

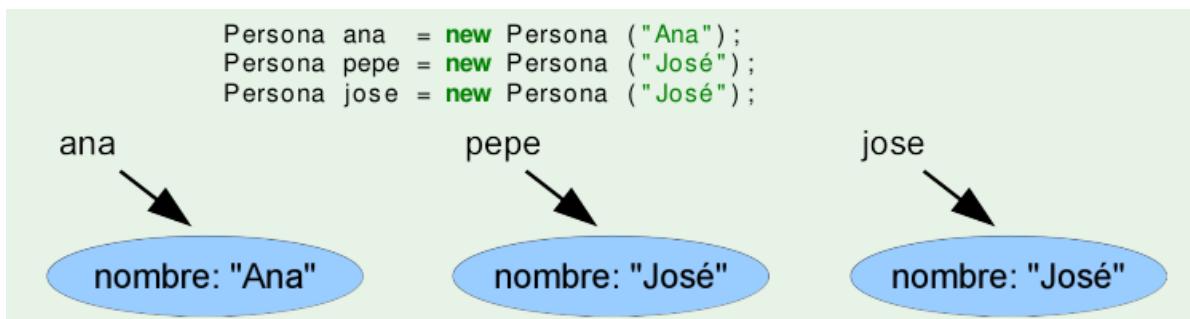
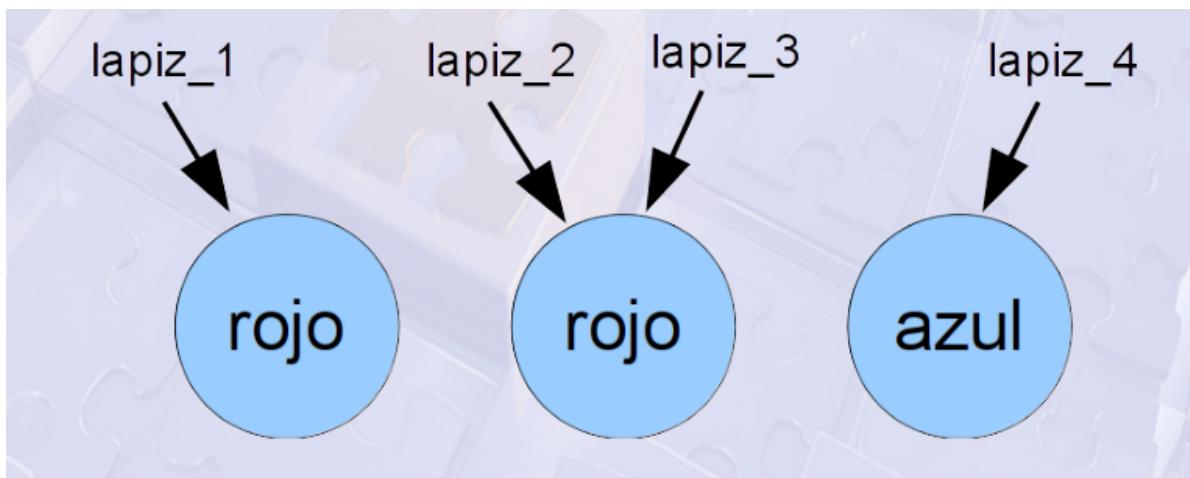


PDOO

- Visual Paradigm** para crear diagramas
- ¡IMPORTANTE!** En **Java** y **Ruby** **solo** hay **punteros**, por lo que **todo** son **referencias**
- Cada **instancia** de una **clase** (**objeto**) tiene su **propia** **identidad** que define su propia **zona de memoria**

▼ Ejemplo

```
// La asignación no declara objetos distintos, sino punteros  
// objeto  
lapiz2 = lapiz3  
// Representa lo siguiente (TODO SON PUNTEROS)
```



- Los **métodos de instancias** son aquellos métodos que son **usados** por **cada objeto**
- Hay **atributos de clase** que sirven para que se pueda **acceder a dicha información** desde la **propia clase**, por lo que **no son parte de cada objeto**
- En el caso de los **métodos de clase** trabajan con los **atributos de clase** y **NO** pueden acceder a los **atributos normales**

▼ Ejemplo

Java: Contador de instancias

```

1 class Persona {
2     // atributo y método de clase
3     static private int numPersonas = 0;
4     static int getNumPersonas () {
5         return numPersonas;
6     }
7     // atributo de instancia
8     private String nombre;
9     // inicializador
10    Persona (String unNombre) {
11        nombre = unNombre;
12        numPersonas++;
13    }
14 }
```

Ruby: Contador de instancias

```

1 class Persona
2     # atributo y método de clase
3     @@num_personas = 0
4     def self.num_personas
5         @@num_personas
6     end
7     # inicializador
8     def initialize (un_nombre)
9         # atributo de instancia
10        @nombre = un_nombre
11        @@num_personas += 1
12    end
13 end
14 }
```

- ☐ Cuando se **devuelve** el **valor de una variable**, se está **devolviendo** una **referencia** a un **objeto**

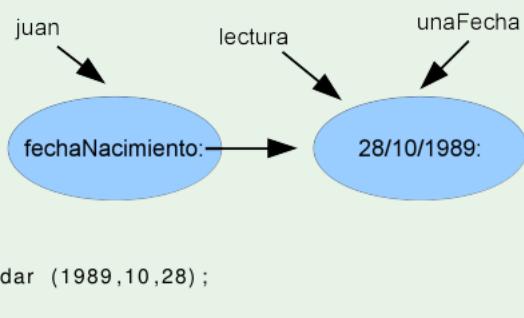
▼ Ejemplo ERROR FOR THAT

Java: Asignación y devolución de referencias

```

1 class Persona {
2     private GregorianCalendar fechaNacimiento;
3
4     Persona (GregorianCalendar nace) {
5         fechaNacimiento = nace;
6     }
7
8     GregorianCalendar getFechaNacimiento () {
9         return fechaNacimiento;
10    }
11
12    // . .
13 }
14
15 GregorianCalendar unaFecha = new GregorianCalendar (1989,10,28);
16
17 Persona juan = new Persona (unaFecha);
18 System.out.println(juan.toString()); // Nací el 28/10/1989
19
20 GregorianCalendar lectura = juan.getFechaNacimiento ();
21 lectura.set (1985,5,13);
22 System.out.println(juan.toString()); // Nací el 13/5/1985
23 unaFecha.set (2001,1,1);
24 System.out.println(juan.toString()); // Nací el 1/1/2001

```



▼ Clase Random

RUBY

- ☐ Existe la **clase Random** y su método **mas importante <var>.rand** funciona así

```

<nombre> = Random.new
# Numeros aleatorios de 0 a <num>-1
<nombre>.rand(<num>)
# Numero aleatorios de 0 a <num>
<nombre>.rand(0..<num>)
# Numero aleatorio de 0.0 a <num> sin incluir <num>
<nombre>.rand(0.0...<num>)

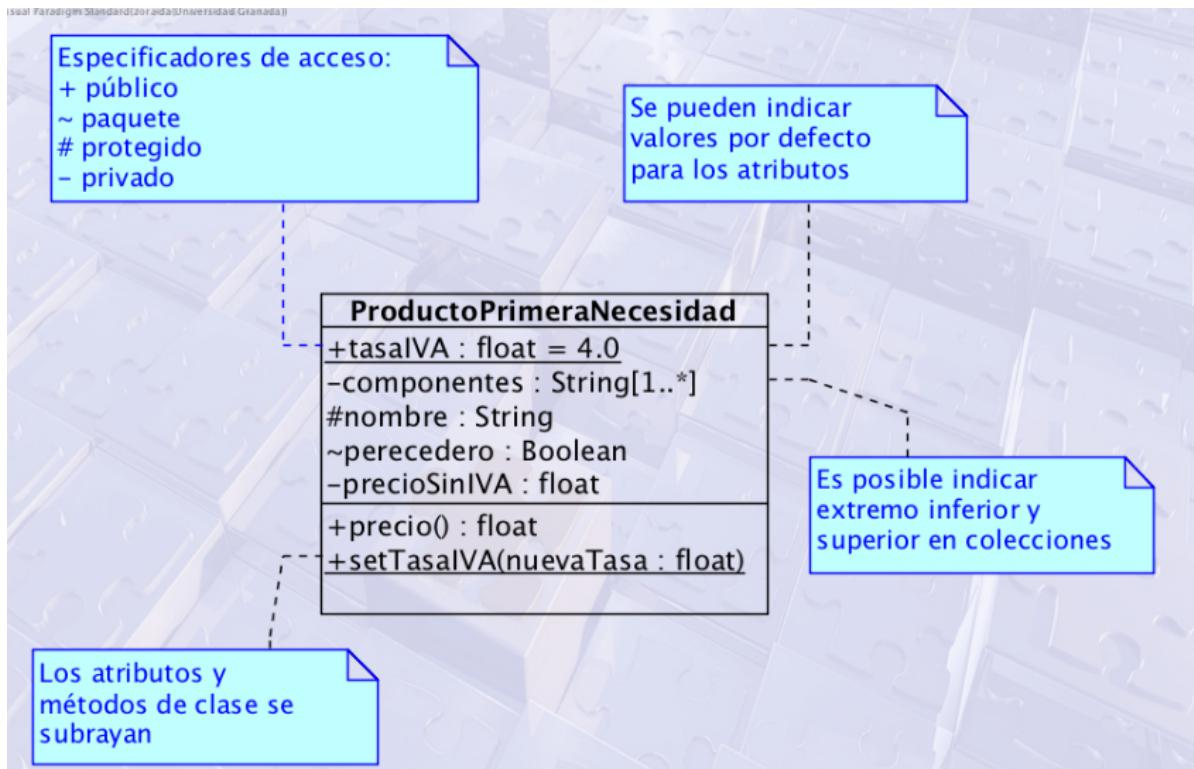
```

- ☐ **Por defecto** saca **numeros** entre **0** y **1** con **decimales**

- Destacar que si queremos `sacar decimales` poner la `acotación` de **numeros float**

Diagramas estructurales

- **UML**: es un “**Lenguaje Unificado de Modelo**”, independiente del lenguaje de programación
 - Veamos **símbolo** que `indican` la **visibilidad**:
 - `#`: **protégido**
 - `+`: **público**
 - `-`: **privado**
 - `~`: **de paquete**
 - Para **definir** una `clase` se divide en los **atributos** (sean de `instancia` o `clase`) y los **métodos**
 - Los `métodos` y `atributos de clase` se **subrayan**
 - Los `atributos` pueden tener **valores por defecto** (se indica)
- ▼ **Ejemplo**



Asociación

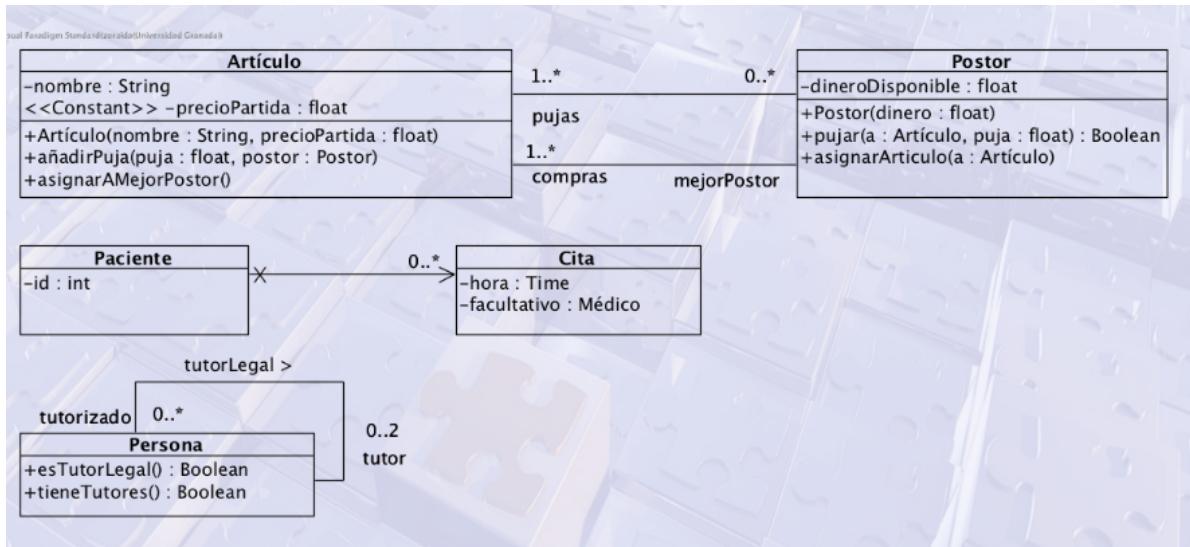
Cardinalidad:

- se `representa` con **números**, * indica **muchos**
- indica **nº instancias** de la `clase de un extremo` están **vinculadas** a la `clase del otro extremo` (se puede poner el **nombre** para `guardar`)
- **por defecto** `nº instancias` es **1**

Navegabilidad:

- se `representa` con **flecha**
- indica si es `posible` **conocer** la/s `instancia/s relacionadas` con la **instancia de origen**
- si en un `extremo` hay `alguna flecha`, en el `otro extremo` **nada=X**
- **por defecto flecha bidireccional**

▼ Ejemplo



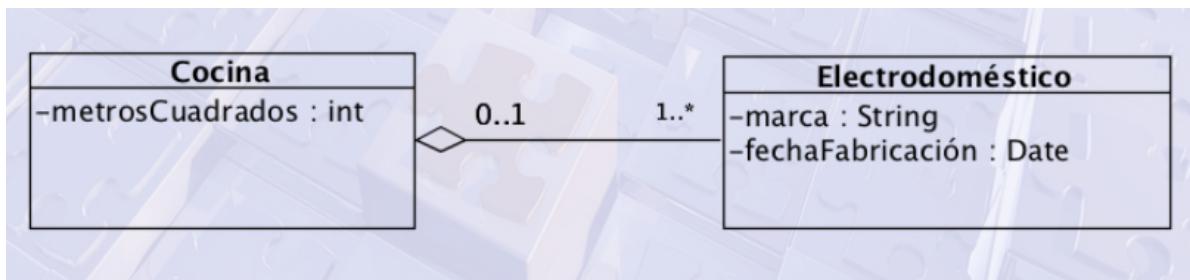
Tipos de relación

Agregación



- la **clase** con **rombo** es el **TODO** y la **otra** es las **PARTES**
- **cualquier cardinalidad** en el **TODO** (una **PARTE** puede estar en **varios** **TODO** o en **ninguno**)

▼ Ejemplo



Composición FUERTE



- las **PARTES NO** tienen sentido sin el **TODO**
- cardinalidad **1** en el **TODO**(una **PARTE siempre** esta en un **solo TODO**)

▼ Ejemplo



Dependencia DEBIL



- cuando en una clase se utilizan instancias de otra
- la punta indica quien es utilizada, la otra parte quien utiliza
- **NO** genera atributos

Herencia

- La clase desde la que se hereda se llama **clase padre** o **superclase** (**generalización**) y la derivada **clase hija** o **subclase** (**especialización**)
- La relación que se construye es la de **es-un** y es una **relación transitiva**
- La clase hija hereda TODO el código del parent
- Si se llama a un método a la clase hija, lo busca en la definición de la clase hija, sino está lo busca en la clase parent y así sucesivamente

▼ Ejemplo

Ruby: Ejemplo de herencia

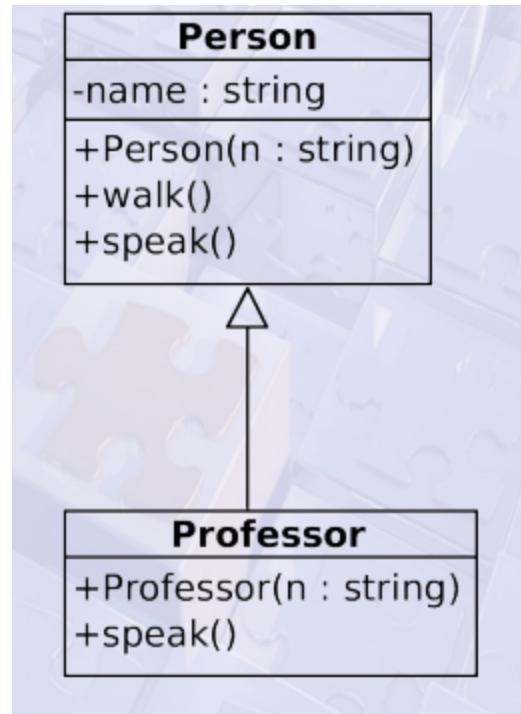
```
1 class Persona
2   def andar
3     "Ando como una persona"
4   end
5   def hablar
6     "Hablo como una persona"
7   end
8 end
9 class Profesor < Persona
10  def hablar
11    "Hablo como un profesor"
12  end
13  def impartir_clase
14    "Impartiendo clase"
15  end
16 end
17 puts Persona.new.andar
18 puts Persona.new.hablar
19 puts Persona.new.impartir_clase
20 puts Profesor.new.andar
21 puts Profesor.new.hablar
22 puts Profesor.new.impartir_clase
```

Java: Ejemplo de herencia sencillo

```
1 class Persona {  
2     public String andar() {  
3         return ("Ando como una persona");  
4     }  
5  
6     public String hablar() {  
7         return ("Hablo como una persona");  
8     }  
9 }  
10  
11 class Profesor extends Persona {  
12     public String hablar() {  
13         return ("Hablo como un profesor");  
14     }  
15 }  
16  
17 //*****  
18  
19 public static void main(String [] args) {  
20     Profesor profe = new Profesor();  
21     profe.andar(); // Los profesores también andan  
22     profe.hablar();  
23 }
```

- En los **diagramas** de la **clase hija** **sale** una **flecha rellena** hacia la **clase padre**, y en la **clase hija** se notan los **atributos** y **métodos nuevos** y **redefinidos** (o **sobrecargados**)

▼ Ejemplo



JAVA

- Para **compilar** y **ejecutar** código así

```
javac <archivos> <archivo_main>
java Main
```

- ¡IMPORTANTE!** Al principio de cada archivo poner el **paquete** al que pertenece
package <paquete>

- Para **escribir en consola** es así

```
System.out.println( "<texto>" + <variables> );
```

▼ Ejemplo

Java: Ejemplo básico

```
1 package basico;
2
3 //Enumerado con visibilidad de paquete
4 /* public */ enum ColorPelo { MORENO, CASTAÑO, RUBIO, PELIROJO }
5
6 public class Persona { // Clase con visibilidad pública
7     private String nombre; // Atributos de instancia privados
8     private int edad;
9     private ColorPelo pelo;
10
11    public Persona (String n,int e,ColorPelo p) { // Constructor público
12        nombre=n;
13        edad=e;
14        pelo=p;
15    }
16
17    void saluda() { // Visibilidad de paquete. Método de instancia
18        System.out.println("Hola, soy "+nombre);
19    }
20 }
21
22 public class Basico { // Clase con programa principal
23     public static void main(String [] args) {
24         Persona p=new Persona("Pepe",10,ColorPelo.RUBIO);
25         p.saluda();
26     }
27 }
```

- Los tipos de **datos simples** **NO** son **objetos**, es decir, son **tipos primitivos** (int, float, char, etc)
- Cada vez que se **cambia un "string"** se crea un **objeto nuevo** al que **apunta** su **respectivo puntero**

▼ Ejemplo

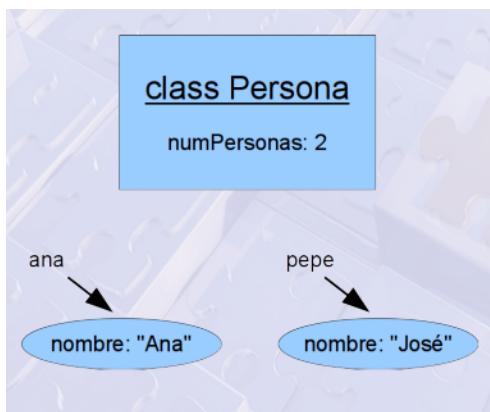
```
// Destacar poner String con S mayuscula porque es una clase
String s1="Hola"
String s2="Hola"
// Ambos apuntan al mismo objeto "Hola"
s1=s1 + "y adios"
// Se crea un nuevo objeto al que apunta s1
```

Ejemplo: La clase Persona

```
1 class Persona {  
2     private String nombre;  
3     // ...  
4     String saludar () {  
5         return "Hola, me llamo " + nombre;  
6     }  
7     void cambiaNombre (String otroNombre)  
    {  
8         nombre = otroNombre;  
9     }  
10}
```

Ejemplo: La clase Persona

```
1 Persona pepe = new Persona ("José")  
2 Persona ana = new Persona ("Ana");  
3  
4 System.out.println (ana.saludar ());  
5 // Hola, me llamo Ana  
6  
7 ana.cambiaNombre ("Ana Belén");  
8  
9 System.out.println (ana.saludar ());  
10 // Hola, me llamo Ana Belén  
11  
12 System.out.println (pepe.saludar ());  
13 // Hola, me llamo José
```



Ejemplo: Contador de instancias

- Tanto `ana`, como `pepe` tienen accesible el atributo de clase `numPersonas`
- Cuando su valor cambia, lo hace para todas las instancias
- El atributo es único, está en una zona de memoria asociada a la clase, no a las instancias

		Mismo paquete		Otro paquete	
		Subclase	Otra	Subclase	Otra
-	private	<i>no</i>	<i>no</i>	<i>no</i>	<i>no</i>
#	protected	<i>sí</i>	<i>sí</i>	<i>sí</i>	<i>no</i>
+	public	<i>sí</i>	<i>sí</i>	<i>sí</i>	<i>sí</i>
~	package	<i>sí</i>	<i>sí</i>	<i>no</i>	<i>no</i>

Enumerados

- Se crean con la siguiente sintaxis

```
<visibilidad> enum <nombre>{  
    <nombre1>,  
    <nombre2>,
```

```
    ...
}
```

Atributos y métodos

- Así se ponen los **métodos de clase** y los **atributos de clase**

```
class <nombre>{
    // atributos de clase
    // "final" indica que no se puede modificar dicho valor (cor
    static private final <tipo> <NOMBRE> = <valor>

    // método de clase
    static <tipo> <nombre>{
        <calculos>
    }
    ...
}

// Para acceder
<clase>.<método_clase>
```

- Se puede **acceder** a **atributos y metodos privados** de un **objeto** siempre **dentro de su propia clase**, aunque sea por **llamada de otro** (como **C++**)

▼ Ejemplo

Java: Visibilidad

```
1 public class UnaClase {  
2  
3     public void metodoPublico() { System.out.println("Público"); }  
4  
5     private void metodoPrivado() { System.out.println("Privado"); }  
6  
7     // Todo esto funciona en Java aunque llame la atención  
8     public void usoDentroDeClase() {  
9         metodoPrivado();  
10    }  
11  
12    public void usoConOtroObjeto() {  
13        UnaClase obj2 = new UnaClase();  
14        obj2.metodoPrivado();  
15    }  
16  
17    public static void main(String []args) { // Seguimos en UnaClase  
18        UnaClase obj = new UnaClase();  
19        obj.metodoPublico();  
20        obj.metodoPrivado();  
21        obj.usoDentroDeClase();  
22        obj.usoConOtroObjeto();  
23    }  
24 }
```

- En los **métodos de clase NO** se puede **acceder** a los **atributos de instancia**, obviamente se puede **acceder** a todos los **metodos y atributos** de esa **misma clase** si se pasa como **parámetro**
- Desde una **instancia** puedo **llamar** a **métodos de clase** (en **RUBY NO**)
- Puedo **llamar** a **métodos de clase (público y privado)** desde **métodos de instancia** así **<nombre_clase>.<metodo>**

Constructores

- Existe un **constructor por defecto**
- Así se usa el **constructor de la clase (new)**

```
// Así se llama para usar el constructor  
new <clase> (<variables>);  
  
// Se crea igual que C++
```

```
class <nombre>{

    // Constructor
    <nombre> (<variables>){

        ...
    }
    ...
}
```

- DESTACAR nuevo uso de **this** como **2ºconstructor** así **this (<parámetros>)** dentro de clase DEBE IR EN LA PRIMERA LINEA

▼ Ejemplo

```
class Persona {
    private String nombre;

    public Persona(String nombre) {
        this.nombre = nombre;
    }

    public Persona() {
        this("Anónimo");
    }
}
```

```

1 class RestrictedPoint3D {
2     private static int LIMITMAX = 100; // Atributos de clase
3     private static int LIMITMIN = 0;
4     private int x; // Atributos de instancia
5     private int y;
6     private int z;
7
8     private int restricToRange (int a) { // Método de instancia
9         int result = Math.max (LIMITMIN, a);
10        result = Math.min (result, LIMITMAX);
11        return result;
12    }
13
14    RestrictedPoint3D (int x, int y, int z) { // Constructor
15        this.x = restricToRange (x);
16        this.y = restricToRange (y);
17        this.z = restricToRange (z);
18        // Debido a la igualdad de nombres,
19        // es necesario usar "this" para referirse a los atributos
20    }
21
22    RestrictedPoint3D (int x, int y) { // Constructor
23        this (x, y, 0); // Se llama al otro constructor
24    }
25 }

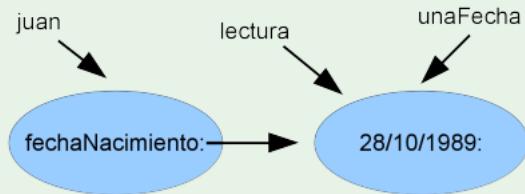
```

- Para los **consultores y modificadores** se hace igual que **C++**, pero **CUIDADO** que se puede **modificar el valor** en **cualquier momento** al trabajar con **referencias**

▼ Ejemplo **ERROR FOR THAT**

Java: Asignación y devolución de referencias

```
1 class Persona {  
2     private GregorianCalendar fechaNacimiento;  
3  
4     Persona (GregorianCalendar nace) {  
5         fechaNacimiento = nace;  
6     }  
7  
8     GregorianCalendar getFechaNacimiento () {  
9         return fechaNacimiento;  
10    }  
11  
12    // . . .  
13 }  
14  
15 GregorianCalendar unaFecha = new GregorianCalendar (1989,10,28);  
16  
17 Persona juan = new Persona (unaFecha);  
18 System.out.println(juan.toString()); // Nací el 28/10/1989  
19  
20 GregorianCalendar lectura = juan.getFechaNacimiento();  
21 lectura.set (1985,5,13);  
22 System.out.println(juan.toString()); // Nací el 13/5/1985  
23 unaFecha.set (2001,1,1);  
24 System.out.println(juan.toString()); // Nací el 1/1/2001
```



Paquetes

- Para añadir paquetes que estén automáticamente en java se hace como

import java.util.<paquete>

- Para añadir archivos que NO estén en el mismo paquete se hace así

import <carpeta>.<pwd>.<archivo>

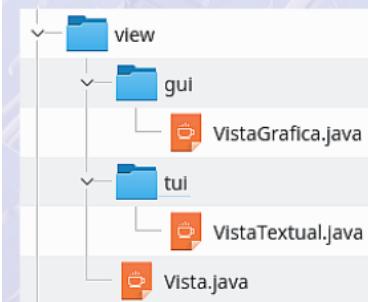
- Se puede tener dos clases con mismo nombre en paquetes distintos

Cada paquete es independiente, aunque con el nombre pareza subpaquete (NO EXISTEN subpaquetes)

▼ Ejemplo

Paquete: view

```
1 package view;  
2 public 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 }
```

← Estructura de carpetas y archivos

★ ¿Se puede quitar la palabra **public**?

Herencia

- Para **declarar** una **clase hija** de una **clase padre** es así

```
class <padre>{  
    ...  
}  
class <hija> extends <padre>{  
    ...  
}
```

- La **pseudovariable super** puede **referenciar** al **constructor de la clase padre** en el **constructor de la clase hija SOLO** si se **usa** en la **primera línea** como **super(<params>)**

- **Super** puede **llamar** a **cualquier método** de la **clase padre** (**NO** recomendable **llamar** a **métodos** **distintos** del **actual**) así **super.<metodo>(<params>)**

▼ Ejemplo

Java: super en métodos

```
1 int metodo1 (int i, int j) {  
2     int a = super.metodo1 (i, j);  
3     // NO recomendable si se ha redefinido  
        metodo2  
4     // Totalmente innecesario si no se ha  
        redefinido metodo2  
5     int b = super.metodo2 ();  
6     return (a+b);  
7 }
```



No usarlo para llamar a métodos
distintos del actual

Java: super en constructor

```
1 class Persona {  
2     private String nombre;  
3     Persona (String nombre) {  
4         this.nombre = nombre;  
5     }  
6 }  
7 class Profesor extends Persona {  
8     private String asignatura;  
9     Profesor (String nomb, String asign) {  
10        super (nomb);           // primera linea  
11        asignatura = asign;  
12    }  
13 }
```



- TODAS las **clases heredan** de Object

- Se pueden **redefinir** (o **sobrecargar**) los **métodos instancia** en la **clase hija** con **misma cabecera** (se usará **este** cuando se **llame**) y pueden **variarse** los **parámetros**, poner **@Override** arriba así

```
@Override  
<tipo> <nombre> (<params>){  
    ...  
}
```

- Se usa **@Override** cuando **sobrecargo** un **método instancia**, pasar de **privado a publico** (padre a hija) o **privado a privado** (padre a hija) es **definir un nuevo método**
- NO** se **sobrecargan** **métodos de clase**

▼ Ejemplo

tengo un metodo exactamente con la misma cabecera redefinido abajo, en el momento que paso de private a public o dejo private (en clase hija) no estoy redefiniendo sino definiendo otro metodo totalmente distinto

Java: Anotación @Override

```
1 @Override      // Cada método redefinido debe llevarla  
2 int metodo (int parametro) { . . . }
```

- Se puede **redefinir**, es decir, crear un **nuevo método** con **más parámetros** (en el caso que se den los **parámetros necesarios** para **solo** el **método padre** se usa **este**)

- NO se pueden **redefinir** **métodos final**
- Para **heredar** el **constructor** de la **clase padre** se hereda el **constructor sin parametros**
 - Si quiero **heredar** el **constructor con parametros** debo **crearlo** en la **hija** y llamar a **super**
 - Da **error** si intento **construir** una clase **hija** si **NO** tiene **constructor** y el **padre** **NO** tiene el **sin parametros**

Visibilidad

- La **visibilidad por defecto** es **visibilidad de paquete**, que implica **privado out paquete** y **público in paquete**
- Se puede acceder a **elementos privados** (de **instancia** y de **clase**) de una **instancia distinta** si es **dentro** de la **misma clase**
- La **visibilidad protected** es == **publico** en el **mismo paquete**
- Para **dentro** de una **subclase** **acceso libre** a **elementos** (**instancia** y **clase**) de una clase **padre**
- Para **instancias** de **subclases** (< **NO** ≤) el **atributo** o **método** (**instancia** y **clase**) **protected** de **padre** es **visible** (**independiente del paquete**), pero si es **abuelo→padre** **NO** se puede **acceder** desde **abuelo**
- Dada **otra instancia** para acceder a **elementos** (desde **ámbito de instancia** y **de clase**) debe
 - **clase** de esa **instancia misma** o **subclase de la clase** donde **estamos**, debe **ser-un-yo**
 - **elemento** **accedido** declarado en **misma clase** o **superclase** de la **clase** donde **estamos**
- Si tengo un **elemento privado** en **clase padre** **NO** pueden **acceder** a él en **clase hija** (desde **ámbito instancia** y **ámbito clase**), aun pasando como **parámetro** a una **instancia de clase padre o hija** (o **subclases**) (**NO** estamos en **clase padre** **NO** se **autoriza**)

▼ Ejemplo

Java: Acceso a elementos de otra clase

```
1 package unPaquete;
2
3 public class Padre {
4     private int privado;
5     protected int protegido;
6     int paquete;
7     public int publico;
8
9     public void testInstanciaPadre (Padre o) {
10         System.out.println (o.privado);
11         System.out.println (o.protegido);
12         System.out.println (o.paquete);
13         System.out.println (o.publico);
14     }
15
16    public static void testClasePadre (Padre o) {
17        System.out.println (o.privado);
18        System.out.println (o.protegido);
19        System.out.println (o.paquete);
20        System.out.println (o.publico);
21    }
22 }
```

Java: Acceso a instancia de la superclase

```
1 package unPaquete;
2
3 public class HijaPaquete extends Padre{
4
5     public void testInstanciaHijaPaquete (Padre o) {
6         System.out.println (privado);
7         System.out.println (o.privado);
8
9         System.out.println (protegido);
10        System.out.println (o.protegido);
11
12        System.out.println (o.paquete);
13        System.out.println (o.publico);
14    }
15
16    public static void testClaseHijaPaquete (Padre o) {
17        System.out.println (o.privado);
18
19        System.out.println (o.protegido);
20
21        System.out.println (o.paquete);
22        System.out.println (o.publico);
23    }
24 }
```

- Si tengo un **elemento de paquete** en **clase padre** podré **acceder** desde **clase hija** si está en **mismo paquete** sino **NO**

▼ Ejemplo

Java: Acceso a instancia de la misma clase, los elementos heredados se declaran en otro paquete

```
1 package otroPaquete;
2
3 public class HijaOtroPaquete extends Padre {
4
5     // El mismo código cambiando solo el tipo del parámetro
6
7     public void testInstanciaHijaOtroPaquete (HijaOtroPaquete o) {
8         System.out.println(o.privado);
9         System.out.println(o.protegido);
10        System.out.println(o.paquete);
11        System.out.println(o.publico);
12    }
13
14    public static void testClaseHijaOtroPaquete (HijaOtroPaquete o) {
15        System.out.println(o.privado);
16        System.out.println(o.protegido);
17        System.out.println(o.paquete);
18        System.out.println(o.publico);
19    }
20 }
```

▼ Ejemplo

Java: Acceso a instancia de la superclase los elementos heredados se declaran en otro paquete

```
1 package otroPaquete;
2
3 public class NietaOtroPaquete extends HijaOtroPaquete {
4
5     // Ahora probamos con un parámetro de la superclase
6
7     public void testInstanciaNietaOtroPaquete (HijaOtroPaquete o) {
8         System.out.println (o.privado);
9         System.out.println (o.protegido);
10        System.out.println (o.paquete);
11        System.out.println (o.publico);
12    }
13
14    public static void testClaseNietaOtroPaquete (HijaOtroPaquete o) {
15        System.out.println (o.privado);
16        System.out.println (o.protegido);
17        System.out.println (o.paquete);
18        System.out.println (o.publico);
19    }
20 }
```

- Puedo acceder a **elementos publicos** de **clase** desde **fuera** con **<clase>.**
<elemento>

Clases abstractas

RUBY

- Para **ejecutar** **código** (**NO compila** ruby)

```
ruby <archivo>
```

- Poner en **todos los programas** **#encoding: UTF-8**
- Incluso los **datos simples** son **punteros**, pero al **cambiarse** se **crea un nuevo objeto** al que **apuntará la variable** (string, integer, char, etc)

▼ Ejemplo

- Se puede poner **5.times** ...
- NO** hay **tipos** para las **variables**, de modo que **cualquier variable** puede tomar **cualquier valor** y referenciar a **cualquier objeto** (**lenguaje interpretado**)
- Si nos **interesa** usar **tipo float** y se pasa un **parámetro** poner **<param>.to_f**
- Destacar que para **mostrar calculos** de **coma flotante** hay que **utilizar** coma flotante

▼ Ejemplo

```
puts 3/2 #muestra 1  
puts 3.0 /2 muestra 1,5
```

- Para **escribir en consola** es así

```
puts "<texto>" + @nombre
```

- Para **incluir** en un **string** una **variable** hacerlo así

```
" <texto> #{<var>} <texto> "
```

▼ Ejemplo

Ruby: Ejemplo básico

```
1 #encoding : UTF-8
2 module Basico
3   module ColorPelo
4     MORENO= :moreno
5     CASTAÑO= :castaño
6     RUBIO= :rubio
7     PELIROJO= :pe Ricojo
8   end
9
10  class Persona
11    def initialize(n,e,p) # "constructor"
12      # Atributos de instancia (son privados)
13      @nombre=n
14      @edad=e
15      @pelo=p
16    end
17
18    public # aunque los métodos son públicos por defecto
19    def saluda # Método público de instancia
20      puts "Hola, soy "+@nombre
21    end
22  end
23
24  p=Persona.new( "Pepe" ,10 ,ColorPelo ::RUBIO)
25  p.saluda
26 end
```

- Para **hacer comentarios** de varias líneas hacer esto

```
=begin
<comentarios>
=end
```

Podemos poner `<variable>.class` para saber el `tipo de variables` (**NO RECOMENDABLE**)

El método `<objeto>.inspect` nos da **información** del `objeto`

Podemos poner `ifs` en `una línea`

```
# Sólo hace <algo> si se cumple la condición  
<algo> if <condición>
```

El método `<int>.times` sirve para iterar tantas veces como `<int>`, esta es su sintaxis

```
<int>.times do |<var>|  
    # código  
end
```

Enumerados

Se crean en un archivo aparte con la cabecera `module <nombre>` y su sintaxis es está

```
module <nombre>  
    # El símbolo es con lo que se comparará el enumerado <nombre>  
    # <nombre1> va en mayúsculas  
    <nombre1> = :<símbolo>  
    ...  
end
```

Se llama así `<nombre>::<nombre1>` si está `dentro` del **mismo modulo** ("paquete") sino es así `<module_enumerate>::<nombre>::<nombre1>`

Módulos

¡IMPORTANTE! Al `principio de cada archivo` poner el **modulo** al que `pertenece`
`module <modulo>` (poner al `final` su correspondiente `end`)

- **Facilita acceso** a otros archivos así `<clase>.new`
- Si **NO** se pone en `algún archivo` se haría así `<modulo>::<clase>.new` `desde ese archivo`
 - `Siempre` se **abren** (**module**) y se **cierran** (**end**)
 - Para **añadir ficheros** en otros ficheros poner **require_relative** '`<pwd>/<fichero>`' solo `si es necesario`
 - Si hay `modulos` **dentro** de `módulos`
 - Podemos **copiar** el `contenido de un módulo` con **include <modulo>** si está en `otro archivo` (poner al inicio **require_relative '<archivo>'** sin `.rb`)

▼ Ejemplo

Ejemplo: Ruby

```

1 module Externo
2   class A
3   end
4
5   module Interno
6     class B
7     end
8   end
9 end
10
11 module Test
12   def test
13     puts "Testeando"
14   end
15 end
16
17 class C
18   include Test  # Literalmente, se copia el contenido del módulo Test
19 end
20
21 a = Externo::A.new
22 b = Externo::Interno::B.new
23 c = C.new
24 c.test

```



- Ruby solo **procesará** la **clase x** si se ha `incluido` antes

▼ Ejemplo MAL

: cosa.rb

```
1 class Cosa
2   @@Maximo = 3
3
4   attr_reader :nombre
5
6   def initialize (unNombre)
7     @nombre = unNombre
8   end
9
10  def self.Maximo
11    @@Maximo
12  end
13 end
```

: persona.rb

```
1 require_relative 'cosa' # por línea 4
2
3 class Persona
4   @@MaximoPermitido = Cosa.Maximo
5
6   def initialize (unNombre)
7     @nombre = unNombre
8     @cosas = []
9   end
10
11  def otraCosaMas (unaCosa)
12    if @cosas.size < @@MaximoPermitido
13      @cosas << unaCosa
14    end
15  end
16
17  def to_s
18    salida = "Me llamo #{@nombre} y
19      tengo:\n"
20    for unaCosa in @cosas do
21      salida += "- #{unaCosa.nombre}\n"
22    end
23    salida
24  end
```

: principal.rb

```
1 require_relative 'cosa' # por línea 4
2 require_relative 'persona' # por línea 5
3
4 mochila = Cosa.new("Mochila")
5 juan = Persona.new("Juan")
6 juan.otraCosaMas (mochila)
7 puts juan.to_s
```

: cosa.rb

```
1 # Se añade un require_relative innecesario
2 # No se menciona la clase Persona en este archivo
3
4 require_relative 'persona'
5
6 class Cosa
7   # La clase se define igual que en el ejemplo anterior
8 end
9 # No cambia nada más en ningún otro archivo
```

Ejecución: Mensaje de error obtenido

```
persona.rb:4:in `<class:Persona>': uninitialized constant Persona::Cosa (NameError)
```

- La clase Persona espera Cosa que no esta definida todavía

Atributos y métodos

- Tienen dos tipos de **atributos de clase**: **atributos de clase** y **atributos de instancia de la clase**

▼ Ejemplo

Ruby: Confusión entre atributos

```
1 class Clase
2   @@variable = "De clase"
3   @variable = "De instancia de la clase"
4
5   def initialize
6     @variable = "De instancia"
7   end
8
9   def muestraValores
10    puts @@variable
11    puts @variable
12  end
13
14  def self.muestraValores
15    puts @@variable
16    puts @variable
17  end
18 end
19
20 objeto = Clase.new
21 objeto.muestraValores
22 Clase.muestraValores
```

- En los **métodos de clase** se alude a **atributos de instancia de clase**, en los **métodos de instancia** se alude a los **atributos de instancia**
- Así se ponen los **métodos de clase** y los **atributos de clase**

```
class <nombre>
  @@<NOMBRE> = <valor> # Atributo de clase
  @<NOMBRE> = <valor> # Atributo de instancia de clase
  <NOMBRE> = <valor> # Constante (NO RECOMENDABLE)

  # Método de clase, se puede utilizar para devolver atributo
  # de instancia de clase, sino no se puede acceder a él
  def sel.<nombre>
    <calculos>
    <NOMBRE> = <nuevo_valor> # ERROR no se puede cambiar
    # el resto de atributos (arriba definidos) si se pueden
```

```

    end
    ...
end

# Llamada a métodos de clase
<nombre_clase>.<metodo_clase>
# Acceso a constantes son publicas
<nombre_clase>::<constante>

```

- Desde **dentro de la clase** para **llamar** a **métodos de instancia** se hace **<metodo>** (o **self.<metodo>**) y para **métodos de clase** **self.class.<metodo>** o **<nom_clase>.<metodo>**
- TODOS** los métodos **devuelven** la instancia del **último calculo**
- NO** existe **++** sino **+1**

▼ Ejemplo

```

class Persona
    # Crea un método getter para :nombre y un método setter para :nombre
    attr_reader :nombre
    attr_writer :edad
    # Crea tanto un getter como un setter para :direccion
    attr_accessor :direccion

    def initialize(nombre, edad, direccion)
        @nombre = nombre
        @edad = edad
        @direccion = direccion
    end
end

persona = Persona.new('Juan', 30, 'Calle Falsa 123')
puts persona.nombre      # => Juan
persona.edad = 31         # Cambia la edad de Juan a 31

```

```

persona.direccion = 'Calle Verdadera 456' # Cambia la dirección
puts persona.direccion      # => Calle Verdadera 456

# Intentar acceder directamente a @edad desde fuera de la clase
# puts persona.edad          # Error: undefined method `edad' for ...

```

- Los `métodos declarados` son **por defecto públicos** para `otra visibilidad` así

```

private
  def <metodo>
    ...
  end
  ... # El resto de métodos debajo serían privados
public
  def <metodo>
    ...
  end
# Otra forma (NO RECOMENDABLE)
public :<metodo>, :<metodo>
private: ...

```

- Una **instancia NO** puede `llamar` a **métodos de clase**

Constructores

- Declara `public o private` **initialize NO** cambia nada, **OBLIGATORIO** usar `new` (**NO** deja `llamar directamente` a **initialize**)
- Así se usa el **constructor de la clase (initialize)**, este es `llamado por new`

```

<clase>.new(<variables>

<clase>
  def initialize (<variables>) # Constructor
  ...

```

```
    end  
end
```

- Hay que poner **@<variables>** a las **instancias privadas** de los **objetos**
- Para **poner** **privado** los **métodos de clase** así **private_class_method :**
<metodo>, ...
- Podemos **crear varios constructores**

```
class <nombre>  
    # Se suele poner nombre= new_<texto>  
    def self.<nombre>  
        new (<variables>)  
        # No poner def de atributos (es def de atributos de clase)  
        # por estar en ámbito de clase)  
        # @<atributo>= ...  
    end  
end  
# se suele poner privado el new por defecto  
private_class_method :new
```

▼ Ejemplo ERROR COMUN

```

1 class RestrictedPoint3D
2
3 # Forma ERRÓNEA de implementar estos constructores
4
5 def self.new_3D(x,y,z) # método de clase
6   @x = restric_to_range (x)
7   @y = restric_to_range (y)
8   @z = restric_to_range (z)
9 end
10
11 def self.new_2D(x,y)    # método de clase
12   @x = restric_to_range (x)
13   @y = restric_to_range (y)
14   @z = 0
15 end
16
17 private_class_method :new # pasa a ser privado
18 end

```

- esto hace cosas distintas pues @x, @y, @z son atributos de instancia de la clase y no atributos de instancias (los de los objetos)
- Si encontramos un parámetro con *<parámetro> (llamado **splat parameter**) indica que dicho parámetro **recogerá** el **resto de argumentos adicionales** **pasados** **al método** y se **almacenarán** en un **array**

▼ Ejemplo

```

1 def initialize (x, y, *z)
2   # *z es un array con el resto de parámetros que se pasen
3
4   @x = restric_to_range (x)
5   @y = restric_to_range (y)
6   if (z.size != 0) then
7     z_param = z[0]
8   else
9     z_param = 0
10  end
11  @z = restric_to_range (z_param)
12 end
13
14 # En algún lugar fuera de la clase ...
15
16 puts RestrictedPoint3D.new(1,2,3,4,5,6).inspect
17
18 # los parámetros extra son ignorados

```

- Se pueden **poner** **valores por defecto** en los **métodos**
- Si encontramos **métodos** con **cabecera** donde aparece **<parámetro>: <valor_defecto>** (**llamado parámetros nombrados**) indica que se puede **cambiar** el **orden de los parámetros** **indicandolo** así **<clase_u_objeto>. <metodo> (<parámetro>:<valor>, ...)**

▼ Ejemplo

```

1 # Parámetros nombrados con valores por defecto
2
3 def initialize (x:, y:, z:0)
4   @x = restric_to_range (x)
5   @y = restric_to_range (y)
6   @z = restric_to_range (z)
7 end
8
9 # En algún lugar fuera de la clase ...
10
11 puts RestrictedPoint3D.new(x:-1, y:101, z:-2000).inspect
12
13 # Puedo cambiar el orden
14 puts RestrictedPoint3D.new(y:2, z:3, x:1).inspect
15
16 puts RestrictedPoint3D.new(x:1, y:99).inspect

```

los dos puntos después
parámetros, como ver

Consultores y modificadores

- Se pueden crear **metodos get y set** así

```
# Crea un método getter y un método setter para
attr_reader :<atributo>
attr_writer :<atributo>
# Crea tanto un getter como un setter
attr_accessor :<atributo>
```

- Para **crearlos a mano** serían así

```
...
# método get
def <atributo>
  @<atributo>
end

# método set, funciona así porque su llamada es
# <objeto>.<atributo>=<parámetro> (Persona.edad=15)
def <atributo>=[()<parámetro>[]]
  @<atributo>=<parámetro>
end

# método para instancias de clase, es igual
# NO se puede attr_...
def self.<atributo_class>...
  ...
end
```

Herencia

- Se pueden **redefinir** los **métodos** en la **clase hija** con **misma cabecera** (se usará **este** cuando se **llame**)
- Para **declarar** una **clase hija** de una **clase padre** es así

```

class <padre>
  ...
end
class <hija> < <padre>
  ...
end

```

- La **pseudovariable super** `accede` al **método padre** cuando estamos en el **MISMO** `método` de la **clase hija**
- Si el `método hijo redefinido` tiene **mismos parámetros** que el `padre` se puede llamar a **super solo** y **automaticamente** coge el los `parámetros`, aun así puedo poner **super(<parámetros>)**

▼ Ejemplo

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

```



- TODAS** las `clases heredan` de `Object`
- NO** hay **sobrecarga**, `siempre` se **redefine**

Visibilidad

- Los **atributos** son **privados** **NO** se puede `cambiar` (igual que `initialize`)
- Los **métodos por defecto** son **públicos**
- Para acceder a **elementos privados** desde `clase` (`código`) **SOLO** puedo hacerlo con `self.<elemento>`, es decir, **NO** puedo `acceder` a **elementos privados de instancia** de una `instancia` de mi `misma clase` dentro de ella

- Para acceder a **elementos protected** desde `clase` (`código`) puedo hacerlo con `self.<elemento>` (`mis elementos`) o dada **otra instancia** puedo acceder si `clase del código` que invoca (`nuestra`) es la **misma**, o una **subclase**, de la `clase donde se declaró` dicho `elemento`
- Fuera de `clase` (`código`) **NO** se puede `acceder` a **elementos protected**
- NO** puedo `acceder` a **métodos privados de clase** desde **ámbito de instancia**, **Sí** puedo hacerlo desde **ámbito de clase**
- NO** se puede `acceder` a **métodos privados de instancia** desde el **ámbito de clase**, **Sí** **método publico de instancia**
- Para `acceder` a **atributos de clase** desde `fuera` necesito **método get** (**NO** vale `attr_reader`), `igual` ocurre con `set`
- Si **B** `hereda` de **A**
 - desde un **ámbito de instancia** de **B** se puede `llamar` a **métodos de instancia privados** de **A** (**NO** si pasamos `instancia de misma clase`)
 - desde un **ámbito de clase** de **B** se puede `llamar` a **métodos de clase privados** de **A**

▼ Ejemplo

Ruby: Acceso a privados y protegidos

```
1  class Padre
2    private
3    def privado
4    end
5
6    protected
7    def protegido
8    end
9
10   public
11   def publico
12   end
13
14   def test(p)
15     privado
16     self.privado #Correcto solo a partir de Ruby 2.7
17     p.privado
18     protegido
19     self.protegido
20     p.protegido
21   end
22 end
```

Ruby: Acceso a privados y protegidos con relación de herencia

```
1  class Hija < Padre
2    def test(p)
3      privado
4      self.privado #Correcto solo a partir de Ruby 2.7
5      p.privado
6      protegido
7      self.protegido
8      p.protegido
9      publico
10     self.publico
11     p.publico
12   end
13 end
14
15 # Fuera de cualquier clase
16
17 Hija.new.test(Hija.new)
18 Hija.new.test(Padre.new)
19 h=Hija.new
20 h.privado
21 h.protegido
22 h.publico
```

- Los **atributos de clase** se pueden **acceder** desde **ámbito de instancia** (da igual la **subclase**)
- Las **clases e instancias** son de **distintas clases**
- Los **atributos de instancia de clase NO** se heredan y los **atributos de clase** se pueden **cambiar** por los **hijos** (son **compartidos**)