



UML

Diagrama de Clases

01 {

[UML]

- Lenguaje Unificado de Modelado (UML)
- Diagrama de Clases
- Clase
- Visibilidad

}

Lenguaje Unificado de Modelado {

El **Lenguaje de Modelado Unificado(UML)** es un estándar visual de **modelado utilizado para modelar negocios y procesos similares**, así como para el análisis, diseño e implementación de sistemas basados en software.

UML es un lenguaje en común para arquitectos de software, desarrolladores y otros, usado para **describir, especificar, diseñar y documentar artefactos de software** u otros artefactos del negocio.

}

Lenguaje Unificado de Modelado {

Entonces, el **Lenguaje Unificado de Modelado (UML)** puede ayudarte a modelar sistemas de diversas formas.

Uno de los tipos más populares en el UML es el **Diagrama de Clases**. Los diagramas de clases son un tipo de **diagrama de estructura** porque describen lo que debe estar presente en el sistema que se está modelando.

}

Lenguaje Unificado de Modelado {

El **UML** se estableció como un modelo estandarizado para describir un enfoque de **Programación Orientada a Objetos (POO)**. Como las **clases** son los componentes básicos de los **objetos**, los **diagramas de clases** son los componentes básicos del **UML**.

Los diversos **componentes en un diagrama** de clases pueden representar las **clases que se programaran** en realidad, los objetos principales o la interacción entre clases y objetos.

}

Diagrama de Clases {

El **diagrama de clases** es un **diagrama UML** que muestra la estructura del sistema diseñado a nivel de clases, muestra sus características, restricciones y relaciones: asociaciones, generalizaciones, dependencias, etc.

En este diagrama, el elemento principal son las **clases**.

Una **clase** describe un **conjunto de objetos** que **comparten las mismas características y restricciones**.

}

Diagrama de Clases {

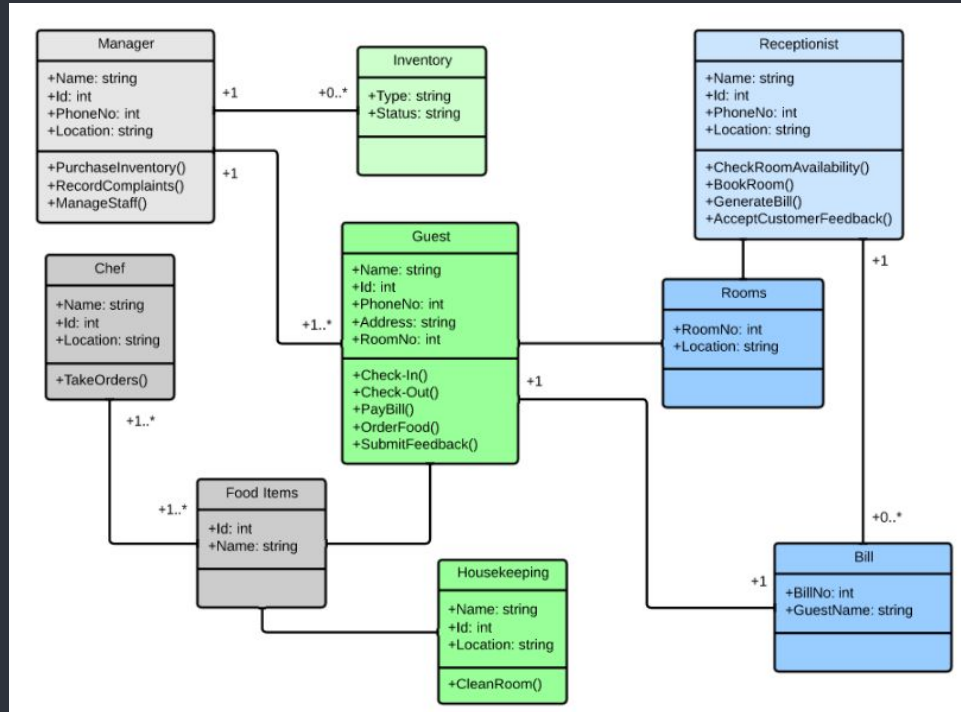
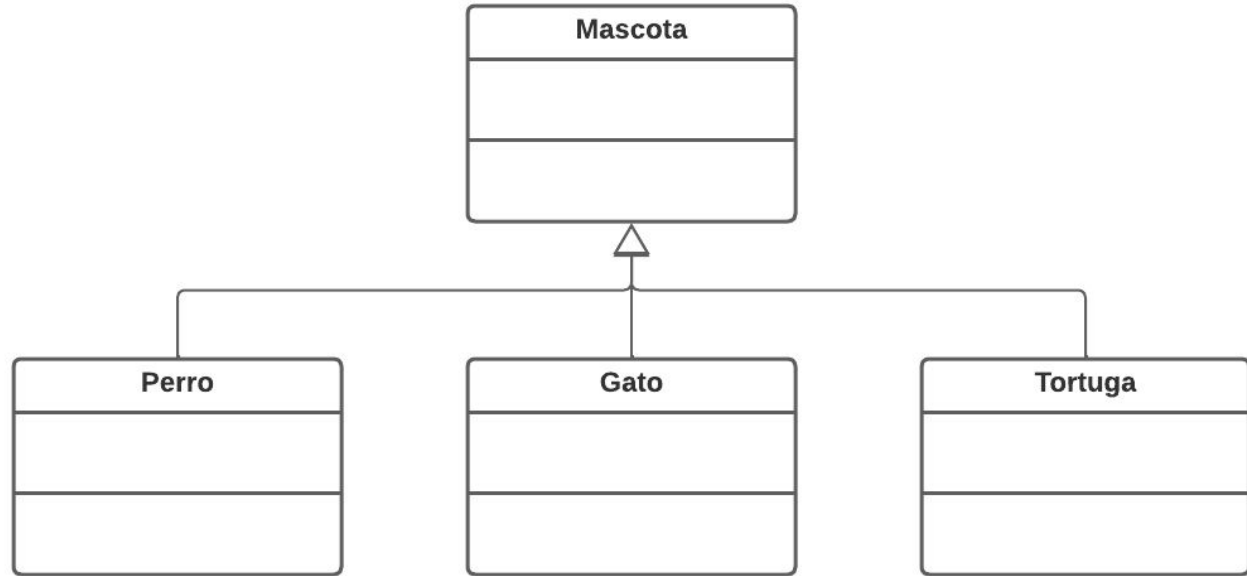


Diagrama de Clases {




```
1 Herramientas {
2
3   |
4
5
6   LucidChart
7
8   Miro
9
10
11
12
13 }
14
```

Clase {

Podemos representar cada **clase** como un **rectángulo de tres filas**:

1. La **fila superior** contiene el **nombre de la clase**. El nombre de la clase debe estar **centrado** y en **negrita**, con la primera letra del nombre de la clase en **mayúscula**.
2. La **fila del centro** contiene los **atributos de la clase**.
3. La **fila inferior** expresa los métodos o las operaciones que la clase puede utilizar.

Las clases y las subclases se agrupan para mostrar la **relación entre cada objeto**.

}

```
1 Clase {
```

```
2  
3  
4  
5  
6  
7  
8  
9  
10  
11
```

```
12  
13 }  
14
```



Clase {

Como se puede apreciar, **las clases en UML tienen mucha similitud con las clases en la Programación Orientada a Objetos (POO)**, por lo que estos diagramas nos permitirán expresar nuestros sistemas de forma detallada y precisa.

}

Visibilidad {

Aunque en Python no tenemos el concepto de **visibilidad**, en UML, si existe y es importante que entendamos cómo leerlo.

Todos los **atributos o métodos** que sean **públicos** (**todos en Python**) se representa con un “+” antes de su nombre.

Mientras que los **privados** (en Python la convención es nombrar a los atributos o los método con “__” al inicio) se presentan con un “-” antes de su nombre

}

Modificadores de acceso {

Todas las clases poseen diferentes niveles de acceso en función del **modificador de acceso** (visibilidad):

- **Público (+)**
- **Privado (-)**
- Protegido (#)
- Paquete (~)
- Derivado (/)
- Estático (subrayado)

}

Multiplicidad {

La **multiplicidad** de una asociación determina cuántos objetos de cada tipo intervienen en la relación: El **número de instancias de una clase que se relacionan con una instancia de la otra clase**.

Cada asociación tiene dos multiplicidades (una para cada extremo de la relación).

}

Multiplicidad {

La **multiplicidad** es una **definición de cardinalidad**, es decir, **número de elementos de una colección**, se representa utilizando un **intervalo de enteros no negativos** para especificar el número permitido de instancias del elemento descrito.

El intervalo de multiplicidad tiene un **límite inferior** y un **límite superior**.

}

Multiplicidad {

Tenemos los siguientes tipos de multiplicidad:

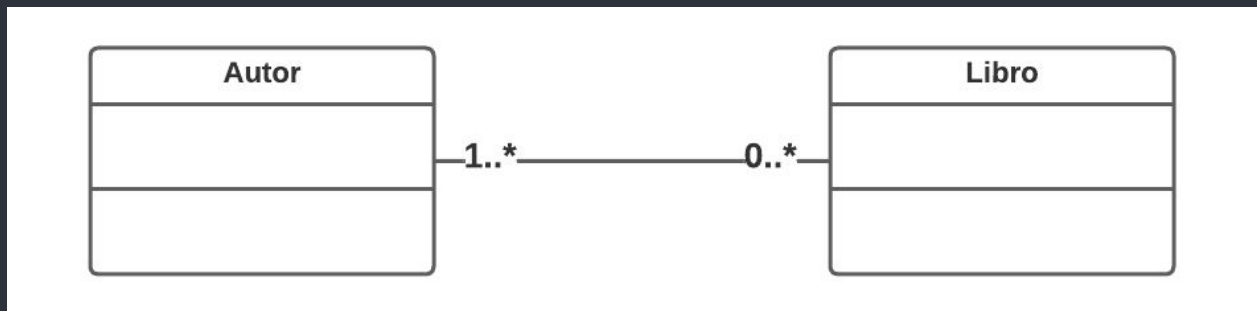
- **1**: No mas de uno.
- **0..1**: Cero o uno.
- **0..***: Cero o muchos.
- **1..***: Uno o muchos.
- *****: Muchos.

}

Multiplicidad {

Multiplicity	Opción	Cardinalidad / Multiplicidad
0..0	0	La colección debe estar vacía
0..1		Ninguna instancia o una
1..1	1	Exactamente 1 instancia
0..*	*	Cero o más instancias
1..*		Al menos una instancia
5..5	5	Exactamente 5 instancias
m..n		Al menos m pero no más de n instancias

Multiplicidad {



Aquí podemos ver un uso de cardinalidad entre dos clases, esta nos dice que **1.. o más autores**, pueden ser autores de **0.. o más libros**.

}

Multiplicidad {

Estas relaciones pueden ser difíciles de visualizar de vez en cuando, por lo que podemos tratar de ver **casos particulares** para poder comprender mejor sus limitaciones:

¿Cuántos autores puede tener un libro?

Dependiendo del tipo de libro, puede haber sido escrito por una persona, o puede haber sido un trabajo en equipo en el que varios expertos participaron en su redacción. Por lo tanto, un libro puede tener **1.. o más autores**.

}

Multiplicidad {

¿Cuántos libros puede tener un autor?

Dependiendo de la persona, puede que no le interese mucho participar en este tipo de actividades, por lo que podría escribir **0 libros en su vida.**

Sin embargo, otro autor con más disposición podría haber escrito **varios libros en su vida**, por lo que el rango de posibilidades en este caso quedaría entre **0 o más libros.**

}

Asociación {

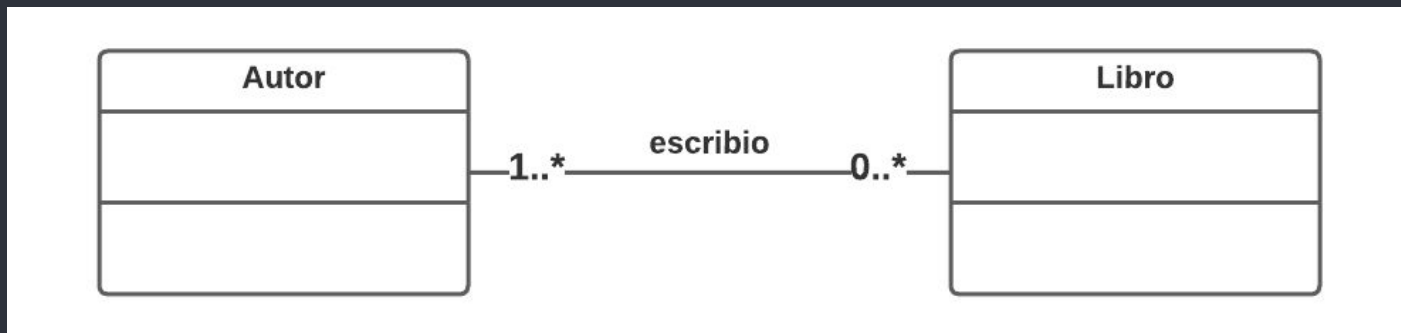
La relación predeterminada entre dos clases. **Ambas clases están conscientes una de la otra** y de la relación que tienen entre sí.

La asociación en un diagrama de clases UML permite realizar **conexiones lógicas**.

Esta asociación se representa mediante una **línea recta entre dos clases**. El **nombre de la asociación** se escribe en la línea. La **multiplicidad** de la asociación se indica escribiendo el número de objetos que pueden participar en la relación **en cada extremo de la línea**.

En resumen, una asociación es una **relación estructural que describe una conexión entre objetos**.

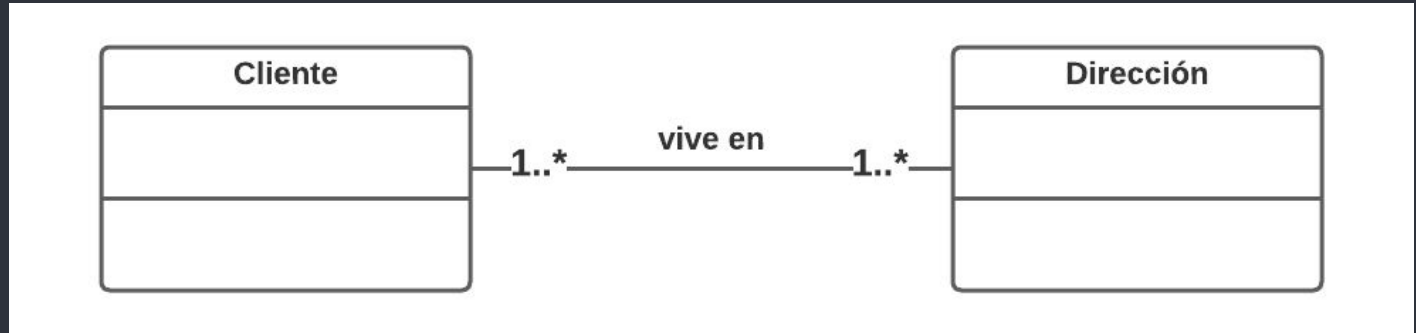
Asociación {



Aquí podemos ver la **relación lógica** entre autor y libro, indicando que **1.. o más autores escribieron 0.. o más libros.**

}

Asociación {



Aquí podemos ver la **relación lógica** entre cliente y dirección, indicando que **1.. o más clientes viven en 1.. o más direcciones**.

}

Agregación {

En **UML**, la **agregación** es una relación entre objetos de diferentes clases en la que **un objeto es una colección de otros objetos. Los objetos agregados pueden existir independientemente del objeto que los contiene.**

La **agregación** se representa en un diagrama de clases UML como una **línea con un diamante blanco en el extremo del objeto agregado.** El nombre de la **agregación** se escribe en la línea.

La **multiplicidad** de la agregación se indica escribiendo el número de objetos que pueden participar en la relación en cada extremo de la línea.

}

Agregación {

La **agregación** es un tipo de asociación que indica que **una clase es parte de otra clase** (composición débil).

Los componentes pueden ser compartidos por varios compuestos (de la misma asociación de agregación o de varias asociaciones de agregación distintas). La destrucción del compuesto no conlleva la destrucción de los componentes.

La agregación se representa en UML mediante un diamante de color blanco colocado en el extremo en el que está la clase que representa el “todo”.

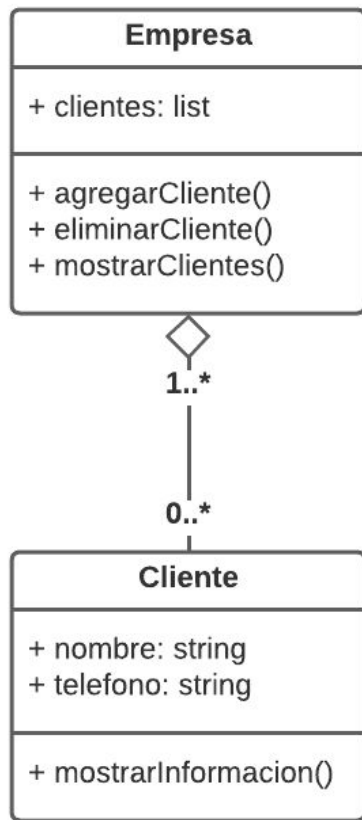
}

Agregación {

- Tenemos una **clase Empresa**.
- Tenemos una **clase Cliente**.
- Una **empresa agrupa a varios clientes**.

Eliminar una empresa no implica eliminar a los clientes de esa empresa. Podemos eliminar la empresa pero **los clientes siguen siendo clientes** (consumidores).

}



Composición {

En UML, la **composición** es una **relación** entre **objetos de diferentes clases** en la que **un objeto es una colección de otros objetos**, y **los objetos agregados no pueden existir independientemente del objeto que los contiene**.

En otras palabras, si el objeto contenedor se destruye, **los objetos agregados también se destruyen**.

}

Composición {

La **composición** es una forma fuerte de composición donde la vida de la **clase contenida** debe coincidir con la vida de la **clase contenedor**.

El símbolo de composición es un **diamante de color negro colocado en el extremo en el que está la clase que representa el "todo"** (compuesto). El **nombre de la composición** se escribe en la línea. La **multiplicidad** de la composición se indica escribiendo el número de objetos que pueden participar en la relación en cada extremo de la línea.

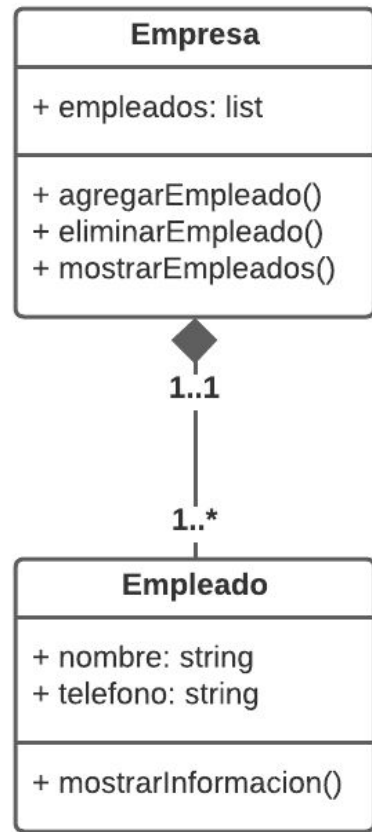
}

Composición {

- Tenemos una **clase Empresa**.
- Tenemos una **clase Empleado**.
- Un **objeto Empresa** está a su vez compuesto por **uno o varios objetos del tipo empleado**.

El tiempo de vida de los **objetos Empleado** depende del tiempo de vida de **Empresa**, ya que **si no existe una Empresa no pueden existir sus empleados**.

}

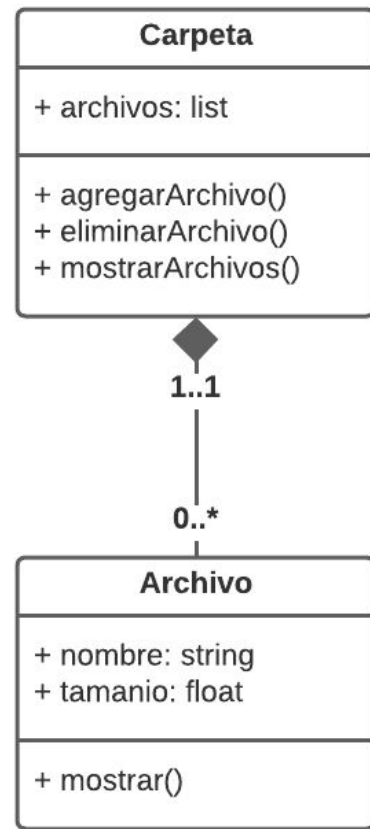


Composición {

- Tenemos una **clase Carpeta**.
- Tenemos una **clase Archivo**.
- Un **objeto Carpeta** está a su vez compuesto por **cero o varios objetos del tipo archivo**.

El tiempo de vida de los **objetos Archivo** depende del tiempo de vida de **Carpeta**, ya que **si no existe una Carpeta no pueden existir sus archivos**.

}



Agregación vs Composición {

En líneas generales, como hemos visto, se podría decir que **la diferencia entre agregación y composición es conceptual**, no se diferencia por código, o al menos, en el mayor de los casos y en la mayoría de los lenguajes de programación.

En definitiva, UML nos permite la posibilidad de diferenciar este tipo de asociaciones con el fin de que, aquella persona que le interese, pueda estipular de una u otra manera que se trata de una composición o una agregación, aunque en términos de implementación no se diferencie tan apenas su uso ni tenga tanta relevancia.

}

Herencia {

En UML, la **herencia** o **generalización** es una relación entre dos clases en la que una clase (hijo) hereda los atributos y métodos de otra clase (padre).

La **clase hijo** puede agregar sus propios atributos y métodos, pero también tiene acceso a todos los atributos y métodos de la **clase padre**.

La herencia se representa en un diagrama de clases UML como una **línea con un triángulo en el extremo de la clase padre**.

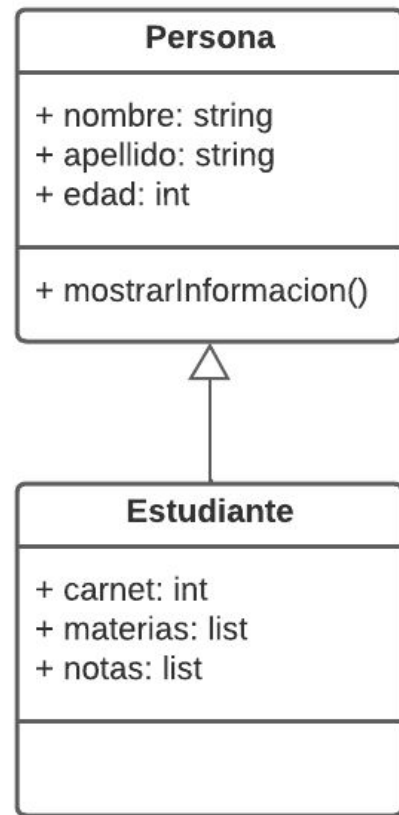
}

Herencia {

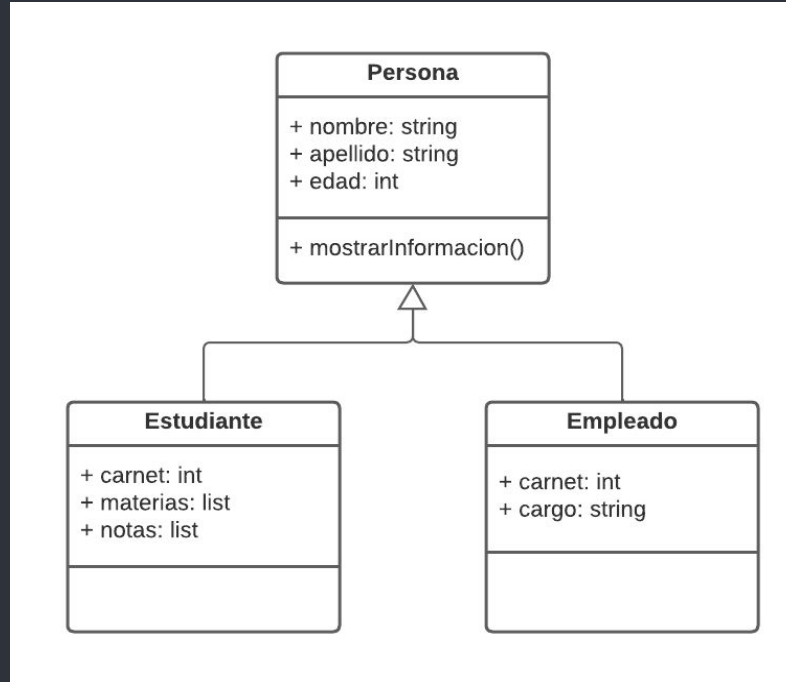
- Tenemos una **clase padre Persona**.
- Tenemos una **clase hijo Estudiante**.

La **clase hijo Estudiante** hereda los atributos y los métodos de la **clase padre Persona**.

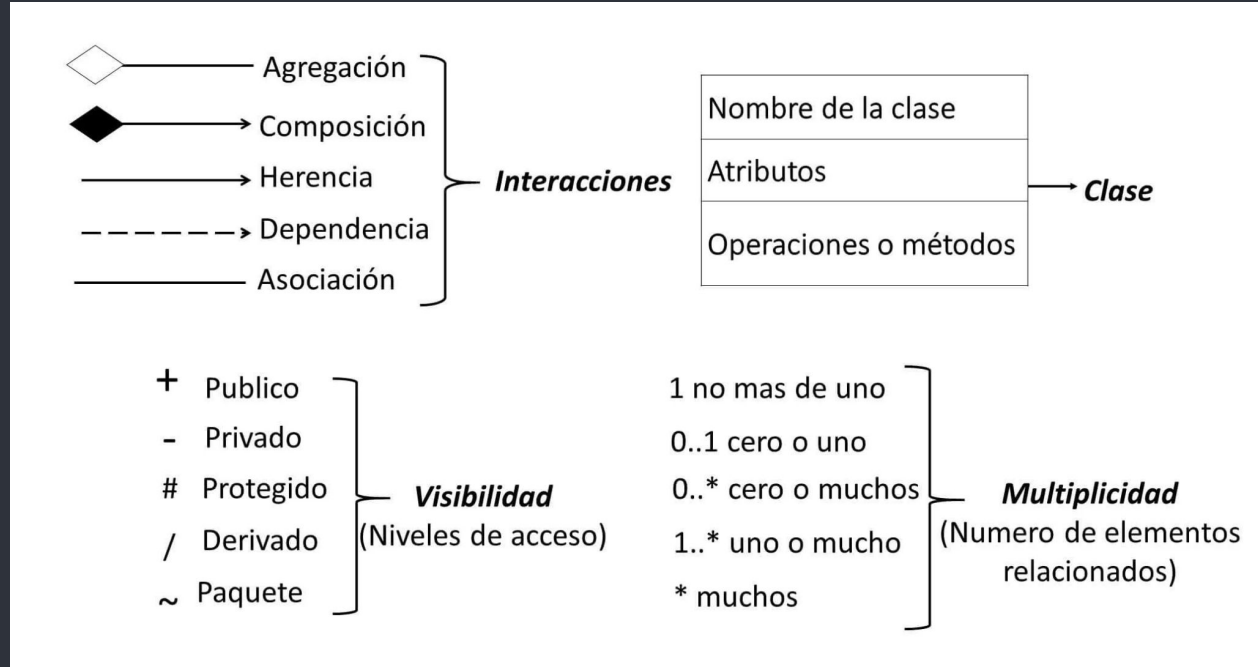
}



Herencia {



Resumen {



Ejercicio {

Crea un diagrama de clases para un sistema de gestión de biblioteca. El sistema debe tener las siguientes clases:

- **Libro**
- **Autor**
- **Bibliotecario**
- **Usuario**
- **Préstamo**

}

Ejercicio {

- Cada **libro** debe tener un título, un autor, un año de publicación y un número de ISBN.
- Cada **autor** debe tener un nombre, una fecha de nacimiento y un país de origen.
- Cada **bibliotecario** debe tener un nombre, una dirección, un número de teléfono y un correo electrónico.
- Cada **usuario** debe tener un nombre, una dirección, un número de teléfono y un correo electrónico.
- Cada **préstamo** debe tener un libro, un usuario, una fecha de préstamo y una fecha de devolución.
- **El diagrama debe mostrar las relaciones entre las clases.** Por ejemplo, un libro puede ser prestado por muchos usuarios, pero un usuario sólo puede tener un libro prestado a la vez.



Recursividad

Funciones que se llaman a sí mismas

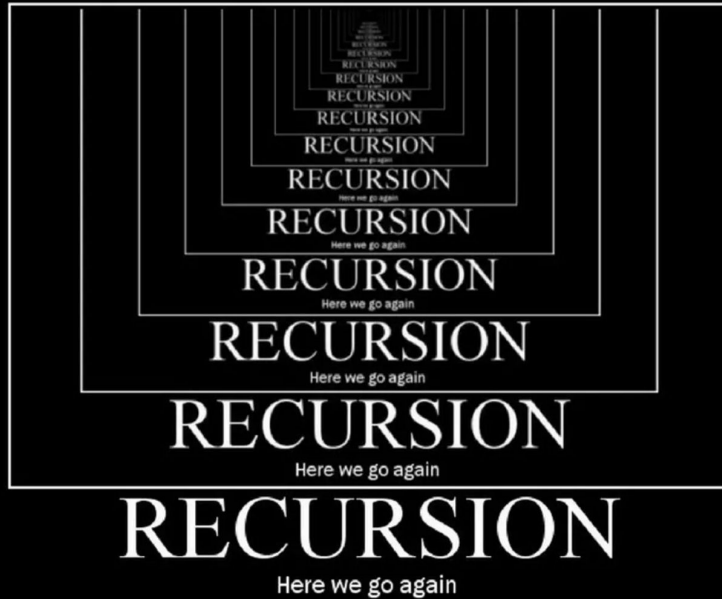
02 {

[Recursividad]

- Recursividad
- Ejemplos

}

Recursividad {



Recursividad {

¿En qué trabajas?

Estoy intentando arreglar los problemas que creé
cuando intentaba arreglar los problemas que creé cuando
intentaba arreglar los problemas que creé.

Y así nació la recursividad.

}

Recursividad {

La **recursividad** es un concepto que proviene de las matemáticas, y que aplicado al mundo de la programación nos permite **resolver problemas o tareas donde las mismas pueden ser divididas en subtareas cuya funcionalidad es la misma.**

Dado que los **subproblemas** a resolver son de la misma naturaleza, se puede usar la misma función para resolverlos.

Dicho de otra manera, **una función recursiva es aquella que está definida en función de sí misma, por lo que se llama repetidamente a sí misma** hasta llegar a un **punto de quiebre.**

}

Recursividad {

Se conoce como **recursividad** cuando **una función se llama a sí misma**. La forma en cómo actúan las funciones recursivas simulan a un **ciclo** y al igual que los ciclos **pueden terminar siendo infinitas** y llevando a un error.

Este proceso se repetiría indefinidamente si no se detiene por alguna condición. Esta condición se conoce como **condición de quiebre**.

Una **condición de quiebre es obligatoria en todas las funciones recursivas**; de lo contrario, seguirá ejecutándose para siempre como un **bucle infinito**.


}

Recursividad {

1. La **función recursiva** es llamada por algún **código externo**.
2. La función recursiva comienza por verificar si se cumple una condición de quiebre.
 - a. **Si se cumple la condición de quiebre**, la función finaliza.
 - b. **Si no se cumple la condición de quiebre**, la función realiza el procesamiento requerido y luego se llama a sí misma para continuar la recursión.

}

Función iterativa 1 {



```
def vaciar_lista_iterativo(lista):  
    inicio = len(lista) - 1 # Inicia en tamaño - 1  
    fin = -1 # No se incluye así que llega hasta 0  
    step = -1 # Va de 1 en 1 decreciendo  
    for i in range(inicio, fin, step):  
        print("Posicion donde se eliminara: ", i)  
        print("Listas ANTES: ", lista)  
        lista.pop(i)  
        print("ListasDESPUES: ", lista, "\n")
```

```
lista = [1, 2, 3, 4]
```



```
def vaciar_lista_iterativo(lista):  
    inicio = len(lista) - 1 # Inicia en tamaño - 1  
    fin = -1 # No se incluye así que llega hasta 0  
    step = -1 # Va de 1 en 1 decreciendo  
    for i in range(inicio, fin, step):  
        print("Posicion donde se eliminara: ", i)  
        print("Listas ANTES: ", lista)  
        lista.pop(i)  
        print("ListasDESPUES: ", lista, "\n")
```

i	ANTES	DESPUES
3	[1, 2, 3, 4]	[1, 2, 3]
2	[1, 2, 3]	[1, 2]
1	[1, 2]	[1]
0	[1]	[]

Función recursiva 1 {

```
def vaciar_lista_recursivo(lista, llamada):
    print("\nllamada numero: ", llamada)
    print("Lista que recibe la llamada: ", lista)
    # Mientras la lista este llena, eliminamos
    if(len(lista) != 0):
        # Eliminamos el elemento final
        lista.pop()
        # Llamamos de nuevo la funcion
        print(f"Llamamos a la funcion de nuevo. PAUSA de la llamada: {llamada}.")
        siguiente_llamada = llamada + 1
        vaciar_lista_recursivo(lista, siguiente_llamada) # PAUSA por llamada
        print(f"PLAY de la llamada: {llamada}.")
    # Una vez la lista esta vacia, se lo indicamos al usuario
    else:
        print("\nPUNTO DE QUIEBRE ENCONTRADO")
        print("La lista esta vacia.")
        print(f"FINALIZA la llamada: {llamada}.\n")
```


lista = [1, 2, 3, 4]

llamada = 1



```
def vaciar_lista_recursivo(lista, llamada):
    print("\nllamada numero: ", llamada)
    print("Lista que recibe la llamada: ", lista)
    # Mientras la lista este llena, eliminamos
    if(len(lista) != 0):
        # Eliminamos el elemento final
        lista.pop()
        # Llamamos de nuevo la funcion
        print(f"Llamamos a la funcion de nuevo. PAUSA de la llamada: {llamada}.")
        siguiente_llamada = llamada + 1
        vaciar_lista_recursivo(lista, siguiente_llamada) # PAUSA por llamada
        print(f"PLAY de la llamada: {llamada}.")
    # Una vez la lista esta vacia, se lo indicamos al usuario
    else:
        print("\nPUNTO DE QUIEBRE ENCONTRADO")
        print("La lista esta vacia.")
    print(f"FINALIZA la llamada: {llamada}.\n")
```

LLAMADA	RECIBE	RESULTA
1	[1, 2, 3, 4]	[1, 2, 3]
2	[1, 2, 3]	[1, 2]
3	[1, 2]	[1]
4	[1]	[]

Función iterativa 2 {



```
def sumatoria_iterativo(numero, total):  
    inicio = numero # Inicia en numero  
    fin = 0 # No se incluye asi que llega hasta 0  
    step = -1 # Va de 1 en 1 decreciendo  
    total = 0 # Resultado suma  
    for i in range(inicio, fin, step):  
        print("Total ANTES: ", total)  
        print("Valor a sumar: ", i)  
        total += i # Sumamos i al total  
        print("Total DESPUES: ", total, "\n")  
    return total
```

numero = 4

4 + 3 + 2 + 1 + 0 = 10



```
def sumatoria_iterativo(numero, total):
    inicio = numero # Inicia en numero
    fin = 0 # No se incluye asi que llega hasta 0
    step = -1 # Va de 1 en 1 decreciendo
    total = 0 # Resultado suma
    for i in range(inicio, fin, step):
        print("Total ANTES: ", total)
        print("Valor a sumar: ", i)
        total += i # Sumamos i al total
        print("Total DESPUES: ", total, "\n")
    return total
```

i	ANTES	DESPUES
4	0	4
3	4	7
2	7	9
1	9	10
0	10	10

Función recursiva 2 {



```
def sumatoria_recursivo(numero, llamada):
    print("\nllamada numero: ", llamada)
    print(" Numero que recibe la llamada: ", numero)
    if numero == 0:
        print("\nPUNTO DE QUIEBRE ENCONTRADO")
        print("Numero es 0.")
        print("Nos empezamos a regresar.")
        return 0
    else:
        llamada += 1
        siguiente_numero = numero - 1
        return numero + sumatoria_recursivo(siguiente_numero, llamada)
```

numero = 4

4 + 3 + 2 + 1 + 0 = 10

```
def sumatoria_recursivo(numero, llamada):
    print("\nllamada numero: ", llamada)
    print(" Numero que recibe la llamada: ", numero)
    if numero == 0:
        print("\nPUNTO DE QUIEBRE ENCONTRADO")
        print("Numero es 0.")
        print("Nos empezamos a regresar.")
        return 0
    else:
        llamada += 1
        siguiente_numero = numero - 1
        return numero + sumatoria_recursivo(siguiente_numero, llamada)
```

numero	RETORNO
4	4 + sumatoria(3)
3	3 + sumatoria(2)
2	2 + sumatoria(1)
1	1 + sumatoria(0)
0	0



numero = 4

4 + 3 + 2 + 1 + 0 = 10



```
def sumatoria_recursivo(numero, llamada):
    print("\nllamada numero: ", llamada)
    print(" Numero que recibe la llamada: ", numero)
    if numero == 0:
        print("\nPUNTO DE QUIEBRE ENCONTRADO")
        print("Numero es 0.")
        print("Nos empezamos a regresar.")
        return 0
    else:
        llamada += 1
        siguiente_numero = numero - 1
        return numero + sumatoria_recursivo(siguiente_numero, llamada)
```

numero	RETORNO	RESULTADO
4	4 + sumatoria(3)	4 + 3 + 2 + 1 + 0
3	3 + sumatoria(2)	3 + 2 + 1 + 0
2	2 + sumatoria(1)	2 + 1 + 0
1	1 + sumatoria(0)	1 + 0
0	0	0



Función iterativa 3 {



```
def factorial_iterativo(numero, total):  
    inicio = numero # Inicia en numero  
    fin = 0 # No se incluye así que llega hasta 1  
    step = -1 # Va de 1 en 1 decreciendo  
    # total = 1 # Resultado factorial  
    for i in range(inicio, fin, step):  
        print("Total ANTES: ", total)  
        print("Valor a multiplicar: ", i)  
        total *= i # Multiplicamos i por total  
        print("Total DESPUES: ", total, "\n")  
    return total
```

numero = 4

4 * 3 * 2 * 1 = 24



```
def factorial_iterativo(numero, total):
    inicio = numero # Inicia en numero
    fin = 0 # No se incluye asi que llega hasta 1
    step = -1 # Va de 1 en 1 decreciendo
    # total = 1 # Resultado factorial
    for i in range(inicio, fin, step):
        print("Total ANTES: ", total)
        print("Valor a multiplicar: ", i)
        total *= i # Multiplicamos i por total
        print("Total DESPUES: ", total, "\n")
    return total
```

i	ANTES	DESPUES
4	1	4
3	4	12
2	12	24
1	24	24

Función recursiva 3 {



```
def factorial_recursivo(numero, llamada):
    print("\nllamada numero: ", llamada)
    print(" Numero que recibe la llamada: ", numero)
    if numero == 0:
        print("\nPUNTO DE QUIEBRE ENCONTRADO")
        print("Numero es 0.")
        print("Nos empezamos a regresar.")
        return 1
    else:
        llamada += 1
        siguiente_numero = numero - 1
        return numero * factorial_recursivo(siguiente_numero, llamada)
```

numero = 4

4 * 3 * 2 * 1 * 1 = 24



```
def factorial_recursivo(numero, llamada):
    print("\nllamada numero: ", llamada)
    print(" Numero que recibe la llamada: ", numero)
    if numero == 0:
        print("\nPUNTO DE QUIEBRE ENCONTRADO")
        print("Numero es 0.")
        print("Nos empezamos a regresar.")
        return 1
    else:
        llamada += 1
        siguiente_numero = numero - 1
        return numero * factorial_recursivo(siguiente_numero, llamada)
```

numero	RETORNO
4	4 * factorial(3)
3	3 * factorial(2)
2	2 * factorial(1)
1	1 * factorial(0)
0	1



numero = 4

4 * 3 * 2 * 1 * 1 = 24



```
def factorial_recursivo(numero, llamada):
    print("\nllamada numero: ", llamada)
    print(" Numero que recibe la llamada: ", numero)
    if numero == 0:
        print("\nPUNTO DE QUIEBRE ENCONTRADO")
        print("Numero es 0.")
        print("Nos empezamos a regresar.")
        return 1
    else:
        llamada += 1
        siguiente_numero = numero - 1
        return numero * factorial_recursivo(siguiente_numero, llamada)
```

numero	RETORNO	RESULTADO
4	4 * factorial(3)	4 * 3 * 2 * 1 * 1
3	3 * factorial(2)	3 * 2 * 1 * 1
2	2 * factorial(1)	2 * 1 * 1
1	1 * factorial(0)	1 * 1
0	1	1

