TEMA 1 OBJETOS

CONCEPTOS

Un **objeto** es una entidad perfectamente delimitada que encapsula estado y funcionamiento y posee una identidad. Elemento, unidad o entidad individual e identificable, real o abstracta, con un papel bien definido en el dominio del problema.

La clase actúa de molde o plantilla para la creación de objetos. Los objetos que se crean a partir de una clase se llama **instancia**.

Ejemplo con Java:

Lapiz milapiz = new Lapiz (Color.Rojo)

Lapiz tulapiz = new Lapiz (Color.Verde)

Ejemplo con Ruby:

milapiz = Lapiz.new(atributos)

tulapiz = Lapiz.new(atributos)

La **identidad** define la posición de memoria y cada instancia tiene su propia identidad. Objetos distintos residirán en zonas de memoria distintas.

El **estado** de un objeto viene definido por los valores de sus atributos. Cada objeto tiene una zona de memoria propia para el almacenamiento de sus atributos. Los objetos exhiben **comportamiento**, tienen una serie de métodos que pueden ser invocados. Al estado de un objeto no se puede acceder si no es a través de su comportamiento.

PARADIGMA DE PROGAMACIÓN ORIENTADA A OBJETOS (POO)

Paradigma: conjunto de teorías cuyo núcleo central se acepta sin cuestionar y que suministra base y modelo para resolver problemas y avanzar en el conocimiento.

Paradigma de programación: conjunto de reglas que indican como desarrollar software.

Base de la orientación a objetos: se unen datos y procesamiento en entidades → objetos.

Los objetos son las entidades que se manejan en el software. Programar es modelar el problema mediante un universo dinámico de objetos. Cada objeto pertenece a una clase y tiene una responsabilidad dentro de la aplicación.

La funcionalidad del programa se consigue con que unos objetos le pidan a otros objetos (envío de mensajes) que hagan cosas (ejecución de métodos). **Cada objeto se tiene que ocupar de lo suyo y no del trabajo del otro**. El objetivo es obtener **alta**

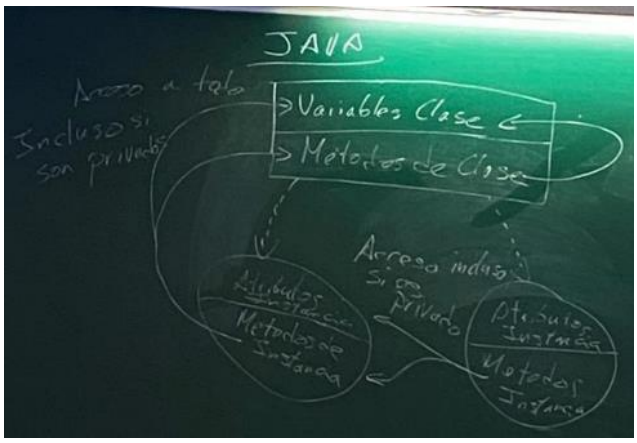
cohesión (métodos similares en muchos aspectos) y **bajo acoplamiento** (cambios que se realizan en un objeto no afecta a los otros objetos que hacen su propio trabajo).

Los principales conceptos básicos son:

- Clase (definido arriba)
- Objeto o instancia (definido arriba)
- Estado (definido arriba)
- Identidad (definido arriba)
- Mensaje (definido arriba)
- Herencia: es el mecanismo por el cual una clase permite heredar las características (atributos y métodos) de otra clase.
- Polimorfismo: para un mismo método pueden existir diferentes implementaciones.
- Ligadura dinámica: se puede invocar un determinado método sin preguntar de que tipo es o qué va a usar.

Las clases:

- *Deben tener una responsabilidad muy concreta*: si una clase hace muchas cosas mejor crear varias clases y repartir tareas
- *Deber ser autónomas*: si una clase se ve afectada por cambios realizados en otras clases, esa clase tiene responsabilidades que no le corresponden.
- *Deben ser introvertidas y no altruistas:* el estado de una clase solo se debe modificar desde la propia clase y ninguna clase debe hacer el trabajo que le corresponde a otra clase.



TEMA 2 CLASES, OBJETOS Y MENSAJES

ATRIBUTOS Y MÉTODOS DE INSTANCIA

Los **atributos de instancia** (objeto) son **variables que están asociadas a cada objeto**. Cada instancia tiene sus propias variables de instancia, cada instancia tendrá los mismos atributos que otra instancia de la misma clase, pero en zona de memoria distintas. El **estado** de cada instancia se describe mediante los valores de estos atributos.

Los **métodos de instancia** son funciones o métodos que están asociados a los objetos de dicha clase. Desde esos métodos son accesibles los atributos de instancia de un objeto.

ATRIBUTOS Y MÉTODOS DE CLASE

Los **atributos de clase** almacenan información que se considera asociada a la propia clase y no a cada instancia. Son *variables globales* a todas las instancias de la clase (*en Java se declaran y definen con* ***static***). Se usan cuando se necesita almacenar información que *siempre va a ser común para todos los objetos de la clase*.

Los **métodos de clase** son funciones asociadas a la propia clase. Acceden / actualizan atributos de clase. NO SE PUEDE ACCEDER DIRECTAMENTE A ATRIBUTOS / MÉTODOS DE INSTANCIA DESDE UN MÉTODO DE CLASE (*en Java se declaran y definen con* ***static*** || *en Ruby si el método tiene* ***self (referencia al propio objeto)*** *se basa en un* ***método de clase****, en caso contrario, se basa en un método de instancia de clase. Los* ***atributos de clase*** *se definen con* ***@@nombre_variable*** *y son accesibles directamente desde el ámbito de instancia y estos atributos se comparten con las subclases (herencia), en cambio los atributos de instancia de la clase no. Los* ***atributos de instancia de clase*** *se definen con* ***@nombre_variable*** *y se puede acceder desde cualquier ámbito de la clase. En una clase, cualquier punto dentro de un método de instancia está en ámbito de instancia, lo demás está en ámbito de clase. En un ámbito de instancia, @variable alude a un atributo de instancia, en un ámbito de clase, @variable alude a un atributo de instancia de clase. Las variables locales y parámetros de método no llevan @. Cuando en un programa hay una línea que es require 'date', es que se necesita el archivo que incluye la clase date*).

PSEUDOVARIABLES

Las palabras que referencian al propio objeto o a clase son:

- Java: this.
- Ruby: self.

ESPECIFICADORES DE ACCESO. VISIBILIDAD

Hay distintos niveles de acceso a atributos y métodos:

- Privado: sin acceso desde otra clase y/o desde otra instancia. Diferencias:
    - Java: se puede acceder a elementos privados *desde una instancia a otra instancia de la misma clase, desde el ámbito de clase a una instancia de clase y desde el ámbito de instancia a la clase de la que se es instancia*.
    - Ruby: todo lo anterior no está permitido. Atributos siempre privados.
- Paquete: sin restricciones en el mismo paquete (en Ruby no). Entre todas las clases que forman parte de un paquete permite acceso a otros métodos.
- Público: sin restricciones de acceso.

Se asigna la visibilidad más restrictiva y PRIVADA para atributos.

- Para los que necesiten ser leídos desde fuera de la clase, se creará un método con visibilidad de paquete o público (según corresponda) que proporcionará dicho atributo al exterior (consultor).
- Si lo que se proporciona es una referencia, el atributo podría ser modificado desde fuera de la clase.
- Para los que necesiten ser modificados desde fuera de la clase, se creará un método con la visibilidad adecuada que reciba los parámetros necesarios y, tras realizar las comprobaciones pertinentes, realice la modificación (modificador).
- Los atributos de un objeto no deberían ser modificados por métodos distintos de los propios de dicho objeto (o de su clase).
- Solo se crearán los consultores y modificadores necesarios, y con la visibilidad más restrictiva que sea posible.

CONSTRUCCIÓN DE OBJETOS

Los constructores son métodos especiales que se encargan solo de inicializar las instancias. No devuelven nada.

JAVA

Es igual que los punteros en C++. Se puede reutilizar un constructor desde otro constructor. Si no se crea ningún constructor, existe uno por defecto sin parámetros. Para construirlo se pone la palabra new al nombre de la clase. Es igual que el constructor en C++. Por ejemplo: clase Persona, pues el constructor sería **Persona a = new Persona**.

RUBY

El equivalente al constructor es un método especial llamado initialize. Es un método de instancia privado que se llama automáticamente por el método new. Se ocupa de la creación e inicialización de atributos de instancia. No se puede sobrecargar initialize. Para tener varios constructores se puede o crear métodos de clase que hagan lo que hacen los constructores como new, o hacer que initialize admita varios parámetros.

En Ruby el puts ------.inspect sirve para imprimir por pantalla una cadena de caracteres. En un método de clase no podemos declarar ni definir variables de instancia de clase.

CONSTRUCTOR DE COPIA

Construye otro objeto recibiendo uno como parámetro. Igual que en C++.

JAVA

```
Compleja (Compleja otro) {


        numeros = new Arraylist()a;

        for (Numero n: otro.numeros) {

                numeros.add(new Numero (n));

        }

}
```

RUBY

También se puede llamar a otro constructor con new().

- attr_reader: Método consultor que nos devuelve el valor de un atributo. Idéntico a:

```
def nombre

        return nombre

end
```

- attr_writer Método modificador que modifica el valor del atributo. Idéntico a:

```
def nombre=(n)

        @nombre = n

end
```

- attr_accessor : x --> Método que se basa tanto como consultor como modificador.

MEMORIA DINÁMICA Y PILA

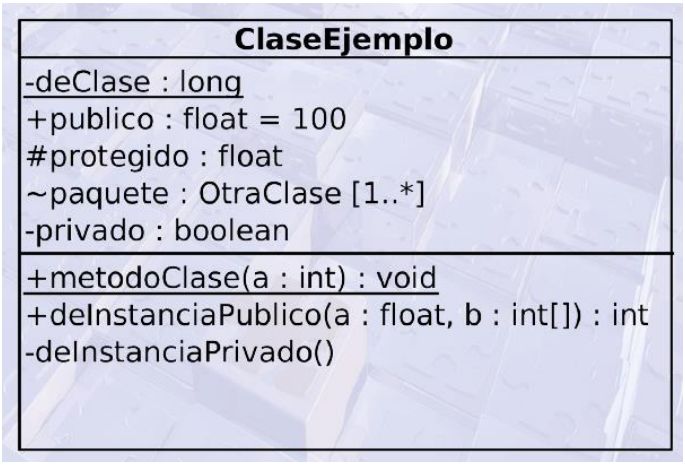
En Java y Ruby todos los objetos se crean en memoria dinámica. Java y Ruby disponen de un recolector de basura que libera automáticamente la memoria utilizada por objetos no referenciados. Funcionamiento del destructor.

UML: DIAGRAMAS ESTRUCTURALES

UML es un "Lenguaje Unificado de Modelado", un lenguaje de diseño y no una metodología. Es independiente del lenguaje de programación a usar. Tiene como objetivo modelar la estructura de un sistema, su comportamiento… Esto permite:

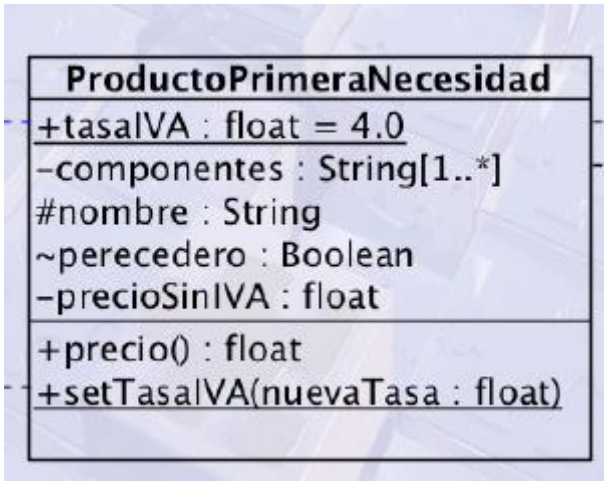
- Especificar mediante modelos las características de un sistema antes de su construcción.
- Visualizar gráficamente un sistema software que sea entendible.
- Documentar un sistema desarrollado para facilitar su mantenimiento, revisión y modificación.

Un diagrama de clases muestra las clases y sus relaciones de tipo asociación, dependencia, generalización y realización.



Las variables / métodos subrayados son:

- En Ruby: variables / métodos de clase.
- En Java: variables / métodos de tipo static.



+ = público ~ = paquete

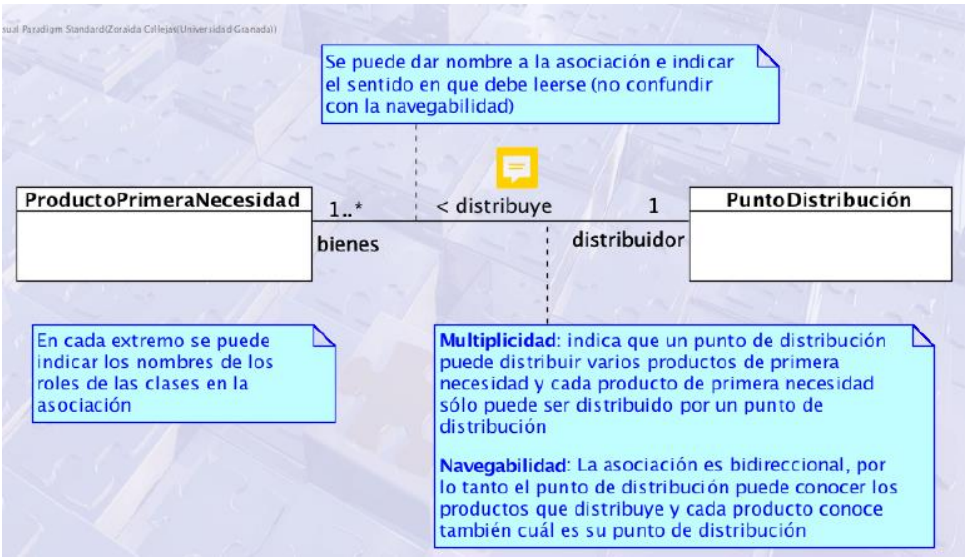
- = privado # = protegido

Ente las distintas relaciones entre clase podemos destacar:

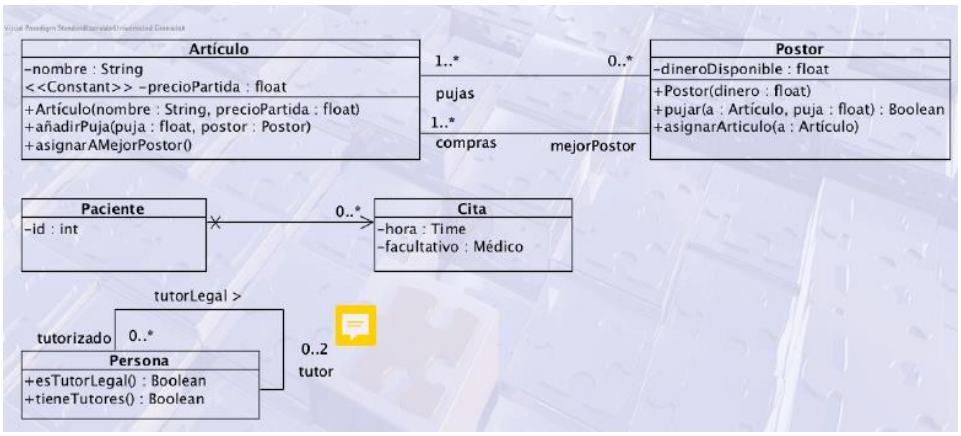
- ASOCIACIÓN: modela relación estructural fuerte y duradera en el tiempo. Las asociaciones generan atributos de referencia. Destaca:

    - *Navegabilidad*: se representa con puntos de flecha. Indica si es posible conocer las instancias relacionadas con la instancia de origen y, si no se indican flechas, por defecto, las relaciones son bidireccionales.

    - *Cardinalidad / multiplicidad*: se representa con números y puede definir un rango. Indica cuántas instancias de clase en un extremo están vinculadas a una instancia de la clase del extremo opuesto. Si no se indica nada, por defecto, su valor es 1.

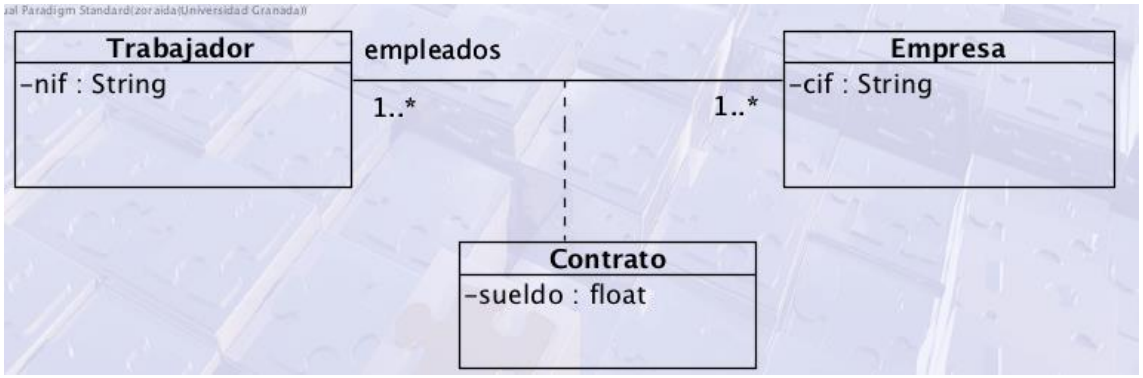


En este caso < distribuye sirve para orientar, sirve de ayuda. En cada extremo se indica los nombres de los roles de las clases en la asociación.



X: no puede ir en otra dirección / Linea continua: relación fuerte / Línea discontinua: relación débil.
0..2: indica que puede haber o 0 o 1 o2 tutores.

Entonces:



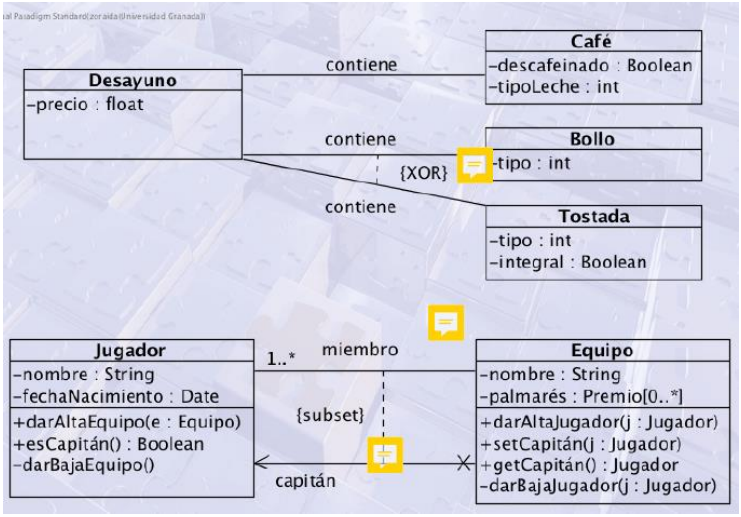
La línea discontinua hacia abajo se denomina ASOCIACIÓN.

Esa manera de los cuadros de forma horizontal se llama RELACIÓN.

De esta manera la clase Trabajador tendría el nif : String y contratos: Contrato [1…*], la clase Contrato tendría sueldo : float , trabajador : Trabajador y empresa : Empresa, mientras que la clase Empresa tendría cif : String y empleados : Contrato [].

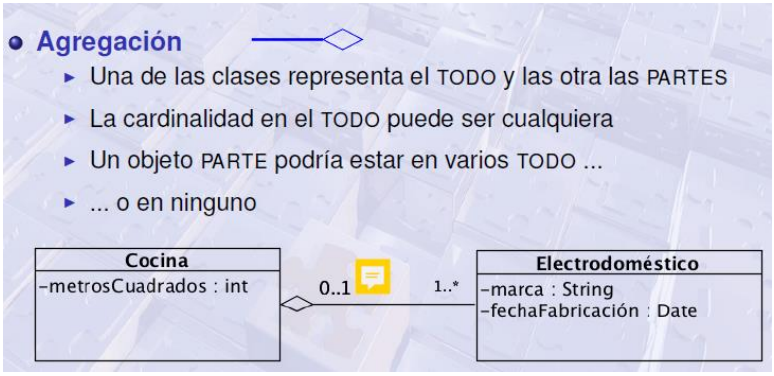
En los extremos de las asociaciones se pueden indicar propiedades. Las más comunes con multiplicidad mayor a 1 son:

- ordered: indica una secuencia ordenada.
- unique: indica elementos no se repiten.

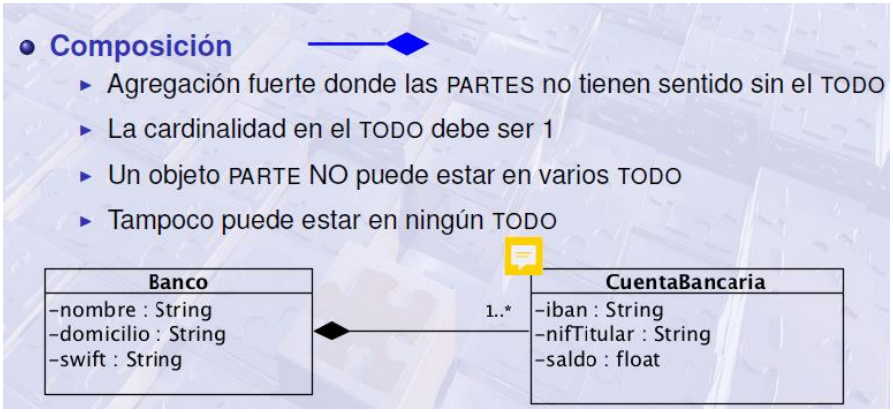


El XOR representa si eljijo o uno u otro.

Un equipo tiene de 1 a más jugadores, luego 1 equipo tiene 1 capitán, subset indica que el capitán debe de ser un miembro del equipo.

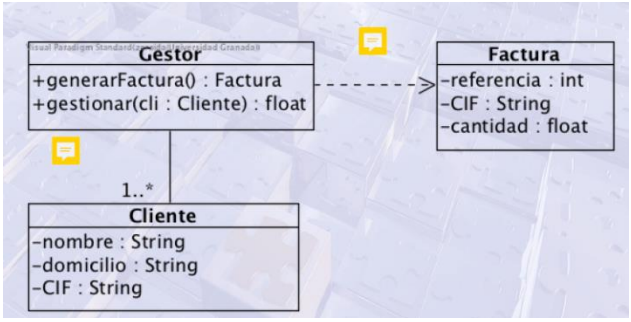


El rombo indica un matiz semántico. Indica que una concina tiene un conjunto de electrodomésticos, y permite que hay electrodomésticos que no estén en ninguna cocina.



Una cuenta bancaria tiene que estar si o si asociada a un banco, no puede haber cuentas bancarias que no estén asociadas a un banco. Una misma cuenta bancaria no puede estar en más de un banco ni en 0 bancos, solo en 1, es decir, cardinalidad = 1.

La dependencia -----→ modela una relación débil y poco duradera en el tiempo. Se usa cuando en una clase se utilizan instancias de otra clase. Se representan con puntos de flecha e indica que una clase utiliza a la otra.



La relación entre Gestor y Factura es una relación de dependencia, en un momento dado Gestor usará un objeto de la clase Factura. Gestor tiene un conjunto de clientes.

Los diagramas de paquetes (agrupaciones) permiten expresar relaciones de dependencia entre paquetes.

CONSULTORES Y MODIFICADORES

Los consultores devuelven el valor de un atributo:

- getAtributo() en Java.
- atributo en Ruby.

Los modificadores se encargan de modificar el valor de un atributo:

- setAtributo(…) en Java.
- Atributo= en Ruby.

En Java, cuando es paquetes se devuelven referencias, en caso contrario se devuelven copias.

Ruby: Consultores y modificadores implícitos

```ruby
class UnaClase

  attr_reader   :atr1
  attr_accessor :atr2
  attr_writer   :atr3

  def initialize (un, dos, tres)
    @atr1 = un
    @atr2 = dos
    @atr3 = tres
  end

end

obj = UnaClase.new(1,2,3)
obj.atr2 = 8
puts obj.inspect
obj.atr2 = 9
puts obj.inspect
obj.atr3 = 7
puts obj.inspect
puts obj.atr1
puts obj.atr2
#puts obj.atr3 # no existe consultor
#obj.atr1 = 23 # no existe modificador
```

ELEMENTOS DE AGRUPACIÓN

Los **paquetes Java** permiten agrupar clases. Se puede tener clases que se llamen igual en distintos paquetes. NO EXISTEN SUBPAQUETES, cada uno es independiente del resto. Ejemplo:

Ejemplo: Paquetes Java

```
1 package miPrograma;
2 // Los elementos que se definan en este archivo pertenecerán al paquete  miPrograma
3 import modelo.Fachada;   // En este fichero se va a usar la clase Fachada del paquete modelo
```

Si pusiéramos import paquete.* incluiríamos todas las clases del paquete.

Paquete: view

```
1 package view;
2 interface Vista {
3   . . .
4 }
```

Paquete: view.gui

```
1 package view.gui;
2 import view.Vista;
3 class VistaGrafica
4       implements Vista {
5   . . .
6 }
```

Paquete: view.tui

```
1 package view.tui;
2 import view.Vista;
3 class VistaTextual
4       implements Vista {
5   . . .
6 }
```

Por ejemplo, en este caso el Paquete view no tiene acceso a las cosas que no sean públicas del paquete view.gui.

Los **módulos Ruby** permiten agrupar varios elementos. Se puede copiar el contenido de un módulo en una clase (include). SI PUEDE HABER MÓDULOS DENTRO DE MÓDULOS. Hay que especificar en Ruby las clases y los archivos que usemos: Cuando se ejecuta en Ruby un archivo, este lo hace línea a línea. Por ejemplo, si en este archivo, se menciona la clase A, debemos haberle indicado a Ruby que previamente haya procesado el archivo que tiene la definición de la clase A.

Esto se hace con instrucciones:

- require: archivos del LENGUAJE.
- require_relative archivos PROPIOS (cuando en ese archivo aparece el nombre de una clase). Por ejemplo, si el fichero se llama cosa.rb y contiene una clase Cosa sería *require_relative 'cosa'*.

Ejemplo:

Objeto = ModuloExterno::ModuloInterno::Clase.new

module Irrgarten

module Directions

UP = up

RIGHT = right

end

end

acceso: Irrgarten::Directions::UP

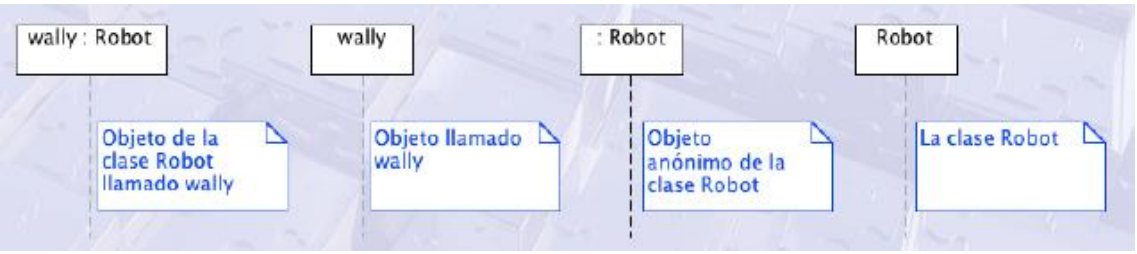
UML: DIAGRAMAS DE INTERACCIÓN

El objetivo de los diagramas de interacción es mostrar el comportamiento del modelo a través de interacciones entre elementos del modelo. Sus principales elementos son:

- *Participantes: objetos y clases que forman parte de la interacción.*
- *Mensajes: flujo y secuencia entre participantes.*

Podemos destacar dos tipos principales de diagramas de interacción (su finalidad es mostrar comportamiento del modelo a través de interacciones entre elementos de este modelo):

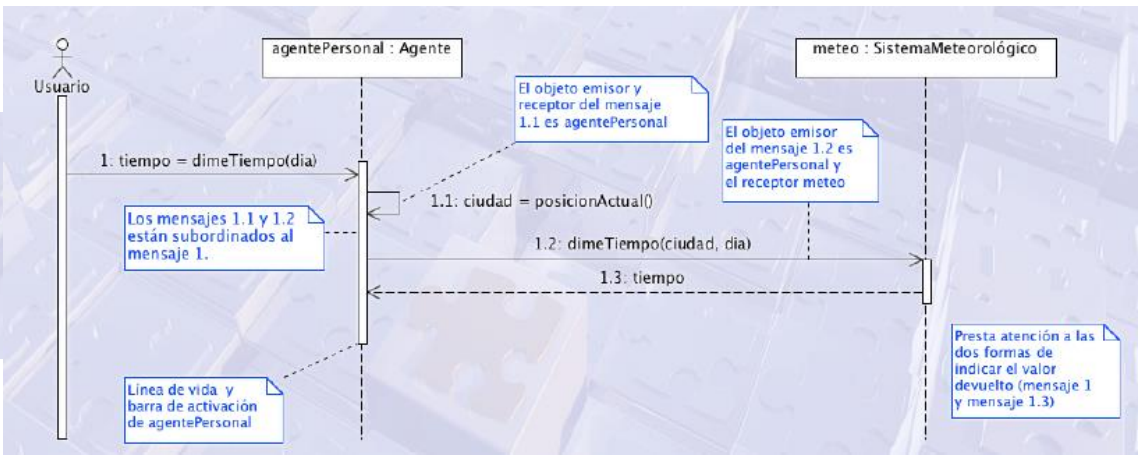
- **DIAGRAMAS DE SECUENCIA**
  Enfatizan la secuencia temporal de mensajes enviados entre objetos.
  Ejemplo (las cajas representan *Participantes*):



  El nombre del objeto en minúscula || El nombre de la clase en mayúscula

  Las colecciones de objetos se representan con doble fondo.

  Ejemplo (emisor y receptor de *Mensajes*):



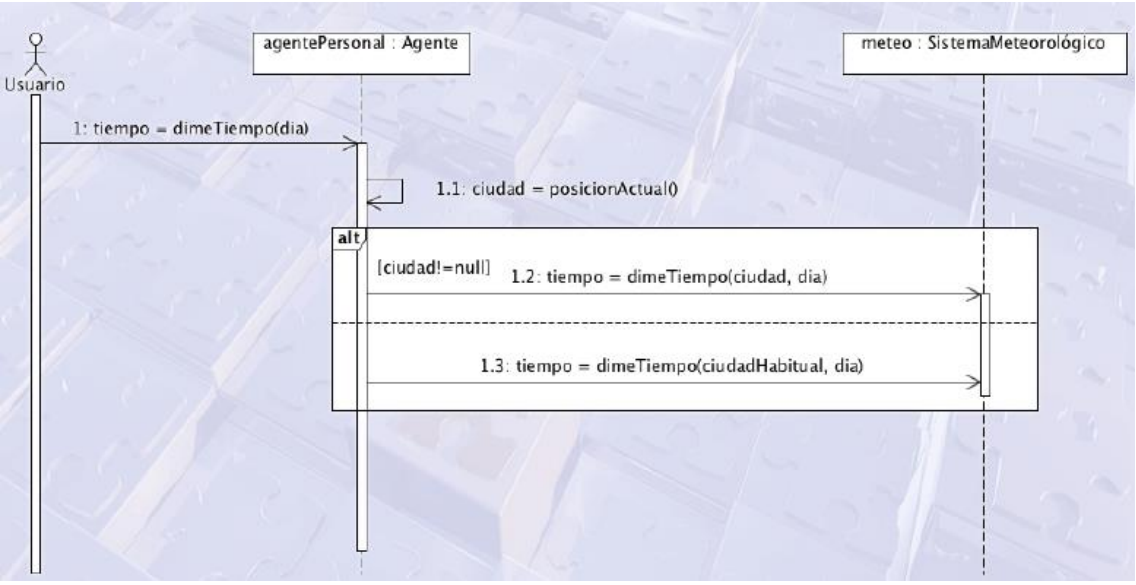
Al principio, el usuario en el **paso 1**, está enviando un mensaje hacia la clase Agente (se sabe por el sentido de la *flecha, las cuales indican envío de mensajes, siendo la punta de la flecha el receptor y el principio de la flecha el emisor*), por lo que entonces al objeto de la clase Agente agentePersonal le estamos pasando el mensaje agentePersonal.dimeTiempo(dia). La línea más ancha de color blanco es el tiempo que está en ejecución.

En el **paso 1.1,** el emisor y receptor del mensaje es el objeto agentePersonal de la clase Agente. El método posicionActual() es un método de instancia de la clase Agente y lo guardo en la variable local ciudad.

En el **paso 1.2**, el objeto emisor del mensaje es agentePersonal de la clase Agente y el objeto receptor del mensaje es meteo de la clase SistemaMeteorlogico. Por lo tanto, el objeto meteo puede llamar al método dimeTiempo(ciudad, dia), que es un método de la clase SistemaMetorlogico. A su vez en el **paso 1.3**, estoy devolviendo el resultado del método dimeTiempo(ciudad, dia) en una variable llamada tiempo, por lo que realmente le estoy devolviendo al objeto agentePersonal el resultado de hacer tiempo = dimeTiempo(ciudad, dia).

Finalmente, para obtener el tiempo (***paso 1***), deben haberse realizado previamente los pasos 1.1, 1.2 y 1.3.

Ejemplo (los fragmentos son *Condicionales*):

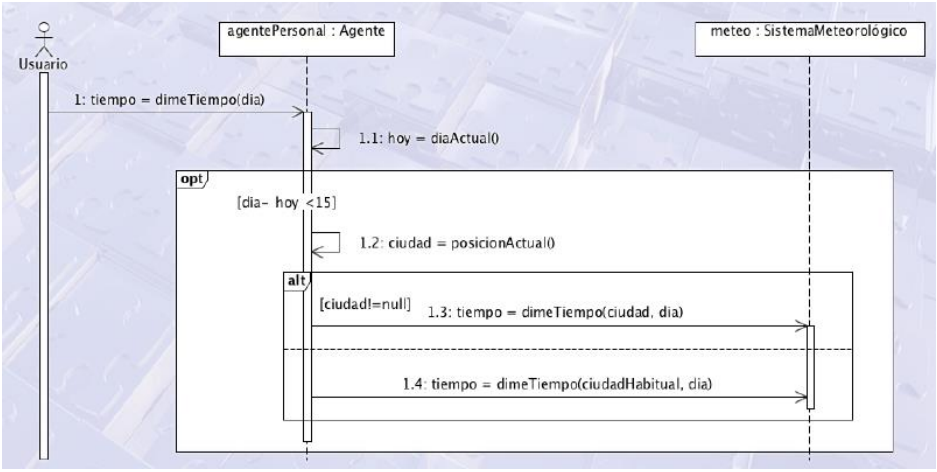


En el cuadro tenemos **alt: alternativas, [ciudad != null]: condición, 1.2: se cumple la condición, 1.3: no se cumple la condición**.

class Agente

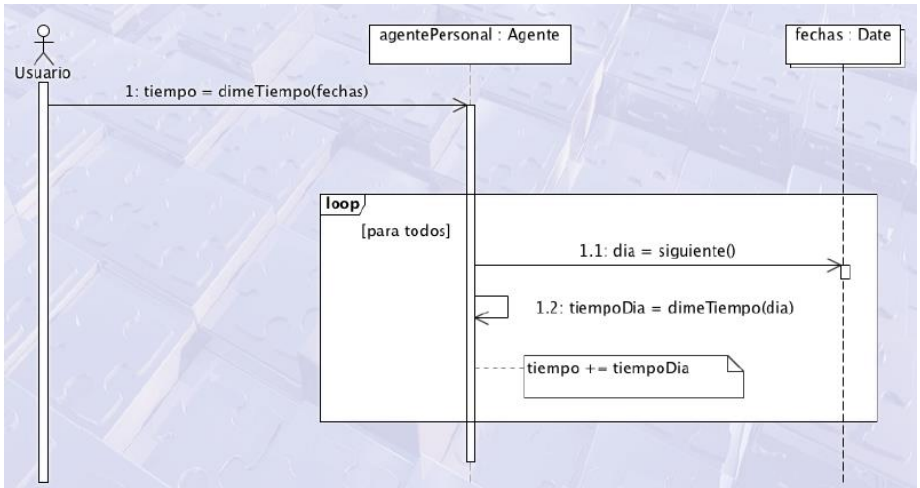
METODO:
def dimeTiempo(dia)

```
        ciudad = posicionActual()

        if (ciutdad != nil) then
                tiempo = @meteo.dimeTiempo(dia, ciudad)

        else
                tiempo = @meteo.dimeTiempo(@ciudadHabitual, dia)
        end
end
```



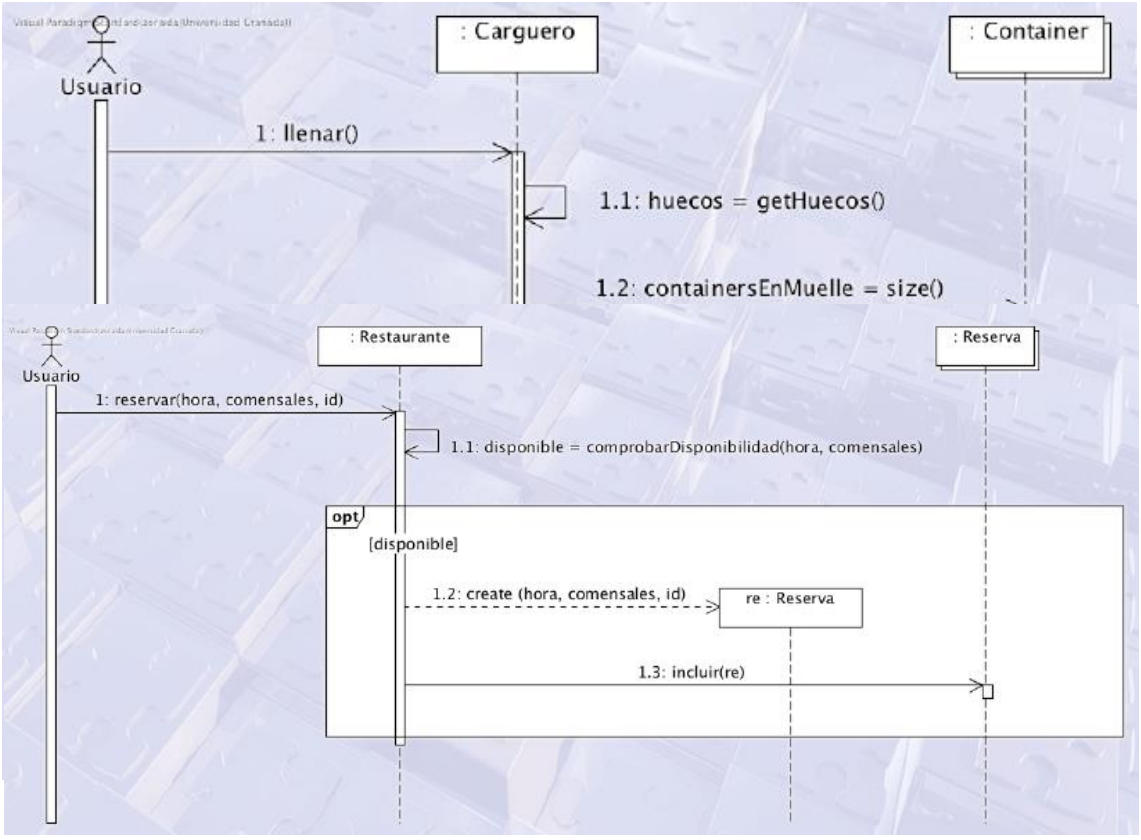
En el cuadro tenemos **opt: opcional y la imagen representa la situación en la que si no se entra en el if, debo tener una nota que indique que hay que hacer**.

Ejemplo (los fragmentos son *Bucles*):



En el cuadro tenemos **loop [para todos] (bucle para todos), fuera del bucle hace falta una nota que inicialize tiempo, en el 1.1 hay que comprender que no hay que implementar un método llamado siguiente, sino que tienes que acceder al siguiente elemento a través de un bucle. La línea de código tiempo += tiempoDia indica que es una nota. Cuando digan siguiente, último, anterior sobre una colección de objetos, hace referencia a un bucle**.

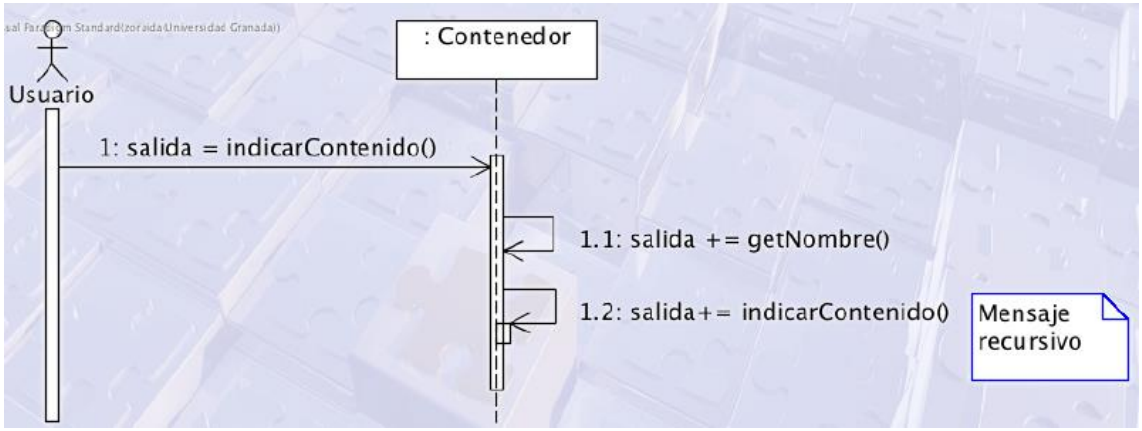
```
class Agente
def dimeTiempo(dia)

        tiempo = " "
        for dia in fechas do
                tiempoDia = dimeTiempo(dia)
                tiempo += tiempoDia
        end

        return tiempo
end
```



Ejemplo (*Creación de instancias*):
En el cuadro de arriba (*re = new Reserva (_hora, _comensales, _id)* de la clase Reserva), por ejemplo, el paso 1.3, no implica implementar un método como tal que se llame incluir, si no que, en este caso, el objeto habría que incluirlo en una colección de reservas.

- Java: .add(re)
- Ruby: << re | .push(re)

Ejemplo (*Recursividad*):



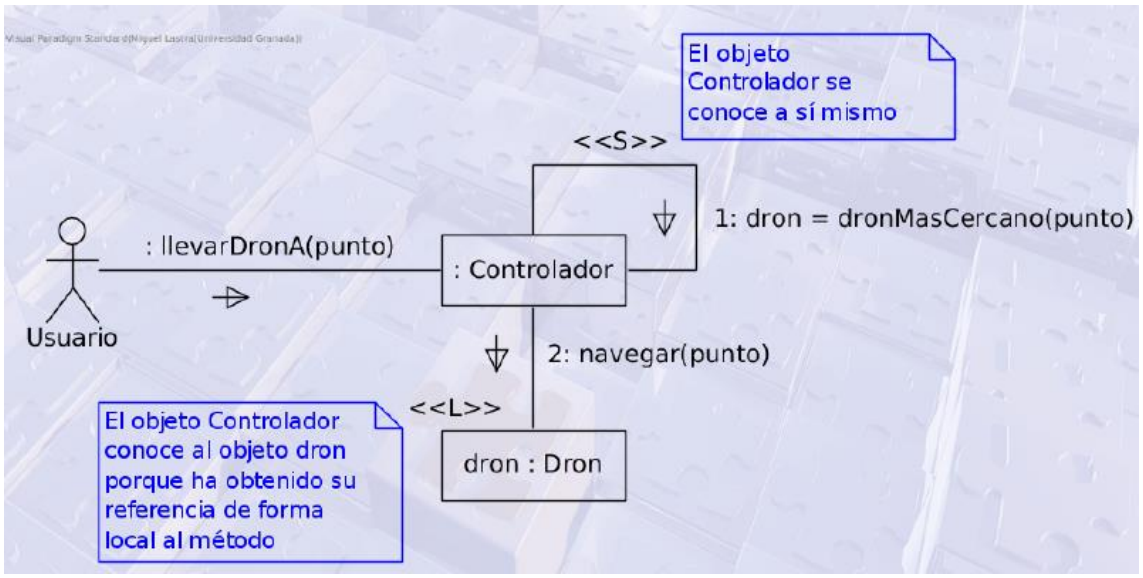
En el cuadro superior, cuando las flechas aparecen así en sentido hacia dentro sobre esa línea ancha con rayas negras verticales hace referencia a la recursividad.

- **DIAGRAMAS DE COMUNICACIÓN**

  Enfatizan la relación o vías de comunicación (enlaces), las cuales son el elemento principal y orden temporal de los mensajes, entre objetos receptores y emisores de los mensajes (participantes). Estas vías de comunicación se representan mediante líneas que unen a los participantes. Los tipos de enlaces son:

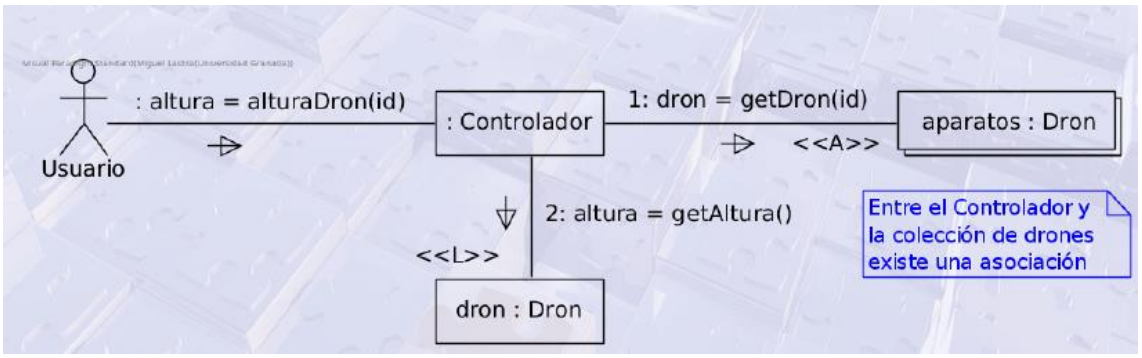
  - Global (G): ámbito superior, por ejemplo, un atributo de clase.
  - Asociación (A): entre los participantes.
  - Parámetro (P): objeto pasado como parámetro.
  - Local (L): participante es un objeto local.
  - Self (S): objeto que se envía mensajes a sí mismo.

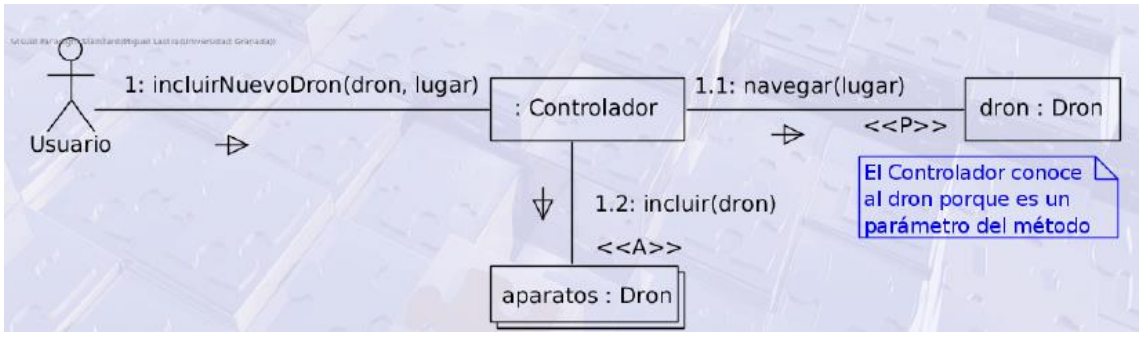
Ejemplos:



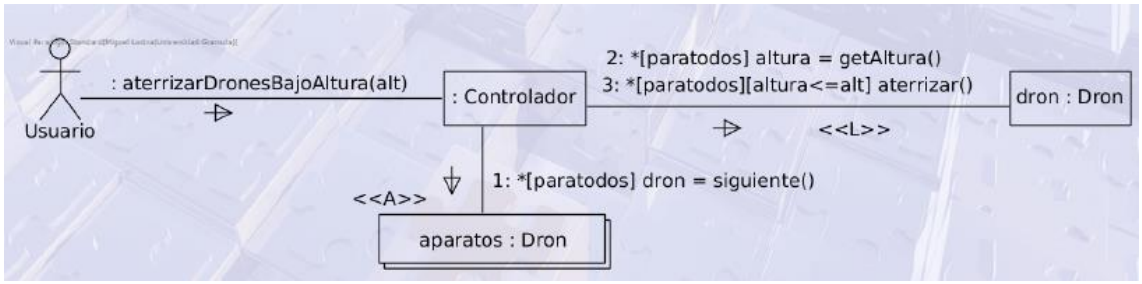
class Controlador

```
        def llevarDronA(punto)
                dron = dronMasCercano(punto)
                dron.navegar(punto)
        end
end
```





Ejemplo: (Condiciones y bucles):



En el cuadro de arriba, los […] indican condiciones, si hay un * indica un bucle y si no hay un * indica que es un condicional. El paratodos, indica mensajes en bucle.

```
class Controlador

        def alBA(alt)

                for dron in aparatos do
                        altura = dron.getAltura()
                        if altura <= alt then
                                dron.aterrizar()
                        end
                end
        end
end
```

TEMA 3 HERENCIA

RELACIÓN DE HERENCIA ENTRE CLASES

La herencia permite derivar clases a partir de clases existentes.

Ejemplo:

class LapizGoma engloba a la class Lapiz

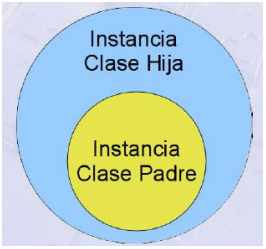
```
class LapizGoma {
        int tamañoGoma;
        void borrar();
        void escribir(): METODO HEREDADO DE CLASE LAPIZ
}
class Lapiz {
        String color;
        int longitud;
        void escribir();
}
```

Con class LapizGoma extends Lapiz, la clase LapizGoma hereda el contenido de la clase Lapiz.

La relación de herencia es una relación **es-un**.

- Clase de la que se deriva: ANCESTRO, SUPERCLASE, CLASE PADRE
- Clase derivada: DESCENDIENTE, SUBLCLASE, CLASE HIJA
- Un descendiente **es-un** ascendente:
  Un lápiz con goma, a todos los efectos, es un lápiz.
- La **relación es-un es transitiva**: Si C hereda de B y B hereda de A, entonces C hereda de A.

La clase hija hereda TODO el código de la clase padre. El acceso depende de la visibilidad.



DIRECTOR → EMPLEADO → PERSONA

@empleado → @persona → @nombre

- Las instancias de Director incluyen referencia a instancias de Empleado, y las instancias de Empleado incluyen referencia a instancias de Persona.
- Cuando se construye un Director hay que construir un Empleado.
- Cuando se construye un Empleado hay que construir una Persona.
- Si a un Director se le pregunta el nombre, se lo pregunta a la instancia de Empleado. Y lo mismo en el caso de instancias de Empleado.
- Toda instancia de Director tiene una instancia de Empleado y Toda instancia de Empleado tiene una instancia de Persona.

Ruby: Implementación (parcial) del DC anterior

```ruby
class Director
  def initialize (n)
    @empleado = Empleado.new(n)
  end
  . . .
end
```

```ruby
class Empleado
  def initialize (n)
    @persona = Persona.new(n)
  end
  . . .
end
```

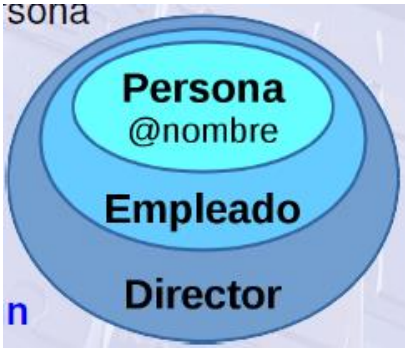
```ruby
class Persona
  def initialize (n)
    @nombre = n
  end
  . . .
end
```

Ruby: Implementación (parcial) del DC anterior

```ruby
class Director
  . . .
  def nombre
    @empleado.nombre
  end
end
```

```ruby
class Empleado
  . . .
  def nombre
    @persona.nombre
  end
end
```

```ruby
class Persona
  . . .
  def nombre
    @nombre
  end
end
```



Ruby: Implementación mediante herencia

```ruby
1 #encoding:utf-8
2
3 # Todas las clases están COMPLETAS
4 #   no se ha omitido ninguna linea
5
6 class Persona
7   attr_reader :nombre
8   def initialize (n)
9     @nombre = n
10  end
11 end
12 class Empleado < Persona
13 end
14 class Director < Empleado
15 end
16
17 el_dire = Director.new("Pedro")
18 puts el_dire.nombre
```

En este caso, con el **símbolo <**:

- Línea 12: la clase Empleado hereda la clase Persona.
- Línea 14: la clase Director hereda la clase Empleado.

La herencia se usa para REUTILIZACIÓN DE CÓDIGO (clase derivada añade / modifica comportamiento de clase padre).

Clase padre → GENERALIZACIÓN DE SUS DESCENDIENTES.

Clase hija → ESPECIALIZACIÓN DE LA CLASE PADRE.

CRITERIOS VÁLIDOS

- Especificación: clases hijas IMPLEMENTAN comportamiento declarado en el padre.
- Especialización: clases hijas MODIFICAN comportamiento de clase padre.
- Extensión: clases hijas AMPLÍAN comportamiento de clase padre.
- Generalización: ascendiente surge de aspectos comunes entre varias clases.

CRITERIOS NO VÁLIDOS

- Construcción: utilizar una clase como base para construir otra sin que haya relación clara entre ellas.
- Limitación: clases descendientes restringen comportamiento especificado por su ascendiente.
- Simplemente reutilización de código: criterio no válido para usar herencia.

Clases Padre, Hija, Nieta

Constructores:

```
Padre (parametros) {
        atributos <-- parametros
}

Hija (param_padre, param_hija) {
        super (param_padre)
        atributos <-- param_hija
}
```

Java: Ejemplo de herencia sencillo

```java
class Persona {
    public String andar() {
        return ("Ando como una persona");
    }

    public String hablar() {
        return ("Hablo como una persona");
    }
}

class Profesor extends Persona {
    public String hablar() {
        return ("Hablo como un profesor");
    }
}

//*******************************

public static void main(String[] args) {
    Profesor profe = new Profesor();
    profe.andar(); // Los profesores también andan
    profe.hablar();
}
```

JAVA:

- Profesor extends Persona: clase Profesor hereda método hablar de la clase Persona.
- Línea 21: profe.andar(): la clase Profesor encuentra el método andar en la clase Persona y como esta clase la hereda, por eso lo puede ejecutar. Si vuelvo a definir un método andar en la clase Profesor, se ejecutaría este último.
- Línea 22: profe.hablar(): se llama al método hablar de la clase Profesor.

Ruby: Ejemplo de herencia sencillo

```ruby
class Persona
  def andar
    "Ando como una persona"
  end
  def hablar
    "Hablo como una persona"
  end
end
class Profesor < Persona
  def hablar
    "Hablo como un profesor"
  end
  def impartir_clase
    "Impartiendo clase"
  end
end
puts Persona.new.andar
puts Persona.new.hablar
puts Persona.new.impartir_clase
puts Profesor.new.andar
puts Profesor.new.hablar
puts Profesor.new.impartir_clase
```

RUBY:

- Línea 17: Persona.new.andar = llamada al método andar de la clase Persona.
- Línea 18: Persona.new.hablar = llamada al método hablar de la clase Persona
- Línea 19: Persona.new.impartar_clase = NO ES POSIBLE, ya que el método impartir_clase es de la clase Profesor y la clase Persona no hereda de la clase Profesor.
- Línea 20: Profesor.new.andar = llamada al método andar de la clase Persona, clase heredada por la clase Profesor.
- Línea 21: Profesor.new.hablar = llamada al método hablar de la clase Profesor.
- Línea 22: Profesor.new.impartir_clase = llamada al método impartir_clase de la clase Profesor.

TIPOS DE HERENCIA

Podemos destacar:

- SIMPLE: clase tiene a lo sumo un ascendiente directo.
- MÚLTIPLE: clase con varios ascendientes directos y heredar de todos ellos.

REDEFINICIÓN DE MÉTODOS

Se redefine o sobrescribe un método cuando una clase proporciona una implementación alternativa a la heredada → IMPLEMENTACIÓN HEREDADA ANULADA.

Se puede reutilizar el código heredado y extenderlo.

PSEUDOVARIABLE SUPER

Cuando se redefine un método, *super* permite ejecutar la implementación proporcionada de la clase padre.

RUBY:

Ruby: Pseudovariable `super`

```
1 class Persona
2   def hablar
3     "Hablo como una persona"
4   end
5 end
6 class Profesor < Persona
7   def hablar
8     tmp = super
9     tmp += ", y también como un profesor"
10    tmp
11  end
12 end
13 puts Profesor.new.hablar
```

- Línea 8: llamada al método hablar de la clase Persona e incorporación de su implementación.
- Línea 13: “Hablo como una persona, y también como un profesor”.

Ruby: `super` en métodos

```
1 def metodo1 (i, j)
2     a = super (i, j)
3     # equivalente en este caso a
4     # a = super
5
6     # error salvo que el objeto devuelto por super
7     # tenga un método llamado metodo2
8     # b = super.metodo2
9
10    return 2*a
11 end
```

- Solo permite acceder en clase padre a implementación del mismo método redefinido.
- Si se utiliza sin argumentos, se pasan los mismos que los recibidos por el método redefinido.

Ruby: Ejemplo de `initialize` sin llamada a `super`

```ruby
class Padre
  def initialize
    @padre = "Padre"
  end
end

class Hija < Padre
  def initialize
    @hija = "Hija"
  end

  def mostrar
    puts "#{@padre} #{@hija}"
  end
end

Hija.new.mostrar     # ¿Qué ocurre aquí?
```

- Es nuestra responsabilidad llamar a super.
- Línea 13: en el puts no va a mostrar a @padre, ya que lo tendríamos que haber puesto nosotros en el initialize de la clase Hija. La manera de ponerlo es llamando a super (para invocar al constructor de la clase Padre).
- Línea 17: llamada al método mostrar de la clase Hija: Padre Hija.

JAVA:

Java: `super` en métodos

```java
int metodo1 (int i, int j) {
    int a = super.metodo1 (i,j);    // Uso adecuado
    int b = super.metodo2();        // Uso NO recomendado
    int c = metodo2();              // Uso recomendado
    return (a+b+c);
}
```

- Permite acceder a implementación de cualquier método de clase padre.
- Usarlo solo para acceder al método de clase padre con el mismo nombre.

Java: `super` en constructor

```java
class Persona {
  private String nombre;
  // Al definir un constructor, deja de existir el constructor por defecto, sin parámetros
  Persona (String nombre) {
    this.nombre = nombre;
  }
}

class Profesor extends Persona {
  private String asignatura;
  Profesor (String nomb, String asign) {
    super (nomb);       // Primera línea. Llamada obligatoria, si no, dará error ¿por qué?
    asignatura = asign;
  }
}
```

- Permite invocar al constructor de clase padre. DEBE APARECER EN PRIMERA LÍNEA DEL CONSTRUCTOR DE CLASE HIJA.
- Si no se invoca expresamente, se invocará a un constructor sin parámetros de la clase padre, y si no existe ese constructor, dará ERROR.
- Línea 12: se llama al constructor por defecto, si no se pone super en la primera línea, y si este constructor no existe, puede provocar ERROR.

PARTICULARIDADES

JAVA:

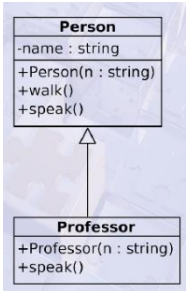
- Todas las clases heredan implícitamente de class Object (es la clase base de la jerarquía de clases. Es la clase principal de la cual todas las demás clases son subclases).
- No pueden ser redefinidos ni métodos constantes (final) ni métodos privados.
- Cuando se redefine un método,
    - Utiliza anotación @Override (encima de cabecera de método).
    - Se permiten cambios en la cabecera: protected a public, valor retornado puede ser subclase.

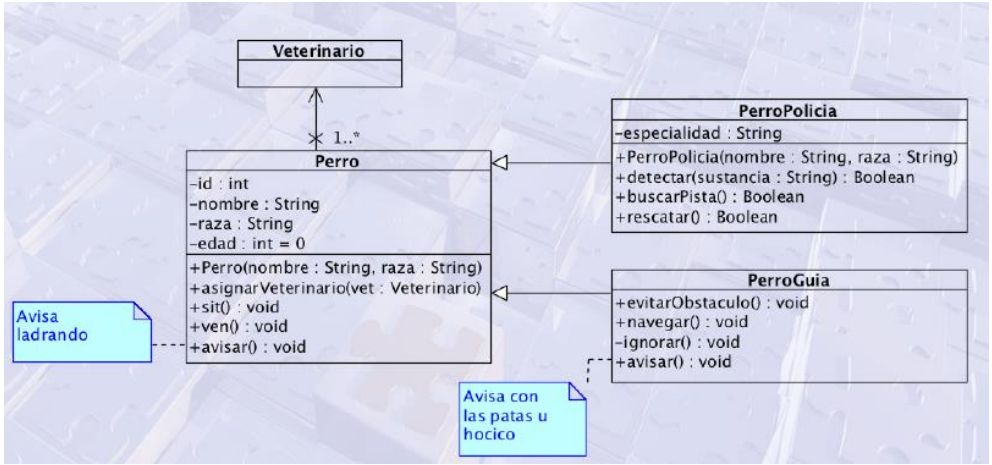
RUBY:

- Todas las clases heredan implícitamente de class Object (es la clase base de la jerarquía de clases. Es la clase principal de la cual todas las demás clases son subclases).
- Cuando se crea un método con el mismo nombre que en la superclase, se produce la redefinición.

DIAGRAMAS DE CLASES CON HERENCIA

La clase Profesor hereda el método speak() de la clase Persona.





private_class_method : nombre_metodo para hacer un método privado.

POLIMOSRFIMO Y LIGADURA DINÁMICA

```java
class Empleado { // entre otras cosas ...
  float calculaSueldo() { return . . . }
}
class Comercial extends Empleado { // entre otras cosas ...
  float calculaSueldo() {
    return super.calculaSueldo() + . . .
  }
}
class Directivo extends Empleado {  // entre otras cosas ...
  float calculaSueldo() {
    return super.calculaSueldo() + . . .
  }
}

// En algún otro sitio ...
ArrayList<Empleado> listaDeEmpleados = new ArrayList<>();
listaDeEmpleados.add (new Empleado (. . .));
listaDeEmpleados.add (new Comercial (. . .));
listaDeEmpleados.add (new Directivo (. . .));

// Se quiere calcular el total de todos los sueldos
float sueldosTotales = 0.0f;
for (Empleado unEmpleado : listaDeEmpleados) {
  sueldosTotales += unEmpleado.calculaSueldo();
}
```

La clase Empleado se puede dividir en Empleado | Comercial | Directivo.

Ligadura dinámica: relación entre la cabecera de un método y su implementación concreta. Ejemplo: método calculaSueldo():

- Ligadura estática: se hace en tiempo de compilación, y limita instancias de la clase del tipo estático o subclases y métodos que pueden ser invocados.
- Ligadura dinámica: tipo dinámico determina que parte del código se ejecutará.
- Casts: se considera temporalmente que un tipo de variable es otro:
    - Downcasting: variable es una subclase del tipo con que se declaró.
    - Upcasting: variable es una superclase del tipo con que se declaró.
    - NO TRANSFORMA OBJETOS | NO CAMBIA TIPOS DIÁMICOS | NO CAMBIA COMPORTAMIENTOS.

Polimorfismo: capacidad de un identificador de referenciar objetos de distintos tipos (clases):

- Tipo estático: tipo (clase) del que se declara la variable.
- Tipo dinámico: es el tipo al que está apuntando en ese momento. En cada iteración el tipo dinámico Empleado será distinto.

```java
class Persona {
    public String andar() {
        return ("Ando como una persona");
    }

    public String hablar() {
        return ("Hablo como una persona");
    }
}

class Profesor extends Persona{
    @Override
    public String hablar() {
        return ("Hablo como un profesor");
    }
}
//*******************************

public static void main(String[] args) {
    Persona p=new Persona();
    Persona p2=new Profesor(); // Puede también referenciar un Profesor

    p.hablar();  // "Hablo como una persona"
    p2.hablar(); // "Hablo como un profesor"
}
```

```java
class Profesor extends Persona{

    public String impartirClase() {
        // Este método no lo tiene Persona ni ninguna de sus superclases
        return ("Impartiendo clase");
    }
}
//*******************************

public static void main(String[] args) {
    Persona p=new Profesor();

    // Error de compilación. Las personas no tienen ese método
    p.impartirClase();

    //Error de compilación. Object no es subclase de Persona
    p=new Object();
}
```

El tipo dinámico de p2 es Profesor por lo tanto hace referencia a un profesor y se puede ver como tal. El tipo dinámico es Profesor pero el tipo estático es Persona por lo que no puedo hacer p.impartir clase().

CLASES PARAMETRIZABLES

Son clases definidas en función de un tipo de dato: *class Lista <T>.*

Este concepto se implementa mediante tipos genéricos: *generics*. Puede haber subclases:

- class Clase <T extends ClaseBase>.
- class Clase <T extends Interfaz>.

Las interfaces también pueden hacerse paramétricas.

CLASES ABSTRACTAS E INTERFACES

CLASES ABSTRACTAS

Representan de manera genérica a otras entidades. Características:

- No se instancian. → ERROR: variable = new ClaseAbstracta.
- Si pueden tener constructores.
- Sí se pueden declarar una variable usando clase abstracta como tipo.

Los métodos sin implementación, que solo llevan cabecera son métodos abstractos.

JAVA:

- Se usa palabra *abstract* para que una clase y métodos son abstractos.
- Permite clases abstractas sin métodos abstractos.

RUBY:

- NO SOPORTA CLASES NI MÉTODOS ABSTRACTOS.
- Se hace privado el método new en clase padre y se hace público el método new en las clases derivadas.

```ruby
class FiguraGeometrica
  . . .   # Atributos y métodos comunes
  private_class_method :new
end

class Cuadrado < FiguraGeometrica
  public_class_method :new
  . . . # Atributos y métodos específicos de esta subclase
end
```

La representación de una Clase Abstracta o Métodos Abstractos pueden ser mediante:

- CURSIVA: ***ClaseAbstracta***
- LETRAS: **<< abstract >>**

INTERFACES

Una interfaz define un contrato que cumplen clases que realizan esta interfaz.

Ejemplo:

Interface Lista { *DEFINO QUÉ PUEDO HACER*

Tan solo declaro los métodos.

}

class ListaVector ***implements*** Lista { *DEFINO CÓMO PUEDO HACER*

atributos privados

Declaro, defino e implemento los métodos de la interfaz Lista

}

Lista l = new ListaVector(); || l.(método de interfaz)

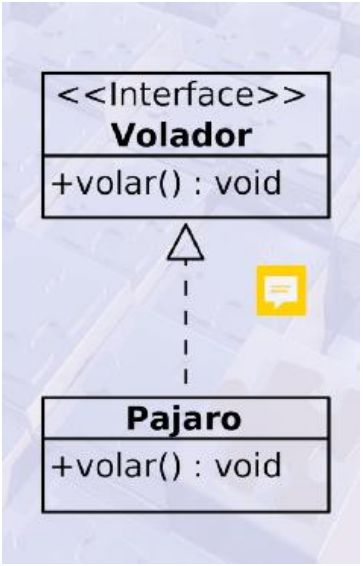
JAVA:

- Clase puede realizar varias interfaces y heredar de una clase.
- Interfaz hereda una o más interfaces.
- Interfaz tiene constantes (PUBLIC, STATIC, FINAL) métodos default, static… (PUBLIC)
- NO PUEDEN SER INSTANCIADAS, SOLO REALIZADAS POR CLASES O EXTENDIDAS POR OTRAS INTERFACES.

RUBY:

- NO PUEDE CONCEPTO INTERFAZ.

REPRESENTACIÓN DIAGRAMA UML DE INTERFACES



- Si tiene línea continua: **HERENCIA**: Clase Pájaro hereda Clase Volador.
- Si tiene línea discontinua: **INTERFAZ**: Clase Pajaro implementa Interfaz Volador.

HERENCIA EN EL ÁMBITO DE CLASE

JAVA

No se permite redefinir métodos de clase al mismo nivel que instancia y no se obtienen los mismos resultados a nivel de clase que a nivel de instancia.

Java: Ejemplo de herencia en el ámbito de clase

```
1 class Padre {
2   public static final int DECLASE = 1;
3   public static int getDECLASE() { return DECLASE; }
4 }
5
6 class Hija extends Padre {
7   public static final int DECLASE = 2; // Variable shadowing
8 }
9
10 class Nieta extends Hija{
11   public static int getDECLASE() { // No es una redefinición
12       // super.getDECLASE() No permitido
13       return DECLASE;
14   }
15 }
16
17 public static void main(String[] args) {
18   System.out.println (Padre.DECLASE);   // 1
19   System.out.println (Hija.DECLASE);    // 2
20   System.out.println (Nieta.DECLASE);   // 2
21   System.out.println (Padre.getDECLASE()); // 1
22   System.out.println (Hija.getDECLASE());  // 1
23   //porque "redefine" el método de clase
24   System.out.println (Nieta.getDECLASE()); // 2
25 }
```

- La "variable shadowing" en Java se refiere a la situación en la que una variable local dentro de un bloque de código tiene el mismo nombre que una variable en un ámbito exterior (como una variable de instancia o una variable de clase). Cuando esto ocurre, la variable local "sombra" o "oculta" la variable de ámbito exterior dentro de ese bloque específico.

- No se puede hacer en la línea 12 super.getDECLASE() porque:

    - El método es estático y super se usa para acceder a miembros de instancia de la clase padre, pero NO para métodos estáticos de la clase padre. Estos métodos se llaman directamente sin usar super. La palabra super se utiliza en contextos NO ESTÁTICOS.

Java: Ejemplo de herencia en el ámbito de clase

```java
public static void main(String[] args) {

  // El tipo estático de las instancias influye

  // Aunque Java lo permite, no se debe invocar a métodos de clase así
  // Lo digo en serio

  Padre p=new Padre();
  System.out.println (p.getDECLASE()); // 1

  p = new Nieta();
  System.out.println (p.getDECLASE()); // 1

  Nieta n = new Nieta();
  System.out.println (n.getDECLASE()); // 2
}
```

- Línea 9: imprime 1 porque devuelve DECLASE de la clase Padre.
- Línea 12: imprime 1 porque a pesar de que estamos creando un objeto de la clase Nieta, el tipo dinámico es Padre, entonces imprime la variable DECLASE de la clase Padre.
- Línea 15: imprime 2 porque aquí si crea un nuevo objeto de la clase Nieta, la cual hereda de Hija, entonces simplemente coje el valor de DECLASE de la clase Hija que corresponde a 2.

RUBY

Ruby: Ejemplo de herencia en el ámbito de clase

```ruby
class Padre
  @atributo_clase1  = 1
  @atributo_clase2  = 2
  @@atributo_clase3 = 5
  def self.salida
    puts @atributo_clase1+1
    puts @atributo_clase2+1 unless @atributo_clase2.nil?
    puts @@atributo_clase3+1
  end
  def self.salida2
    salida
  end
end
Padre.salida # 2 3 6
class Hija < Padre
  @atributo_clase1  = 3
  @@atributo_clase3 = 7
  def self.salida2
    super # Las clases son "first class citizens"
    puts @atributo_clase1+1
  end
end
Padre.salida  # 2 3 8
Hija.salida   # 4 8
Padre.salida2 # 2 3 8
Hija.salida2  # 4 8 4
```

- Importante: los @ encima del initialize solo se puede acceder a ellos a través de MÉTODOS DE CLASE.
- Importante: mirar bien QUIEN LLAMA EN CADA MÉTODO.

- Línea 7: ejecuta la instrucción a no ser que el atributo de la clase 2 sea nulo (nil).

- Línea 23: ejecuta salida de la clase Padre lo que pasa es que al ejecutarse de arriba a abajo y los @@ son comunes a todas las clases el valor que coge del @@atributo_clase3 es 7 y por eso la ejecución será 2,3,8.

- Línea 24: como en Hija no está el método salida, lo busca en el padre y entonces imprime la salida del padre, como no existe el @atributo_clase2, incrementa en 1 el @atributo_clase1 y @@atributo_clase3 de la clase Hija, por eso sale 4 y 8.

- Línea 25: misma explicación que Línea 23, llama a salida2 que llamará a salida.

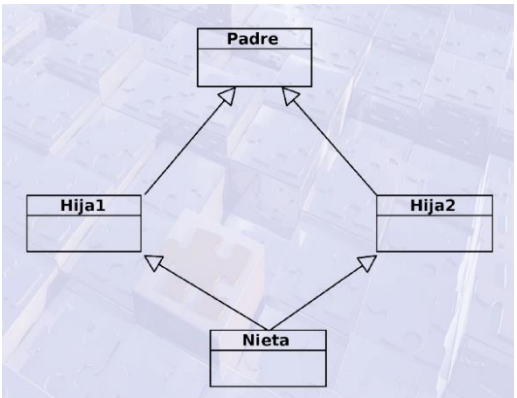
- Línea 26: se llama al método salida2 de la clase Hija, que con el super llama al salida2 del Padre, que a su vez llama a salida del Padre, pero con los atributos de la clase Hija, entonces incrementa en 1 el @atributo_clase1 de la clase Hija (3+1), la variable @atributo_clase2 ni la mira ya que en Hija no existe esta variable, y luego incrementa en 1 el @@atributo_clase3 de la clase hija (7+1). Posteriormente a eso ya ha finalizado todas las llamadas y vuelve al método salida2 de Hija y falta ejecutar la última línea de código que incrementaría en 1 el @atributo_clase1 de la clase Hija (3+1). Por eso el resultado de ejecución es 4, 8, 4.

HERENCIA MÚLTIPLE

Ocurre cuando una clase es descendiente en más de una superclase. **Tanto java como ruby no tienen herencia múltiple**. Representa problemas donde un objeto tiene prioridades según criterios distintos.

Principales problemas:

- Colisión de nombres de métodos / atributos.
- Duplicidad en elementos heredados.
- Relación es-un de una clase descendiente con todos sus ascendentes.



La clase Nieta puede heredar de la clase Padre vía dos caminos distintos.

EJEMPLOS

La solución a la herencia múltiple se puede hacer mediante herencia virtual. La herencia virtual es un concepto en programación orientada a objetos (POO) que se refiere a la capacidad de una clase derivada (subclase) para heredar de múltiples clases base sin causar conflictos de ambigüedad. Cuando una clase base se declara como "virtual" en la herencia, significa que las clases derivadas pueden heredar de ella de manera virtual, lo que permite que una clase derivada herede solo una instancia única de la clase base, incluso si está indirectamente relacionada a través de otras clases. Esto ayuda a evitar problemas de ambigüedad que podrían surgir si una clase derivada hereda de varias clases que comparten miembros con el mismo nombre.

ALTERNATIVAS

Las distintas alternativas son:

- Composición: sustituir una o varias relaciones de herencia por composición.

- Interfaces Java: realizar varias interfaces Java y heredar de una superclase.

    - Línea discontinua finalizada en triángulo vacío: INTERFAZ
    - Línea continua finalizada en triángulo vacío : HERENCIA

- Mixins de Ruby: incluir código de varios módulos como parte de una clase.

    - include Volador: incluye todo el módulo Volador

TEMA 4 MODELO VISTA CONTROLADOR

PATRÓN DE DISEÑO

Un patrón de diseño describe un problema que ocurre varias veces en nuestro entorno y describe el núcleo de una solución al problema para que sea reutilizable.

MODELO VISTA CONTROLADOR

Los elementos son:

- Modelo: clases que representan lógica del problema.
- Vista: representación visual de datos del modelo para mostrar al usuario. La interacción del usuario se produce con elementos de la vista. Para un mismo modelo se pueden tener varias vistas.
- Controlador: actúa de intermediario entre vista y modelo. Controla que los cambios del modelo se vean reflejados en las vistas.

REFLEXIÓN

Es la capacidad de un programa para manipularse a sí mismo. Destacan:

- Introspección: habilidad del programa para observar y razonar sobre su mismo estado (objetos y clases) en tiempo de ejecución.
- Modificación: habilidad del programa para cambiar su estado (objeto y clases) durante la ejecución.

JAVA

Permite introspección.

Java: Ejemplos

```
1 // Ejemplos
2 MiClase obj = new MiClase();
3 Class clase = obj.getClass() //método definido en Object
4 Field[] varInstancia = clase.getFields();
5 Constructor[] construct = clase.getConstructors();
6 Method[] metodosInstancia = clase.getMethods();
7 String nombreClase = clase.getSimpleName();
```

RUBY

Permite introspección y modificación. En ejecución se puede consultar y modificar una clase o consultar y modificar la estructura y funcionalidad de un objeto haciéndolo distinto de lso demás de la misma clase.

Ruby: Modificando la clase. Afecta a todas las instancias

```ruby
class Libro
  def initialize(titulo)
    @titulo = titulo
  end
end

libro1 = Libro.new("El señor de los anillos")

# Se modifica la clase y afecta a todas las instancias
Libro.class_eval do
  def publicacion(añopublicacion)
    @añoPublicacion = añopublicacion
  end
end

libro1.publicacion(1997) # Se invoca el nuevo método
puts libro1.inspect      # Ahora tiene un atributo adicional
```

Ruby: Modificando una única instancia

```ruby
# Se modifica solo una instancia
libro1.instance_eval do
  def autor(autor)
    @autor = autor
  end
end

libro1.autor("J. R. R. Tolkien")
puts libro1.inspect

libro2 = Libro.new("Cien años de soledad")
libro2.autor("G. García Márquez") # ERROR: undefined method `autor'
```

Ruby: Ejemplos de introspección

```ruby
puts  Libro.instance_methods(false) # publicacion
# El parámetro indica si queremos solo los métodos de esa clase (false)
# o también los heredados (true)

puts libro1.instance_variables
                # @autor
                # @titulo
                # @añoPublicacion

puts libro2.instance_variables          # @titulo
puts libro1.instance_of?(Libro)         # true
```

COPIA DE OBJETOS

Podemos destacar:

- Profundidad de copia: nivel en el que se realizan copias de estado.
- Inmutabilidad de los objetos: objeto inmutable: aquel que no tiene métodos que modifiquen su estado.

COPIA DEFENSIVA

Devuelve una copia de estado en vez de una copia de identidad.

- Objetivo: evitar que el objeto de un estado se modifique sin usar métodos de clase.
- Requisito: hacer copias profundas y no superficiales.
- Cuando: consultores usan recurso con objetos mutables que devuelvan.

Podemos destacar clone y la interfaz cloneable:

Crea y devuelve una copia de este objeto. El significado preciso de "copiar" puede depender de la clase del objeto. Por convención, el objeto devuelto por este método debe ser independiente de este objeto (que está siendo clonado).

```
1    x.clone() != x
2    x.clone().getClass() == x.getClass()
3    x.clone().equals(x)
4    //These are not absolute requirements!!!!
```

La interfaz cloneable no define ningún método. En la clase Object se comprueba si la clase que llamó a clone implementa la interfaz, si no, se produce una excepción.

En Ruby el método clone de la clase Object también realiza la copia superficial. La copia profunda la hace el programador. También se puede usar serialización, deserialización para crear una copia profunda: b = Marshal.load (Marshal.dump(a)).

COPIA PROFUNDA

Se hace de manera más segura y garantizando el nivel requerido en cada caso.

También se pueden copiar objetos mediante un constructor de copia, que se encarga de hacer la copia profunda. Tiene problemas cuando se utilizan jerarquías de herencia.

IMPORTANTE

- *attr_accessor (consultor + escritor)* es un método en Ruby que se utiliza para definir atributos en una clase.
- *attr_reader (consultor) :atributo_lectura* en Ruby # Crea un método de lectura público.
- *attr_writer (escritor) :atributo_escritura* en Ruby # Crea un método de escritura público.
- *Inspect* es un método en Ruby para representar el objeto de tu manera personalizada (es como tu to_s personalizado).
- cout en Ruby es con puts
- cout en Java es con System.out.println()
- Cuando en Ruby necesitas un fichero se usa *require 'nombre_fichero'.* En Java, si estas en el mismo paquete, no necesitas indicar nada.
- En Ruby los atributos son privados por defecto. En son de paquete por defecto.
- En Ruby las variables de clase @@ no se inicializan en el constructor.
- En Ruby se puede copiar todo el contenido de un módulo dentro de una clase con *include mimodulo*.
- Clase Enum en Java: *Orientation.DOWN public enum … {PLAYER}*
- Clase Enum en Ruby: *Orientation::DOWN module … PLAYER=: player*
- *JAVA:* variable constante se indica como ''final''.
- *JAVA:* this → solo puede ser usado con métodos de instancias de clase
- *RUBY:* self → solo puede ser usado con métodos de instancias de clase
- Cuando tienes una variable @ encima del initialize, SOLO PUEDES ACCEDER A ELLA A TRAVÉS DE METODOS DE CLASE.
- Ruby: upto(9): crear un array de 9 instancias.
- Ruby: #{@variable} para pasarla a String
- Una variable de instancia SÍ PUEDE llamar a un método de clase en JAVA.
- Una variable de instancia NO PUEDE llamar a un método de clase en RUBY.
- Contains: contiene en Java para ArrayList.
- Include?: contiene en Ruby para Array.
- Asociación: línea normal || Dependencia: líneas discontinuas || Clase Asociación: la que aparece en líneas discontinuas así como una T || Composición: rombo negro || Agregación: rombo blanco || Restricción: entre paréntesis.
- Método equals en Java: comprueba si son iguales.
- En esquinas mezclar todo, en el centro se mantiene.
- En ArrayList, lo pones en un sitio o en otro, no en los dos a la vez.
- Declarar float en JAVA: float a = 5.0f;
- En ruby pasar a float con función to_f.
- Importar numeros aleatorios en Java: import util.java.Random
- Importar ArrayList en Java: import java.util.ArrayList
- Generar nuevo Random en Java: new Random();

- Generar numero aleatorio entero entre 0 y max-1 en Java: nextInt(max).
- Generar numero aleatorio flotante entre 0 y 1 en Java: nextFloat.
- Generar nuevo Random en Ruby: r = Random.new
- Generar numero aleatorio entre min y max = r.rand(min. Max)
- JAVA: para pasar int a string: String.valueOf(float) y Integer.toString(int)
- RUBY: declarar un array: a = Array.new(r) {Array.new(c, ele)}.

Ruby: Consultores y modificadores implícitos

```ruby
class UnaClase

  attr_reader   :atr1
  attr_accessor :atr2
  attr_writer   :atr3

  def initialize (un, dos, tres)
    @atr1 = un
    @atr2 = dos
    @atr3 = tres
  end

end

obj = UnaClase.new(1,2,3)
obj.atr2 = 8
puts obj.inspect
obj.atr2 = 9
puts obj.inspect
obj.atr3 = 7
puts obj.inspect
puts obj.atr1
puts obj.atr2
#puts obj.atr3 # no existe consultor
#obj.atr1 = 23 # no existe modificador
```

Ejemplo: Ruby

```ruby
module Externo
  class A
  end

  module Interno
    class B
    end
  end
end

module Test
  def test
    puts "Testeando"
  end
end

class C
  include Test   #  Literalmente , se copia el contenido del módulo  Test
end

a = Externo::A.new
b = Externo::Interno::B.new
c = C.new
c.test
```

Creación de objetos con Java:

Lapiz milapiz = new Lapiz (Color.Rojo)

Lapiz tulapiz = new Lapiz (Color.Verde)

Creación de objetos con Ruby:

milapiz = Lapiz.new(atributos)

tulapiz = Lapiz.new(atributos)

Paquetes en Java:

package Basico con class Persona y class ColorPelo (enum). Si quiero importarlas estando en otro paquete (package Otro) se importa así: *import Basico.Persona e import Basico.ColorPelo* y así puedo usar la parte publica de Persona y ColorPelo.

Módulos en Ruby

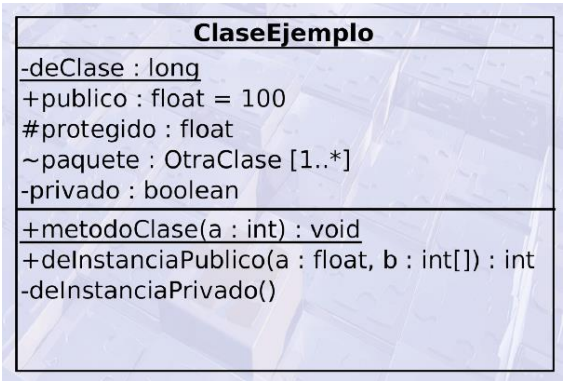
Tenemos un módulo Basico y estamos fuera de el, y queremos crear el objeto manolo, pues sería *manolo = Basico::Persona.new("Manolo", 20, Basico::ColorPelo::MORENO)*

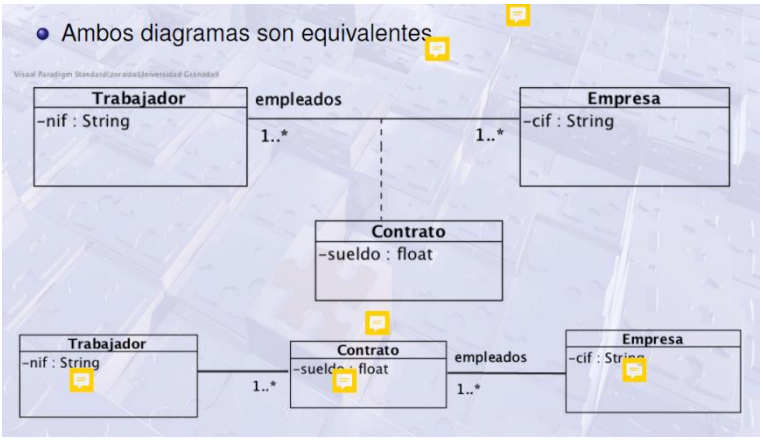
ArrayList de Java

- ***import java.util.ArrayList;***
- añadir un elemento: ***add(ele)*** o ***insert(pos, ele)***
- tamaño del ArrayList: ***size()***
- obtener un elemento: ***get(posición)***
- eliminar un elemento: ***remove(posición)*** o ***remove(ele)***
- posición del elemento: ***indexOf(elemento)***
- ¿hay un elemento?: ***contains(elemento)***
- Iterar: ***for (Tipo ele: array) {***
  operaciones con ele
  }

Arrays de Ruby (I)

- creación: ***a =*** **[ ]**
- añadir un elemento: ***push(ele)*** o ***<< ele*** o ***insert(pos, ele)***
- obtener tamaño: ***length()*** o ***size()***
- obtener un elemento: ***[posición]***
- eliminar un elemento: ***delete_at(posición)*** o ***delete(ele)***
- posición del elemento: ***index(elemento)***
- ¿hay un elemento?: ***include?(elemento)***
- Iterar: ***for ele in array***
  operaciones con ele
  end





- : private | +: público | #: protegido | ~: paquete | subrayado: estático.

En diagramas UML tenemos:

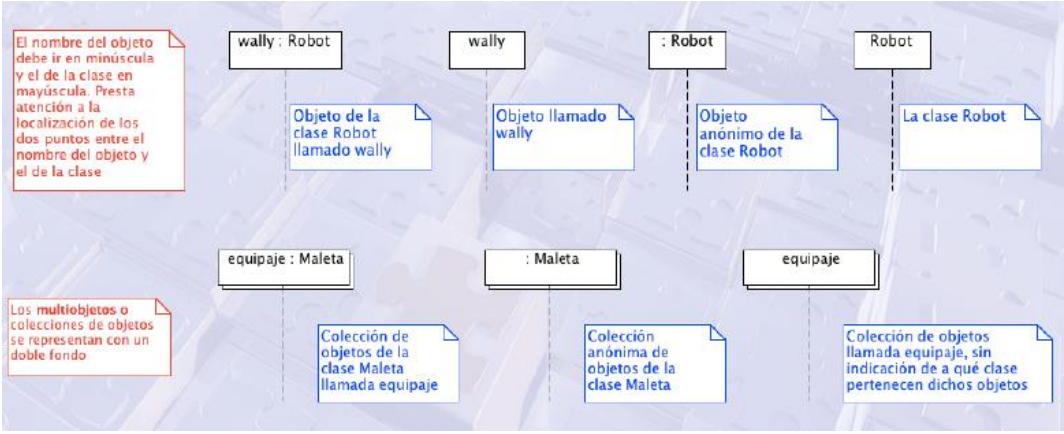
- Navegabilidad: se representan con las flechas. Si no se indica nada, las relaciones son bidireccionales.
- Cardinalidad / multiplicidad: se representan con números. Si no se indica nada, su valor por defecto es 1.
- Propiedades de los extremos: unique (elementos no se repiten), ordered (se trata de una secuencia ordenada).

Para crear un objeto, y añadirlo sería:

- Java: .add(re)
- Ruby: << re | .push(re)

En los diagramas de interacción tenemos:

- Participantes: Objetos y clases que forman parte de la interacción.
- Mensajes: El flujo y su secuencia entre los participantes.



JAVA:

- **Por defecto (si no se indica nada):** visibilidad de paquete (puedo acceder a esta variable desde cualquier clase DENTRO del mismo paquete, pero no desde otro paquete).

- **publico**: puedes acceder a esa variable desde cualquier clase, siempre que el atributo y la clase sean públicas y si la quieres usar desde otro paquete tienes que importar la clase correspondiente (import paquete.clase)

- **private**: solo puedes acceder dentro de la misma clase en el mismo paquete a elementos privados ya sea desde ámbito de instancia como ámbito de clase.

- **protected**: son elementos públicos en la misma clase y dentro del mismo paquete (accesibles desde el mismo paquete) y también desde subclases de otros paquetes (clases que tengan alguna relación con el paquete original). Si las clases involucradas están en mismo paquete los elementos protected son accesibles siempre. Si una clse está fuera del paquete o no es subclase, no puede acceder a los métodos protected.

RUBY

- **Por defecto (si no se indica nada)**: los atributos son siempre privados, mientras que los métodos son públicos.

- **publico**: puedo acceder desde otra clase dentro del mismo paquete o desde otra clase en un paquete distinto, siempre importando lo necesario (require_relative …)

- **private**: tan solo puedo acceder en la misma clase del mismo paquete y en el mismo ámbito. A un método privado de instancia solo se puede acceder desde otro método de instancia. Y si el método privado es de clase, solo puedo acceder desde otros métodos de clase. Método initialize siempre privado.

- **protected**: la clase del código que invoca debe ser la misma o una subclase donde se definió el método. No existen métodos protegidos de clase.