenemos una clase Impresora.
- Tenemos una clase Documento con un atributo texto.
- La clase Impresora se encarga de imprimir los Documentos.



Documento.java

```java
/* Clase Documento */
class Documento {
    private String texto;

    public Documento(String texto) {
        this.texto = texto;
    }

    public String getTexto() {
        return this.texto;
    }
}
```

Impresora.java

```java
/* Clase Impresora */
class Impresora {

    public Impresora() {

    }

    public void imprimir(Documento documento) {
        String texto = documento.getTexto();
        System.out.println(texto);
    }

}
```

En funcionamiento:

```java
Documento miDocumento = new Documento("Hello World!");
Impresora miImpresora = new Impresora();
miImpresora.imprimir(miDocumento);
```

2. RELACIÓN DE ASOCIACIÓN: AGREGACIÓN Y COMPOSICIÓN.

En este tipo de relación donde la existencia de un objeto depende de la existencia de otro objeto, encontramos las relaciones de Agregación y Composición son dos tipos de especialización de la relación de Asociación.

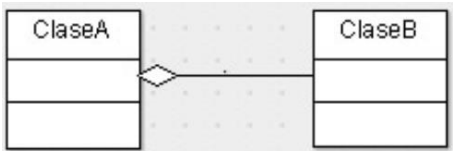
2.1. RELACIÓN DE AGREGACIÓN. (Relación débil)

La relación de agregación implica que una clase es "el todo" y la otra "es parte". Pero no implica que los objetos tengan una existencia dependiente una de la otra.

Se basa en la idea de entender una clase como una composición de otras clases.

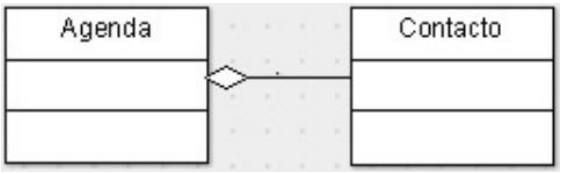
Se representa con una flecha que parte de una clase a otra en cuya base hay un rombo de color blanco.

La *ClaseA* agrupa varios elementos del tipo *ClaseB.*



Ejemplo:

- Tenemos una clase Agenda.
- Tenemos una clase Contacto.
- Una Agenda agrupa varios Contactos.



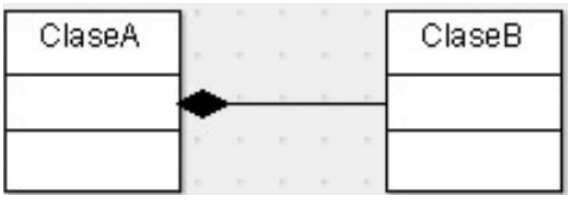
Ejemplo2:

- Una clase Coche
- Una clase Motor.
- Un coche contiene un motor.

2.2. RELACIÓN DE COMPOSICIÓN. (Relación fuerte)

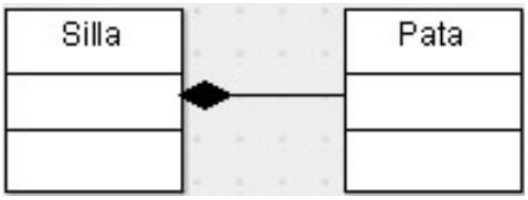
La relación entre el objeto compuesto y componente implica una dependencia fuerte o "relación de vida", es decir, el tiempo de vida de un objeto está condicionado por el tiempo de vida del objeto que lo incluye.

- *La* ClaseA *agrupa varios elementos del tipo* ClaseB.
- *El tiempo de vida de los objetos de tipo* ClaseB *está condicionado por el tiempo de vida del objeto de tipo* ClaseA.



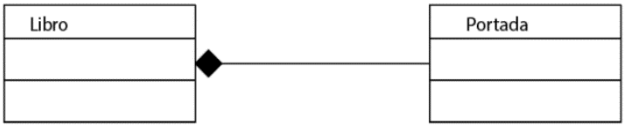
A diferencia de la agregación, esta relación sí obliga a la existencia de la otra clase. No puede existir de forma independiente

Ejemplo:



- Tenemos una clase Silla.
- Un objeto Silla está a su vez compuesto por cuatro objetos del tipo Pata.
- El tiempo de vida de los objetos Pata depende del tiempo de vida de Silla, ya que si no existe una Silla no pueden existir sus Patas.

Ejemplo2:

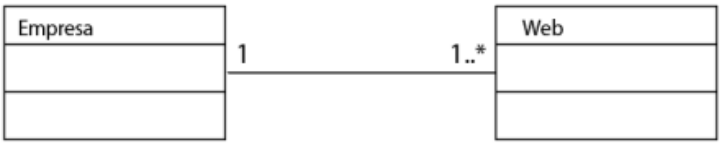


- Clase Libro.
- La clase Libro contiene un objeto de la clase Portada.
- No puede existir Portada sin libro. La eliminación de un objeto "Libro" de este soporte elimina también el objeto "Portada".

MÁS EJEMPLOS:

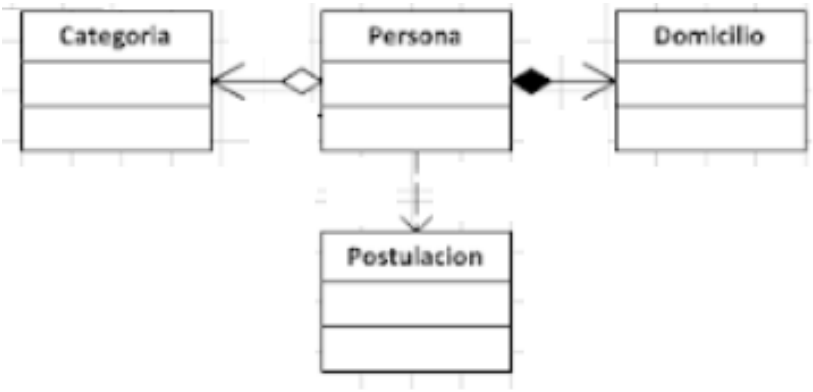
Un cliente tiene cero o varias facturas

Un Empleado tiene un o ningun portatil



DIFERENCIA AGRUPACIÓN Y COMPOSICIÓN EN JAVA + RELACIÓN DEPENDENCIA

- En composición, la clase componente debe ser creada/instanciada (con new) dentro de la clase compuesta.
- En agrupación, se crean ambos objetos (compuesto & componente) y luego ya sea por el constructor o paso por parámetros se inserta en el objeto compuesto.
- En dependencia, la clase compuesto no tiene como atributo ningún objeto de la clase dependiente. Sino que únicamente utiliza un objeto de forma puntual, ya sea creándolo para una funcionalidad concreta o recibiéndolo por argumento.




```
class Domicilio{
    private int personaID;
    private int calle;
    private int numero;
}
class Categoria{
    private int Id;
    private String IdCategoria;
}
class Postulacion{
    private int Id;
    public void Enviar() {

    }
}

class Persona{
    // Relación de composición: hago el new
    // dentro de la clase.
    Domicilio dom = new Domicilio();

    // Relación de agregación: No hago el new,
    // ya viene desde afuera resuelto.
    Categoria cat;
    public Persona(Categoria cat) {
        this.cat = cat;
    }

    // Relación de dependencia: No forma parte
    // de la clase, es utilizada para hacer
    // alguna operación.
    public void Postularse() {
        Postulacion p = new Postulacion();
        p.Enviar();
    }
}
```

2.3. CARDINALIDAD: MULTIPLICIDAD DE LAS ASOCIACIONES.

Indica el número de objetos que se relacionan con otro objeto. Para indicar la multiplicidad hay que indicar un mínimo y un máximo.

MULTIPLICIDAD (CARDINALIDAD)

- 1 Solo Uno
- 0..1 Cero o Uno
- M..N Desde M a N (Enteros Naturales)
- * o 0..* Cero a Muchos
- 1..* Uno a Muchos (al menos uno)

En las relaciones que permiten un factor de 0 son opcionales.

En cambio, las que tienen un factor de multiplicidad de 1 o más son obligatorias, es decir, en el caso de existir un objeto es obligatorio uno o más objetos de sus objetos asociados.

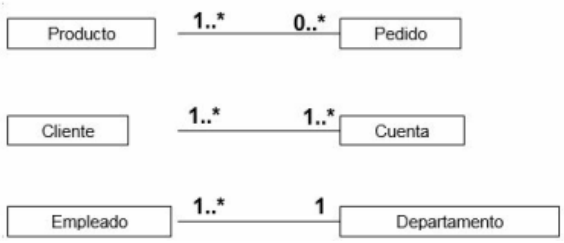
¿Cómo se interpreta la cardinalidad?

CASO 1:

- Un producto puede estar en varios pedidos o en ninguno.
- En cambio, un pedido debe tener al menos un producto, siendo obligatorio para su existencia.

CASO2:

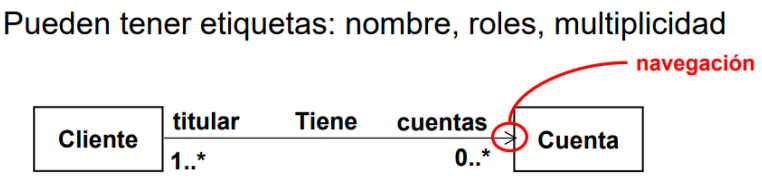
- Un cliente tiene al menos una cuenta. Un cliente puede tener varias cuentas.
- Una cuenta es de al menos un cliente. Una cuenta puede tener varios titulares

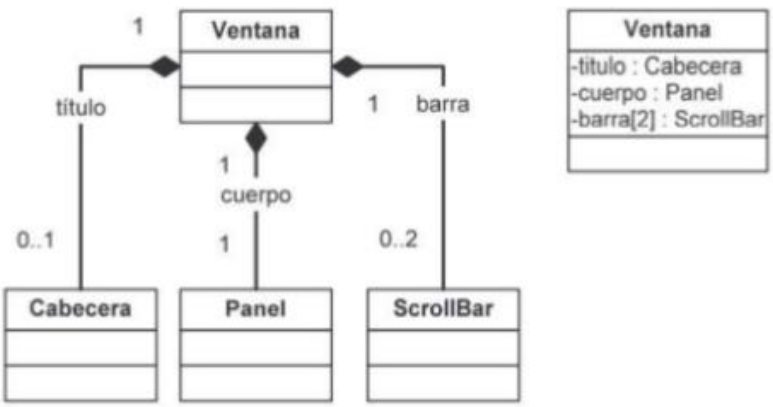


CASO3:

- Un departamento puede debe tener al menos un empleado o varios.
- Un empleado solo puede estar en un departamento, pero estará siempre en uno.

EL DIAGRAMA DE CLASES DE UNA RELACIÓN DE ASOCIACIÓN PUEDE CONTENER LA SIGUIENTE INFORMACIÓN:





3. RELACIÓN DE HERENCIA.

La herencia es clave dentro del paradigma POO. Permite relacionar una clase con otra mediante relaciones de jerarquía: padre – hijo o superclase – subclase.

Y de este modo definir clases (hijo) a partir de otras clases (padre). **La nueva clase hereda todos los atributos y la declaración de los métodos de la clase base, además, puede añadir nuevos atributos, nuevos métodos y también sobrescribir (suplantar) los métodos heredados**.

Una clase hija, o subclase, reúne todas las propiedades/métodos&atributos de la clase paterna, **creando una extensión de su comportamiento con nuevas funcionalidades**. Una subclase debe reunir todas las propiedades de la clase paterna más las que se añadan, pero nunca menos que la clase padre, por lo que extiende o especializa su funcionalidad.

VENTAJAS:

- **Favorece la reutilización de software.** Permite dotar de un comportamiento similar a todas las clases que heredan de una clase padre o superclase.
- **Permite definir un modelo consistente**, ya que el concepto de herencia permite un **esquema intuitivo y jerárquico**.

3.1. SUPERCLASES Y SUBCLASES

Cuando un conjunto de objetos tiene una serie de características similares, es posible agrupar estas características y comportamientos comunes en una entidad de orden superior.

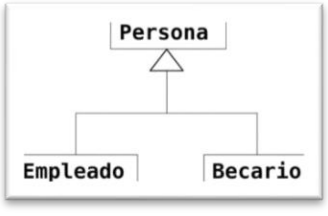
Ejemplo: En el caso de un coche, autobús o camión se pueden agrupar o dentro de una superclase llamada “vehículo” y pese que estos 3 objetos compartan los atributos y métodos comunes de un vehículo luego podrán tener características propias diferenciadoras.

La herencia establece una estructura jerárquica donde cada clase hereda atributos y métodos de las clases que están por encima suyo. Y luego, a su vez pueden tener atributos o comportamientos propios específicos de una clase hija/subclase.

3.1.1. CONCEPTOS:

- **Superclase**, clase padre.
- **Subclase**, clase hija o clase derivada.

3.1.2. TIPOS DE HERENCIAS:



Hay dos tipos de herencia:

- **HERENCIA SIMPLE**: Una clase hereda de una clase y únicamente una superclase. Aunque una clase padre puede tener varias clases hija.

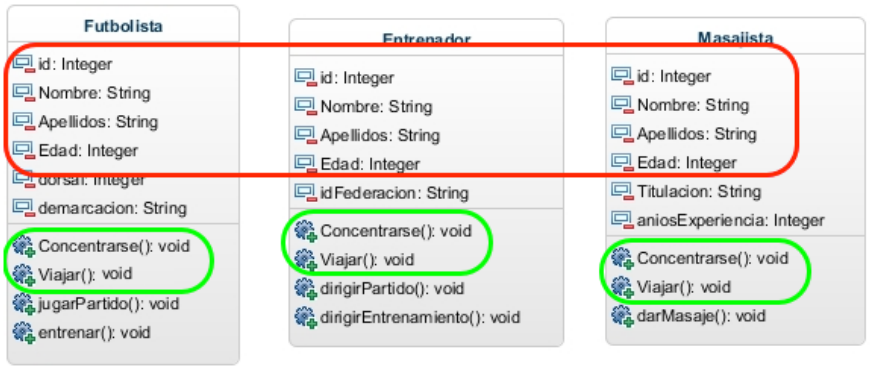
- **HERENCIA MÚLTIPLE**: Una clase hereda de varias superclases.

En el caso de Java únicamente contamos con herencia simple.

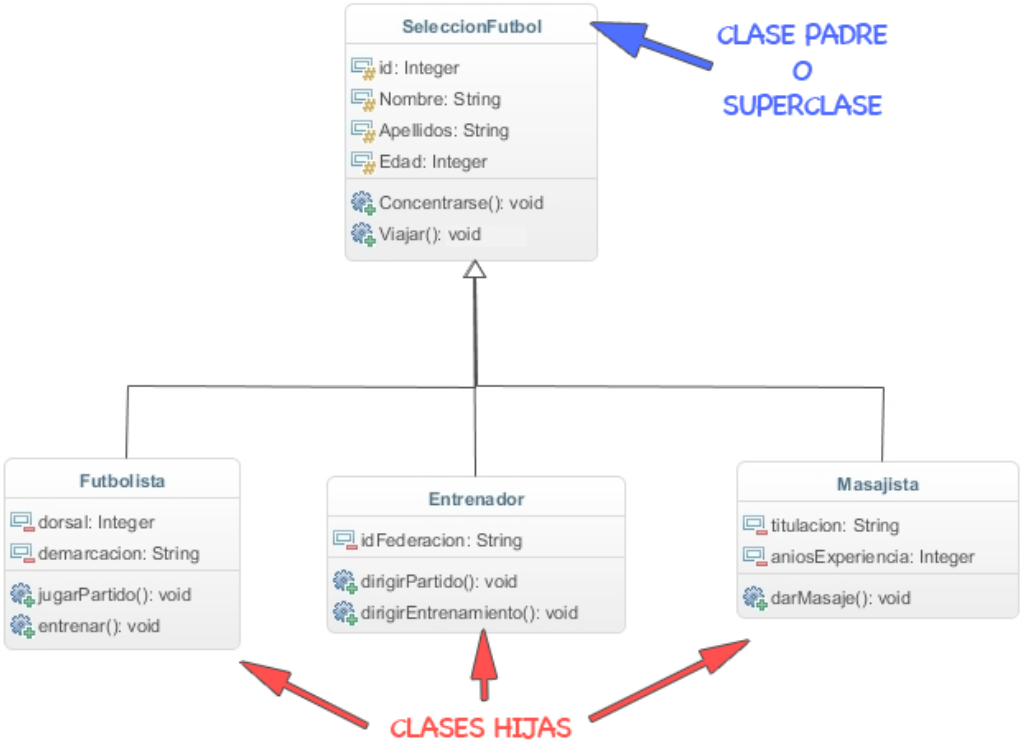
EJEMPLO USO DE HERENCIA:

1. Modelamos 3 clases: Futbolistas, Entrenador y Masajista.

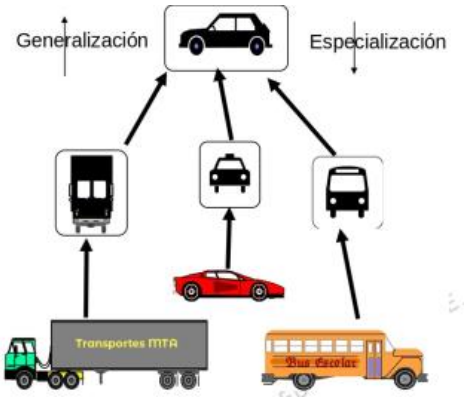
2. Observamos que algunos atributos y métodos son comunes.



3. Es posible agrupar las partes comunes en una clase padre, propiciando la reutilización de código.



3.1.3. GENERALIZACIÓN vs ESPECIALIZACIÓN.



Especialización: La especialización parte de un caso genérico y se concreta sobre casos particulares. La especialización es la forma más común de diseñar la herencia.

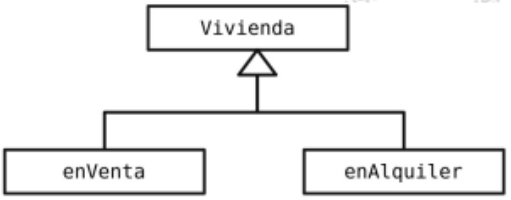
3.2. CARÁCTERÍSTICAS – HERENCIA EN JAVA.

1. Mecanismo de extensión de clases: **extends.**
2. Jerarquía de clases: JAVA sólo se permite un único padre (herencia simple).
3. Una clase que **extiende a otra hereda todos sus atributos y todos sus métodos** (no sus constructores).
    a. Siempre que los métodos sean public y/o protected.
    b. Java recomienda:
        i. Atributos: private
        ii. Métodos: public.
4. Puede añadir atributos y métodos nuevos.
    a. **OJO!** Si un nuevo atributo / método se llama igual que otro de una superclase, lo solapa (`@Override`, y ya no puede acceder al de la clase padre).
5. Existe una asociación entre clase hija y clase padre de tipo "es un…"
    a. Un empleado **es un** trabajador.
    b. Un consultor **es un** trabajador.
6. Si queremos una clase no puede ser extendida / no pueda ser padre de ninguna otra clase. Podemos marcarla como **final**.

```
public final class ClaseFinal {

}
```

7. La clase hija*, no hereda los constructores de la clase padre*. Por lo que al crear un constructor es necesario diferenciar la parte del constructor de la clase padre (**super**) y la parte del constructor de la clase hija.

EJEMPLO 1: Imagínese una inmobiliaria que ofrece casas a la venta y alquiler de pisos. Para las viviendas en venta se maneja información del precio, la señal y la comisión que se aplica, mientras que para las viviendas en alquiler se maneja la información del precio de alquiler y el número de meses de fianza. Para ambos tipos de vivienda se tiene la información de altura del piso, número de habitaciones y el distrito postal donde se ubica.



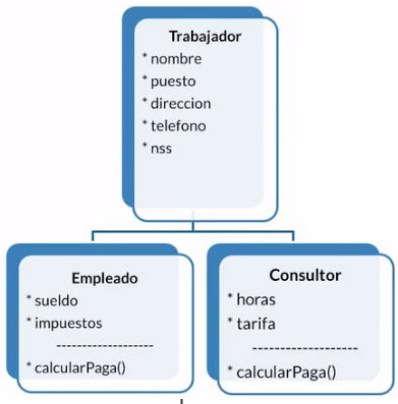

```java
package inmobiliaria;

public class Vivienda {

    private int altura;
    private int habitaciones;
    private int codPostal;

    public Vivienda(int altura, int habitaciones,
            int codPostal) {
        super();
        this.altura = altura;
        this.habitaciones = habitaciones;
        this.codPostal = codPostal;
    }
    public int getAltura() {
        return altura;
    }
    public void setAltura(int altura) {
        this.altura = altura;
    }
    public int getHabitaciones() {
        return habitaciones;
    }
    public void setHabitaciones(int habitaciones) {
        this.habitaciones = habitaciones;
    }
    public int getCodPostal() {
        return codPostal;
    }
    public void setCodPostal(int codPostal) {
        this.codPostal = codPostal;
    }

}
```

```java
package inmobiliaria;

public class EnAlquiler extends Vivienda{

    private double precioAlquiler;
    private int mesesFianza;

    public EnAlquiler(int altura, int habitaciones,
            int codPostal, double precioAlquiler,
            int mesesFianza) {
        super(altura, habitaciones, codPostal);
        this.precioAlquiler = precioAlquiler;
        this.mesesFianza = mesesFianza;
    }
    public double getPrecioAlquiler() {
        return precioAlquiler;
    }
    public void setPrecioAlquiler(double precioAlquiler) {
        this.precioAlquiler = precioAlquiler;
    }
    public int getMesesFianza() {
        return mesesFianza;
    }
    public void setMesesFianza(int mesesFianza) {
        this.mesesFianza = mesesFianza;
    }

}
```

```java
package inmobiliaria;

public class EnVenta extends Vivienda{

    private double precio;
    private double señal;
    private double comision;

    public EnVenta(int altura, int habitaciones,
            int codPostal, double precio,
            double señal, double comision) {
        super(altura, habitaciones, codPostal);
        this.precio = precio;
        this.señal = señal;
        this.comision = comision;
    }
    public double getPrecio() {
        return precio;
    }
    public void setPrecio(double precio) {
        this.precio = precio;
    }
    public double getSeñal() {
        return señal;
    }
    public void setSeñal(double señal) {
        this.señal = señal;
    }
    public double getComision() {
        return comision;
    }
    public void setComision(double comision) {
        this.comision = comision;
    }

}
```

EJEMPLO2:

- Dos tipos de trabajadores: Empleado y Consultor.




```java
public class Trabajador {

    private String nombre;
    private String puesto;
    private String dirección;
    private long telefono;
    private String nss; // numero seguridad social

    public Trabajador(String nombre, String puesto, String dirección, long telefono, String nss) {
        this.nombre = nombre;
        this.puesto = puesto;
        this.dirección = dirección;
        this.telefono = telefono;
        this.nss = nss;
    }
```

```java
public class Empleado extends Trabajador{

    private double sueldo;
    private double impuestos;

    private final int NUMERO_PAGAS = 12;

    public Empleado(String nombre, String puesto,
            String dirección, long telefono,
            String nss, double sueldo,
            double impuestos) {
        super(nombre, puesto, dirección, telefono, nss);
        this.sueldo = sueldo;
        this.impuestos = impuestos;
    }

    public double getSueldo() {
        return sueldo;
    }
    public void setSueldo(double sueldo) {
        this.sueldo = sueldo;
    }
    public double getImpuestos() {
        return impuestos;
    }
    public void setImpuestos(double impuestos) {
        this.impuestos = impuestos;
    }

    public double calcularPaga() {
        return (sueldo-impuestos) / NUMERO_PAGAS;
    }
}
```

```java
public class Consultor extends Trabajador{

    private int horas;
    private double tarifa;

    public Consultor(String nombre, String puesto,
            String dirección, long telefono,
            String nss, int horas,
            double tarifa) {
        super(nombre, puesto, dirección, telefono, nss);
        this.horas = horas;
        this.tarifa = tarifa;
    }

    public double calcularPaga() {
        return horas*tarifa;
    }

}
```

MAIN:

```java
Trabajador trabajador;
Empleado empleado;
Consultor consultor;

trabajador = new Trabajador("Bill Gates", "Presidente", "Redmond", "", "");
empleado = new Empleado("Larry Ellison", "Presidente", "Redwood", "", "", 100000.0, 1000.0);
consultor = new Consultor("Steve Jobs", "Consultor Jefe", "Cupertino", "", "", 20, 1000.0);

System.out.println(trabajador);
System.out.println(empleado);
System.out.println(empleado.calcularPaga());
System.out.println(consultor);
System.out.println(consultor.calcularPaga());
```

3.2. USO DE SUPER PARA ACCEDER A UNA CLASE PADRE Y SUS CONSTRUCTORES.

- Permite acceder a un método o constructor de la clase padre con el objeto de ampliar su implementación.

```
// overrides printMethod in Superclass
public void printMethod() {
   super.printMethod();
   System.out.println("Printed in Subclass");
}
```

- Invocar un constructor de la clase padre desde el constructor de la clase hijo. Sino se hace JVM lo hace automáticamente, eso sí, es necesario una clase por defecto en la clase padre.

```
public class Empleado extends Trabajador{

    private double sueldo;
    private double impuestos;

    private final int NUMERO_PAGAS = 12;

    public Empleado(String nombre, String puesto,
            String dirección, long telefono,
            String nss, double sueldo,
            double impuestos) {

        super(nombre, puesto, dirección, telefono, nss);
        this.sueldo = sueldo;
        this.impuestos = impuestos;

    }
```

```
public class Empleado extends Trabajador{

    private double sueldo;
    private double impuestos;

    private final int NUMERO_PAGAS = 12;

    public Empleado(double sueldo, double impuestos) {
        super();
        this.sueldo = sueldo;
        this.impuestos = impuestos;
    }
```

```
public class Trabajador {

    private String nombre;
    private String puesto;
    private String dirección;
    private long telefono;
    private String nss; // numero seguridad social

    public Trabajador() {
        // TODO Auto-generated constructor stub
    }

    public Trabajador(String nombre, String puesto,
            String dirección, long telefono, String nss) {

        this.nombre = nombre;
        this.puesto = puesto;
        this.dirección = dirección;
        this.telefono = telefono;
        this.nss = nss;

    }
```

Crear dos clase: Clase Padre & Clase Hija con dos métodos imprimirMensaje().

```
public class ClaseHija extends ClasePadre{

    public void imprimirMensaje() {
        super.imprimirMensaje();
        System.out.println("Imprimir desde hija");
    }

}
```

```
public class ClasePadre {

    public void imprimirMensaje() {
        System.out.println("Imprimir desde padre");
    }

}
```

```
public class MainClase {

    public static void main(String[] args) {
        ClasePadre clase = new ClasePadre();
        ClasePadre clase2 = new ClaseHija();

        clase.imprimirMensaje();
        System.out.println("-------");
        clase2.imprimirMensaje();
    }
}
```

3.3. POLIMORFISMO:

"Polimorfismo" es una palabra de origen griego que significa **"muchas formas".**

El polimorfismo en la programación orientada a objetos, es la capacidad que se le da a un método, de comportarse de manera diferente de acuerdo a la instancia creada.

Dependiendo de la clase que instancie el objeto realizará un método u otro.

Y es que una subclase puede ser accedida a través de una referencia de una superclase. En lugar de instanciar con el mismo nombre que la clase con que se fine el define sino la clase que comportamiento queramos.

Por ejemplo, si tenemos:

```
public static void saludar(Trabajador t) {
    System.out.println("Hola, " + t.getNombre());
}
```

USO DE REFERENCIAS DE CLASE PADRE, PERO INSTANCIACIÓN DE OBJETOS HIJO.

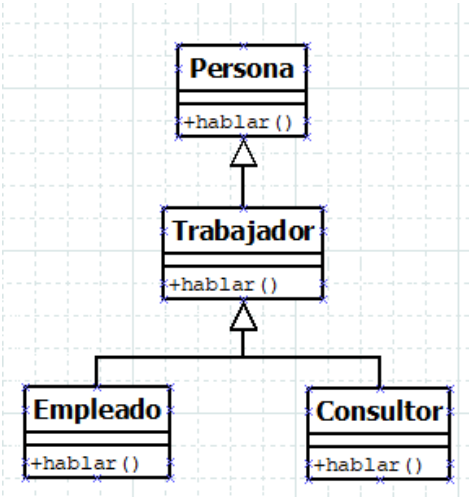
<< LA CLASE TRABAJADOR PODRÁ ADOPTAR DIVERSAS FORMAS >>

```
public class MainTrabajador {

    public static void main(String[] args) {

        Trabajador trabajador = new Trabajador("Pepe", "Directivo", "no_name", 2223333, "333X");
        Trabajador trabajador_empleado = new Empleado("Luis", "Directivo", "no_name", 2223333, "333X", 1200, 100);
        Trabajador trabajador_consultor = new Consultor("Juan", "Directivo", "no_name", 2223333, "333X", 12, 13.3);

        saludar(trabajador);
        saludar(trabajador_empleado);
        saludar(trabajador_empleado);

        trabajador.calcularPaga();
        trabajador_empleado.calcularPaga();
        trabajador_consultor.calcularPaga();

    }

    public static void saludar(Trabajador trabajador) {
        System.out.println("hola: "+trabajador.getNombre());
    }

}
```

Si una clase hija añade un método con mismo nombre y firma que otro de la clase padre, oculta este: Java escoge automáticamente el tipo de objeto.

- Si este oculta un método de la superclase – ejecuta el método de la clase hijo.
- Si no, llama al método de la clase padre.

PRACTICA:



- Al esquema anterior introduciremos una nueva clase llamada Persona y haremos que Trabajador herede dicha clase.
- Crearemos un método llamado hablar() en todos y cada uno de las clases cambiando el texto.

```
public void hablar() {
    System.out.println("Persona habla");
}
public void hablar() {
    System.out.println("Trabajador habla");
}
public void hablar() {
    System.out.println("Empleado habla");
}
public void hablar() {
    System.out.println("Consultor habla");
}
```

- A continuación, si creamos en el main diversos objetos definiéndolos todos con la clase Persona, pero modificando el constructor a la hora de instanciarlo.

```
Persona persona = new Persona();
Persona trabajador = new Trabajador("Pepe", "Directivo", "no_name", 2223333, "333X");
Persona trabajador_empleado = new Empleado("Luis", "Directivo", "no_name", 2223333, "333X", 1200, 100);
Persona trabajador_consultor = new Consultor("Juan", "Directivo", "no_name", 2223333, "333X", 12, 13.3);

persona.hablar();
trabajador.hablar();
trabajador_consultor.hablar();
trabajador_consultor.hablar();
```

3.4. CLASES Y MÉTODOS FINALES.

- Clase final: Indica que no puede tener clases que deriven de ella. No permite herencia con dicha clase.
- Cuando un método no se quiere que sea modificable (@override) en las clases hija.

En el ejemplo anterior (Ejemplo2) si indicamos que la clase “padre” es una clase final, entonces ninguna clase puede heredar de ella, por lo que las clases hijas “Empleado” y “Consultor” daría error, al no poder extender dicha clase.

```
final class Trabajador {

    private String nombre;
    private String puesto;
    private String dirección;
    private long telefono;
    private String nss; // numero seguridad social

    public Trabajador() {
        // TODO Auto-generated constructor stub
    }

    public Trabajador(String nombre, String puesto,
            String dirección, long telefono, String nss) {
```

Igualmente en el ejemplo anterior si ponemos el método calcularPaga() como final, las clase hijas no podrían modificar dentro de estas.

```
final double calcularPaga() {
    return 0;
}
```