El futuro digital es de todos

MinTIC

Hechos QUE CONECTAN

CICLO 2
EJE TEMÁTICO 3
RELACIÓN ENTRE CLASES

Universidad Industrial de Santander

UIS

MisiónTIC2022

Relación entre clases

Ruta de aprendizaje

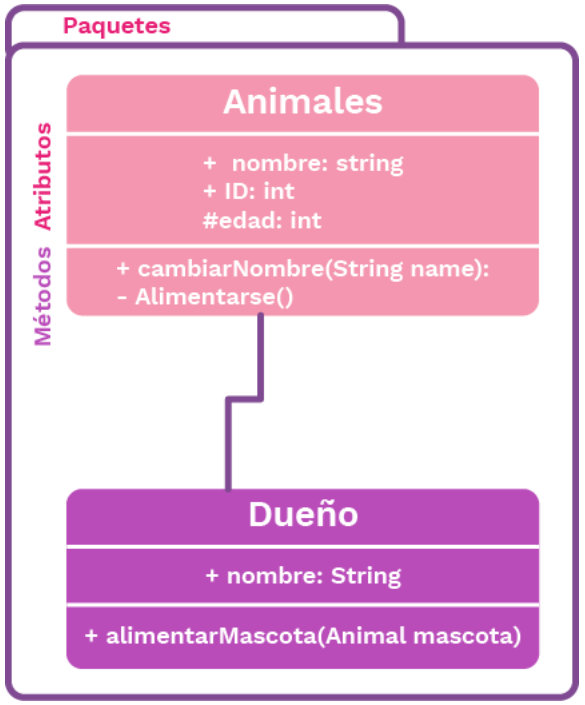
Relaciones entre clases

Previamente, se ha aclarado la diferencia entre la **codificación secuencial** y la **programación orientada a objetos**, explicando que es una clase y como obtener objetos. Ahora veremos las relaciones que pueden existir entre estas clases. Se abordará los conceptos de **asociación**, **agregación** y **composición**.

3. ¿Para qué sirven las relaciones entre clases?

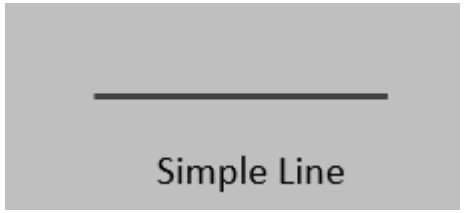
Las relaciones existentes entre las distintas clases indican cómo estas se comunican entre sí; es decir, nos indican cómo se comunican los objetos de esas clases.

Por ejemplo, la primera relación entre clases que vamos a mencionar es la **asociación**, la cual es la relación primaria que se tiene, es decir, la relación de asociación es la base para las demás relaciones entre clases.



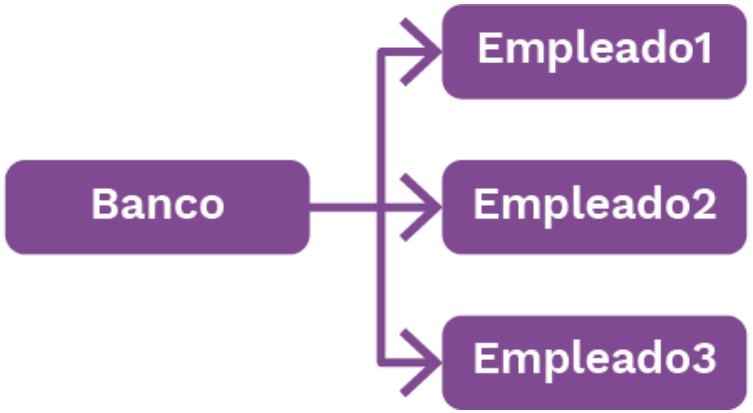
3.1. Asociación

Una forma sencilla de describir la asociación sería la siguiente: Es la conexión o relación entre, al menos, dos clases separadas y diferentes, la cual se da a través de sus objetos. En la siguiente imagen vemos la relación que tiene, por ejemplo, un banco y sus empleados.

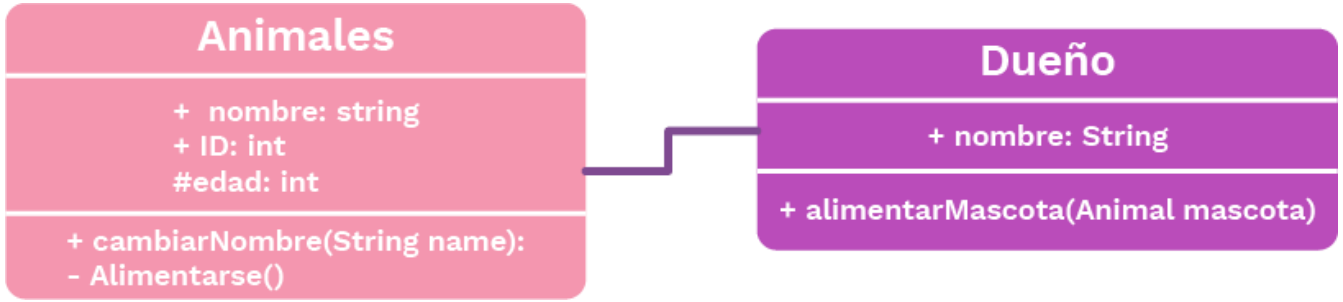


Representación en UML de la asociación

Asociación unidireccional



Asociación bidireccional



3.1. Asociación

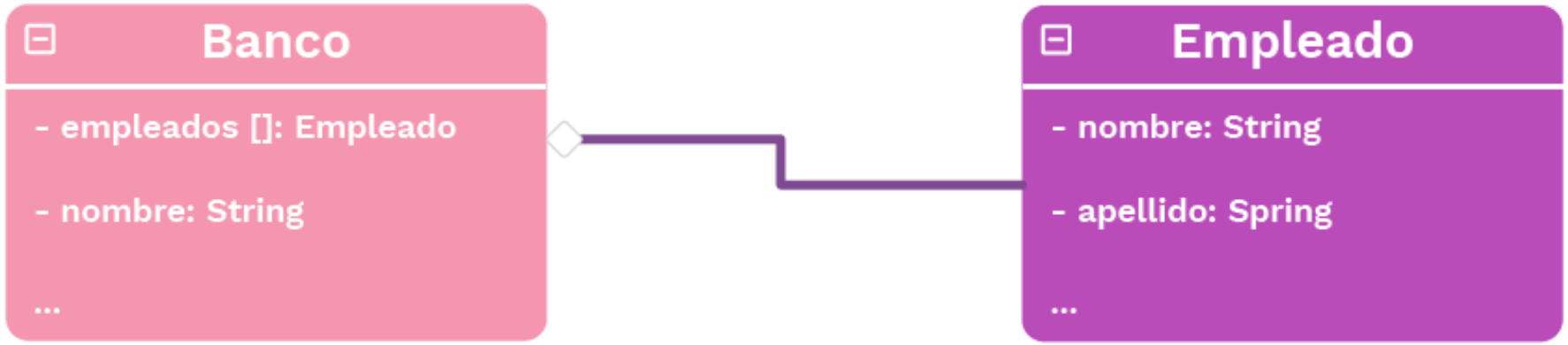
Esta relación es la más abstracta, aunque la más sencilla de entender, porque, en últimas, cuando nos referimos a la asociación estamos haciendo referencia a una relación entre diferentes clases.

Esta relación puede darse de dos formas similares, pero con características específicas: la agregación y composición. Esto significa que la asociación engloba dichas relaciones específicas, como vemos en la siguiente imagen:

3.2. Agregación

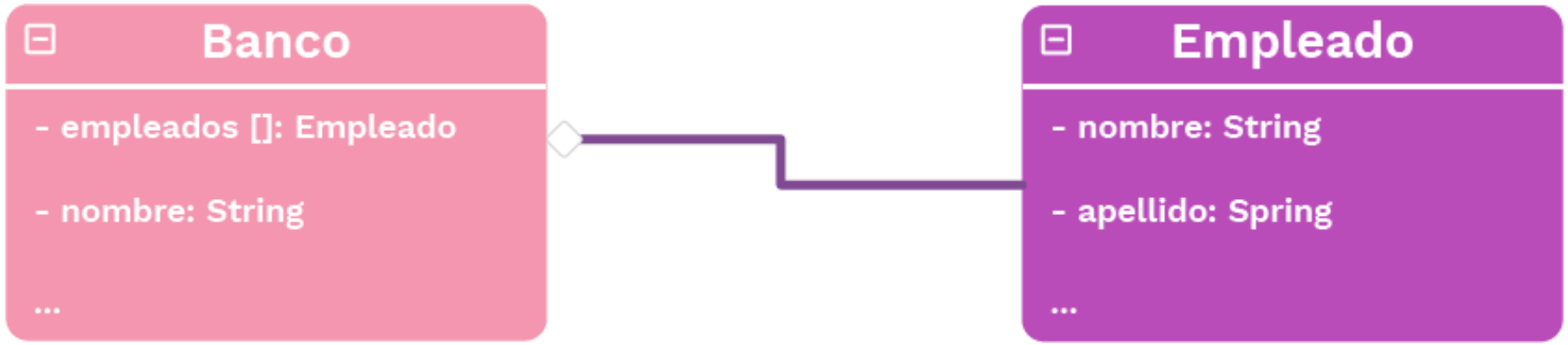
La agregación es la relación entre diferentes clases que se da con más frecuencia, esta tiene un concepto sencillo, consiste en, como dice su nombre, agregar objetos a una estructura que nos almacene objetos del mismo tipo, por ejemplo, un arreglo (Array).

Representación en UML de la agregación



3.2. Agregación

Una característica crucial de la agregación: Los **objetos que nosotros agregamos pueden existir sin la necesidad de que exista** la asociación entre ellos y el otro tipo de objeto, es decir, y haciendo uso de un ejemplo, los empleados pueden existir o vivir perfectamente sin necesidad de que exista el banco.



3.3. Composición

La relación de composición es muy similar a la de agregación, de hecho, consisten en lo mismo, no obstante, tienen una diferencia crucial: a diferencia de la agregación, en la relación de composición, **los objetos agregados SÍ dejan de existir si el objeto que los almacena deja de existir, porque ya no tendría sentido lógico mantenerlos.**

Es decir, en esta relación los objetos que nosotros agregamos, están destinados a dejar de existir si su objeto padre deja de existir. A esa clase de objetos que van a cumplir ese propósito, los llamaríamos componentes.

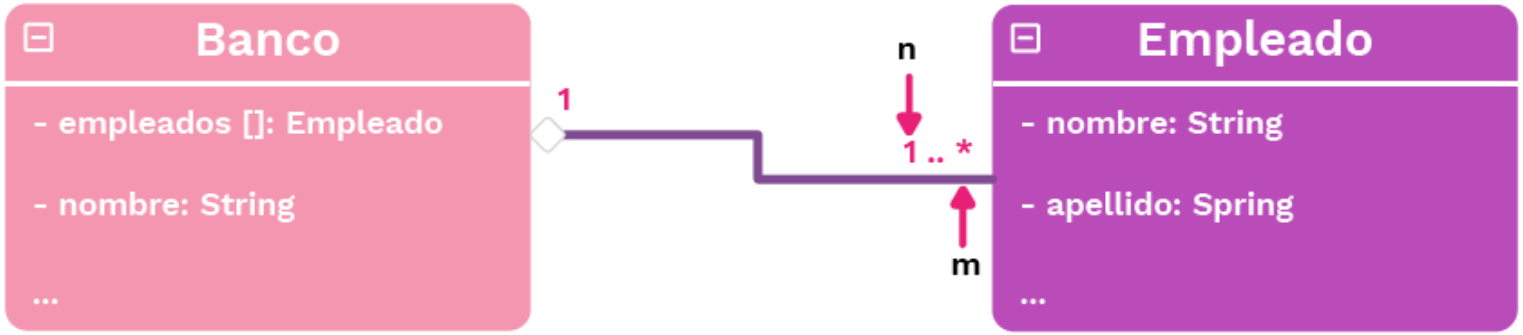
Representación en UML de la composición

3.4. Cardinalidad

Indica un rango de valores que un objeto, dada una relación de asociación, contendrá de otro tipo de objeto. En otras palabras, la cardinalidad, o también llamada multiplicidad, indica cuántas instancias de una clase pueden surgir, fruto de la relación con otra clase.

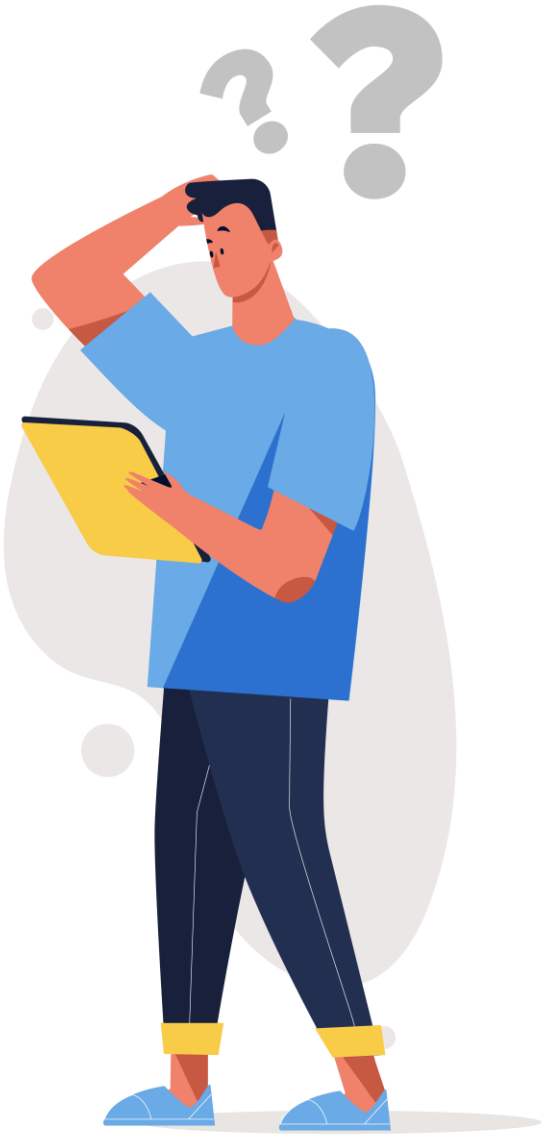
La cardinalidad se representa de 3 formas base diferentes, dependiendo del caso:

- n -> el número máximo de instancias que se puede tener en ese extremo de la relación
- n .. m ->el primer valor, la n, nos indica el mínimo posible, mientras que la m, el máximo.
- n .. * -> mínimo de valores, con un máximo desconocido, es decir, que no tiene límite.



Revisión de lo aprendido

¿Cual de los siguientes códigos es una relación de composición?



A)

```java
public class Casa {
    int dimensiones;
    String hogar;
    Habitacion habitacion;

    public Casa(int dimensiones, String hogar) {
        this.dimensiones = dimensiones;
        this.hogar = hogar;
    }

    public int getDimensiones() {
        return dimensiones;
    }

    public void setDimensiones(int dimensiones) {
        this.dimensiones = dimensiones;
    }

    public String getHogar() {
        return hogar;
    }

    public void setHogar(String hogar) {
        this.hogar = hogar;
    }
```

```java
public class Habitacion {
    String nombreHabitacion;
    int dimension1;
    int dimension2;
    float area;

    public Habitacion(String nombreHabitacion, int dim
            int dimension2, float area) {
        this.nombreHabitacion = nombreHabitacion;
        this.dimension1 = dimension1;
        this.dimension2 = dimension2;
        this.area = area;
    }

    public String getNombreHabitacion() {
        return nombreHabitacion;
    }

    public void setNombreHabitacion(String nombreHabit
        this.nombreHabitacion = nombreHabitacion;
    }

    public int getDimension1() {
        return dimension1;
    }

    public void setDimension1(int dimension1) {
        this.dimension1 = dimension1;
    }
```

B)

```
 */
@Override
public int compareTo(Object a) {
    int estado=MENOR;

    //Hacemos un casting de objetos para usar el metodo get
    Videojuego ref=(Videojuego)a;
    if (horasEstimadas>ref.getHorasEstimadas()){
        estado=MAYOR;
    }else if(horasEstimadas==ref.getHorasEstimadas()){
        estado=IGUAL;
    }

    return estado;
}

/**
 * Muestra informacion del videojuego
 * @return cadena con toda la informacion del videojuego
 */
@Override
public String toString(){
    return "Informacion del videojuego: \n" +
            "\tTitulo: "+titulo+"\n" +
            "\tHoras estimadas: "+horasEstimadas+"\n" +
            "\tGenero: "+genero+"\n" +
            "\tcompañia: "+compañia;
}
```

```
public class CuentaApp {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        Cuenta cuenta_1 = new Cuenta("Federico");
        Cuenta cuenta_2 = new Cuenta("David", 400);

        //Ingresa dinero en las cuentas
        cuenta_1.ingresar(300);
        cuenta_2.ingresar(400);

        //Retiramos dinero en las cuentas
        cuenta_1.retirar(500);
        cuenta_2.retirar(100);

        //Muestro la informacion de las cuentas
        System.out.println(cuenta_1); // 0 euros
        System.out.println(cuenta_2); // 600 euros

    }

}
```

Revisión de lo aprendido

1. Al analizar la estructura de un Objeto se puede considerar como una especie de bloque dividido en tres partes, donde cado uno de ellas desempeñan un papel independiente. ¿Cuál de estos no hace parte de la estructura de un Objeto?

a. Los mensajes
b. Las propiedades
c. Las relaciones
d. Los métodos

Revisión de lo aprendido

2. Un objeto del tipo empleado posee cero o muchos objetos del tipo vehículo. Un objeto del tipo vehículo es de un y solo un objeto del tipo empleado. Según esta definición, en un análisis de la estructura de objeto se puede considerar como:

a. Herencia
b. Asociaciones de Objetos
c. Composición de objetos
d. Jerarquías Compuestas

Revisión de lo aprendido

3. Dados los tipos de objetos PROFESORES y SALONES. Nos interesan datos históricos en esta relación. La multiplicidad sería respectivamente

a. (0,1) ---------- (1,n)
b. (0,n) ---------- (1,n)
c. (1,n) ---------- (1,n)
d. (1,n) ---------- (0,1)

Revisión de lo aprendido

4. La composición es esta relación donde los objetos están interrelacionados, y un elemento es parte del otro, seleccione cuál de las siguientes opciones es adecuada para este tipo de relación:

A) Si los objetos están interrelacionados y uno deja de existir el otro también.

B) Los objetos son independientes y no sucede nada si se elimina alguno.

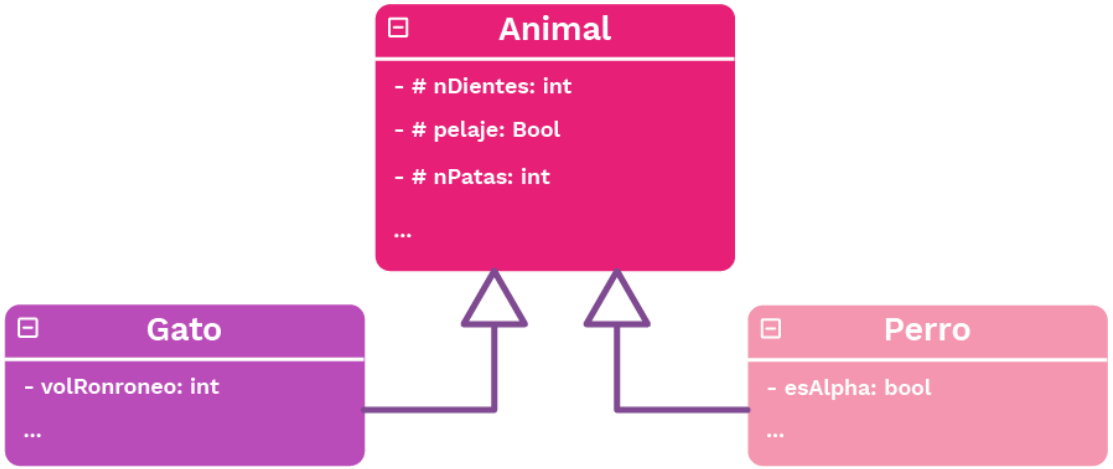
C) Una sola clase depende de la otra, si se elimina la extremidad la clase principal desaparece.

D) Si se elimina el objeto principal, los objetos agregados desaparecen.

3.5. Herencia

Es el poder o capacidad de crear clases que, de forma automática, **tomen las características** (atributos y métodos) de otra clase que exista previamente, donde nosotros, basados en una lógica, elegimos con un fin. A la nueva clase generada, le podremos añadir nuevos atributos y métodos propios. **Esto permite evitar el mal hábito de copiar y pegar código de otras clases**

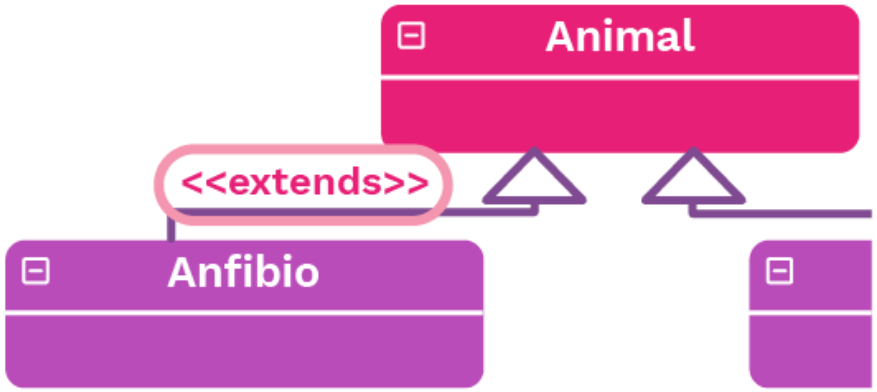
Representación en UML de la herencia



3.5. Herencia

En Java, la forma en la que nosotros podemos implementar la herencia es bastante sencilla. Basta con escribir el código de la superclase una sola vez, luego, cada subclase solo tendrá que añadir una pequeña palabra reservada, "extends", seguida del nombre de la clase a heredar.

```java
public class Gato extends Animal{

    protected int volRonroneo;

    public Gato(int nDientes, boolean pelaje, int nPatas, int volRonroneo) {
        // Atributos heredados
        this.nDientes = nDientes;
        this.pelaje = pelaje;
        this.nPatas = nPatas;
        // Atributos particulares
        this.volRonroneo = volRonroneo;
    }

}
```



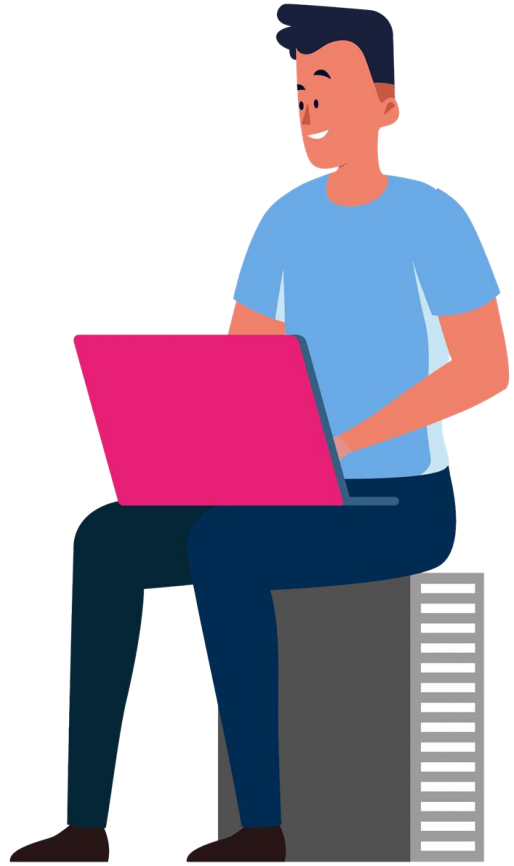
3.5.1. Sobrecarga de métodos

En este apartado,introducimos el uso del this que permite asignar el valor al atributo según el constructor que sea seleccionada cuando existe la sobrecarga de estos.

Palabra this
- Es una referencia implícita al objeto que se está ejecutando
- Es útil para identificar la ambigüedad entre las variables de clase y las locales
- Permite a un objeto enviarse como el parámetro

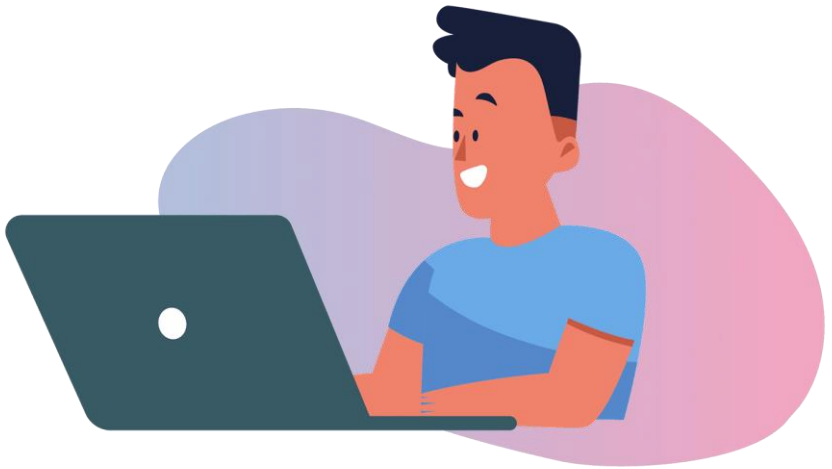
Como se muestra en el ejemplo, las clases pueden tener múltiples constructores y métodos llamados igual; siempre y cuando, tengan un parámetro que les permita diferenciarse.

```java
public class Rectangle {
    private int x, y, width, height;

    public Rectangle() {
        this.x = 0;
        this.y = 0;
        this.width = 1;
        this.height = 1;

    }
    public Rectangle(int width, int height) {
        this.x = 0;
        this.y = 0;
        this.width = width;
        this.height = height;

    }
    public Rectangle(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
}
```

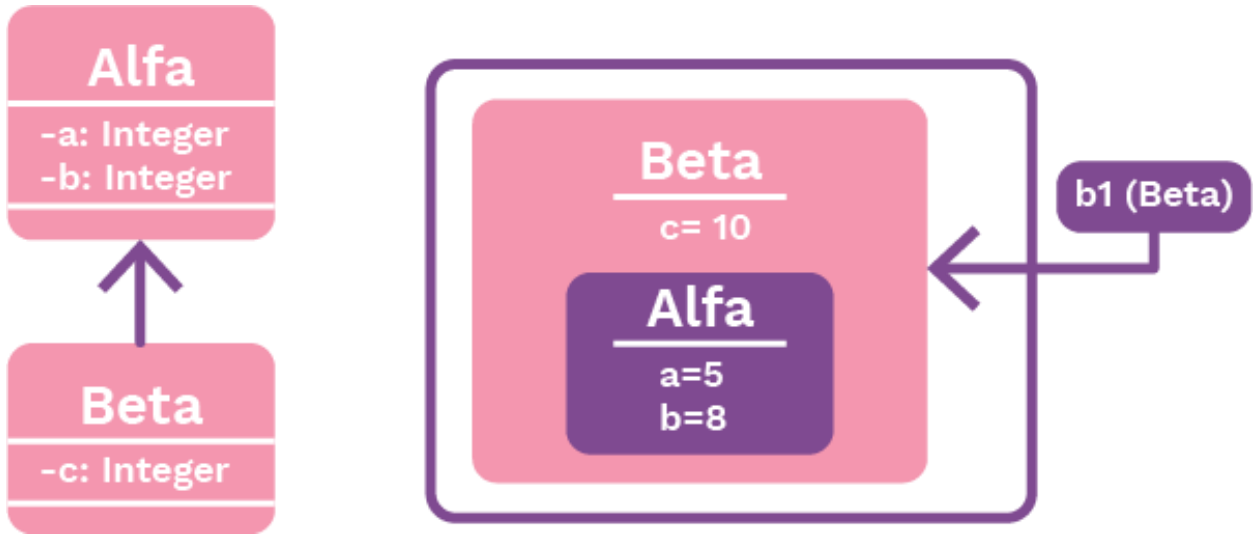


3.5.2. Encadenamiento de Constructores

El uso de **super** en las clases heredadas permite que **las clases hijas puedan hacer uso del constructor padre,** solicitando atributos o agregando instrucciones adicionales, como se muestra en la imagen Beta , la cual contiene instrucciones e información que estaba preestablecida en Alfa.



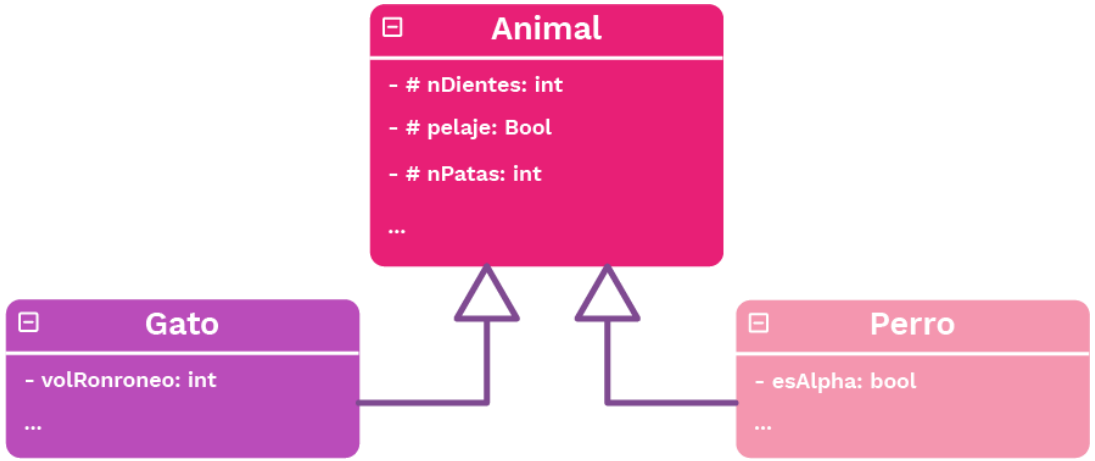
Representación en UML



3.6 Polimorfismo

En los lenguajes fuertemente tipados, una variable siempre deberá apuntar a un objeto de la clase que se indicó en el momento de su declaración.

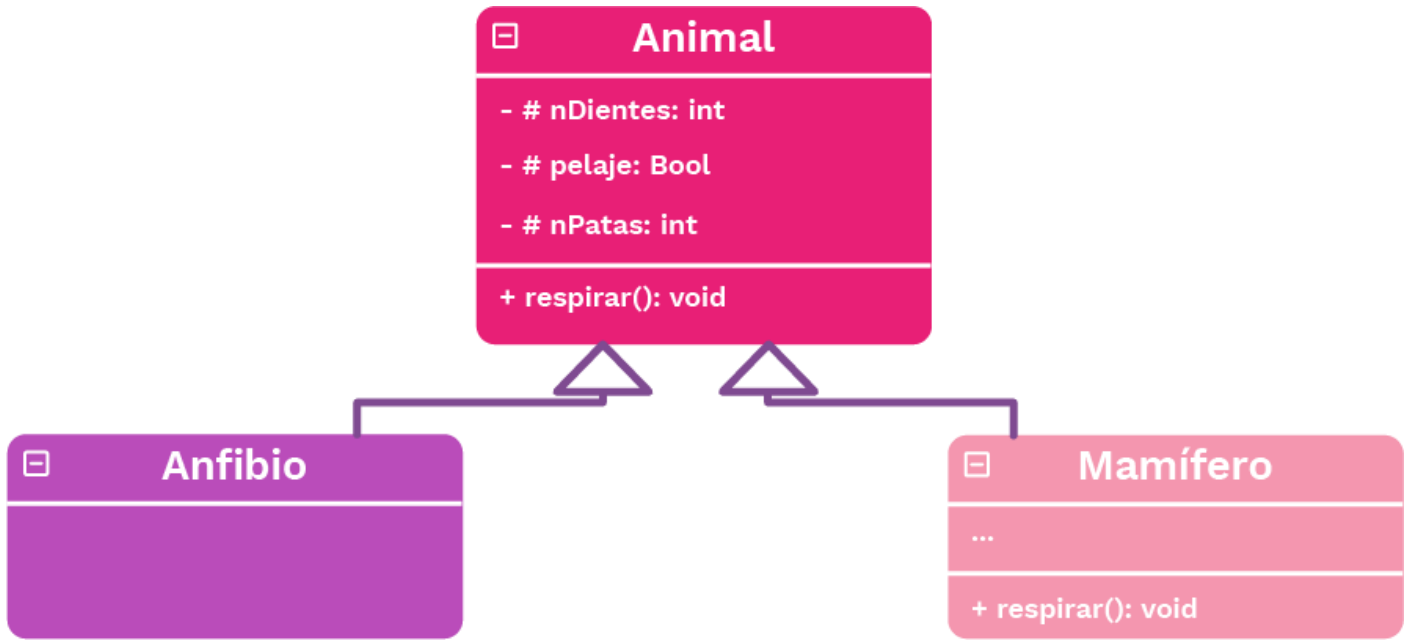
El **polimorfismo** es una flexibilización del sistema de tipos; de tal manera que, una referencia a una clase (atributo, parámetro o declaración local o elemento de un vector) acepta direcciones de objetos de dicha clase y de sus clases derivadas (hijas, nietas, ...).



3.7. Sobreescritura de métodos

La **sobreescritura de métodos** está estrechamente ligada a la herencia y se usa, generalmente, en conjunto con el polimorfismo. Este concepto consiste en algo muy simple: **Sobrescribir un método de una superclase en una subclase**.

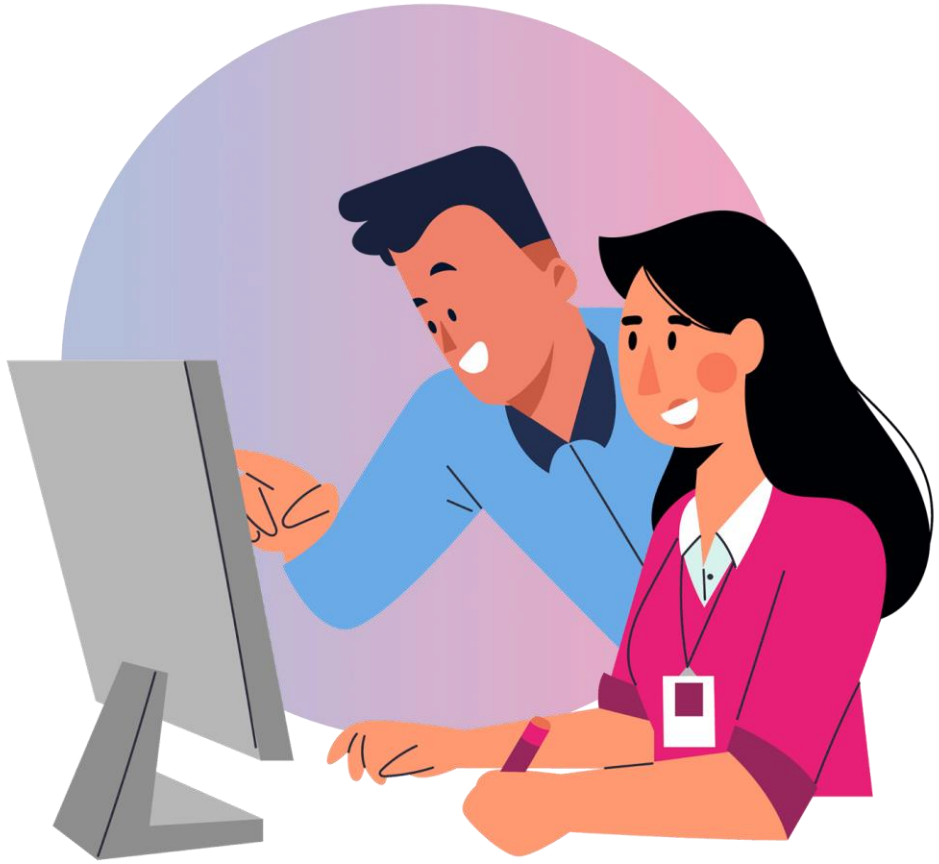
Representación en UML de sobreescritura de métodos

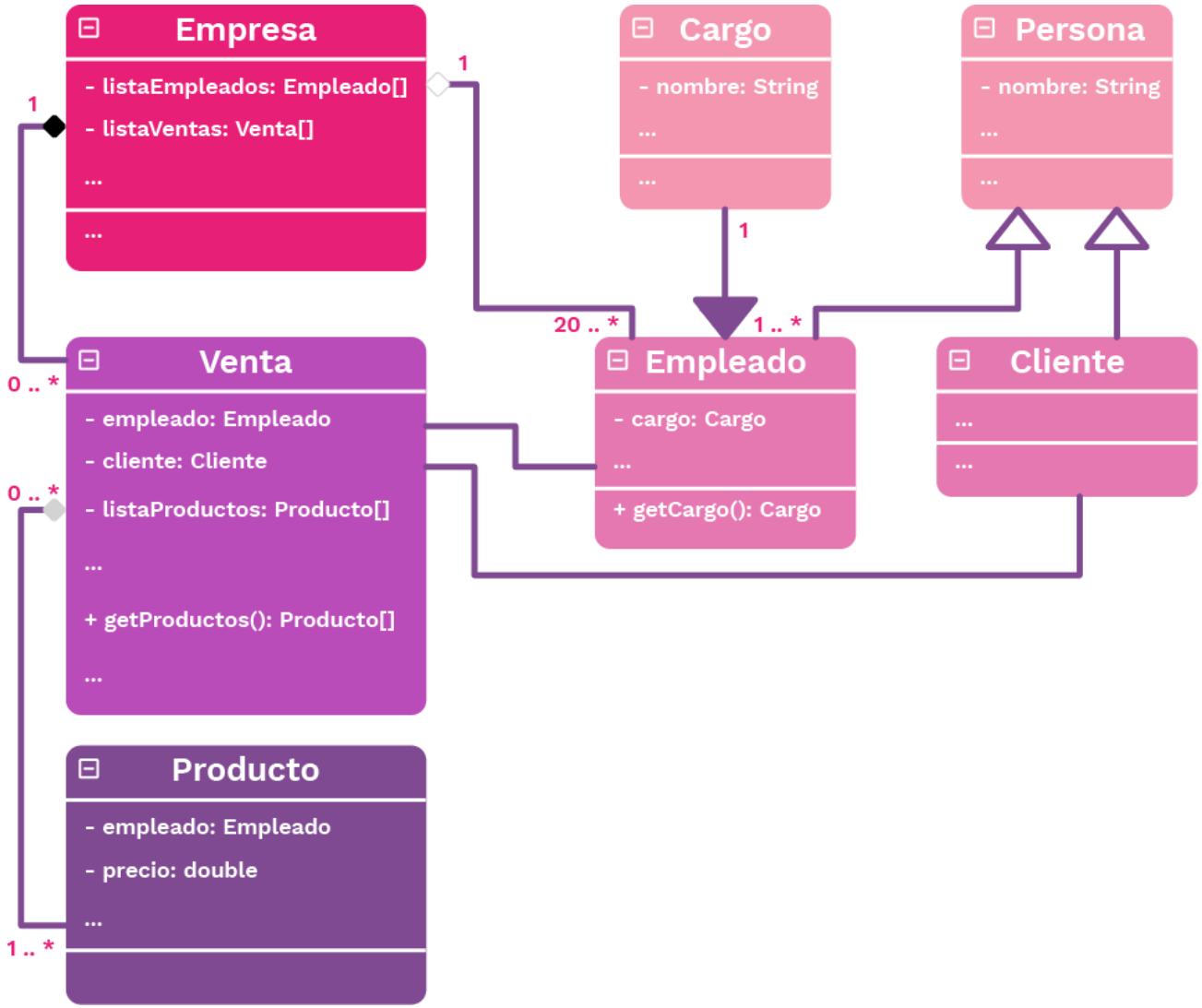


4. Modelado estructural

Este es donde se describen los tipos de objetos que están presentes en un sistema, las relaciones que existen entre ellos (Clases, Interfaces, Relaciones entre clases, Paquetes, Cardinalidad, entre otros).

La forma correcta de presentar este modelado, es a través del diagrama de clase, ya que está es la representación gráfica.




Empresa
- listaEmpleados: Empleado[]
- listaVentas: Venta[]
...
...
Cargo
- nombre: String
...
...
Persona
- nombre: String
...
...
Venta
- empleado: Empleado
- cliente: Cliente
- listaProductos: Producto[]
...
+ getProductos(): Producto[]
...
Empleado
- cargo: Cargo
...
+ getCargo(): Cargo
Cliente
...
...
Producto
- empleado: Empleado
- precio: double
...
1
1
1
20 .. *
1 .. *
0 .. *
0 .. *
1 .. *

Revisión de lo aprendido

1. Cuando se indica que los objetos de distintas clases que pertenecen a una misma jerarquía pueden tratarse de una forma general e individualizada, al mismo tiempo hablamos de:

    a. Todas de las Anteriores
    b. Polimorfismo
    c. Herencia
    d. Encapsulación

Revisión de lo aprendido

2. Concepto que permite proteger a los atributos que conforman al objeto, y permite o niega información al público. La estructura de un objeto se encuentra oculta y la implantación de sus métodos es visible.

a. Jerarquía
b. Modularidad
c. Polimorfismo
d. Encapsulación

Revisión de lo aprendido

3. En la Programación Orientada a Objetos, ¿A qué se le denomina herencia?

a. Cuando un Objeto recibe datos de la Clase.
b. Al listado de objetos que ha creado una misma clase.
c. La propiedad para que el software desarrollado pueda ser utilizado varias veces.
d. Es cuando un Objeto se puede construir con base en otro objeto.

Revisión de lo aprendido

4. La palabra clave **super** permite:

a. Escanear palabras del teclado.
b. Hacer llamado a la versión original del método en la superclase.
c. Insertar nuevos elementos al objeto.
d. Eliminar objetos creados.

El futuro digital es de todos

MinTIC

Hechos QUE CONECTAN

CICLO 2

EJE TEMÁTICO 3

RELACIÓN ENTRE CLASES

Universidad Industrial de Santander

MisionTIC 2022