

# Tema 2



**Clases, objetos y mensajes**

# Objetivos generales



- Comprender el **concepto de objeto**.
- Entender la utilidad de las **clases como mecanismo de abstracción**.
- Conocer los elementos de **definición de una clase**.
- Conocer la sintaxis de un **diagrama de clases UML**.
- Comprender la **relación entre los objetos y las clases que estos instancian**, identificando al mismo tiempo las diferencias que existen entre ambos elementos.
- Conocer la diferencia entre **mensaje y método**.
- Entender el **envío de mensajes** entre objetos como el mecanismo básico de ejecución en un programa orientado a objetos.
- Conocer las **pseudovariables**.
- Saber traducir un **diagrama de clases UML** al “esqueleto de código” correspondiente a las clases que aparecen en él.
- Saber traducir un **diagrama de interacción UML** (de secuencia o de colaboración) a código.

# Contenidos



Lección	Título
2.1	Clases y Objetos: Conceptos básicos
2.2	Diagramas estructurales para la representación de clases
2.3	Diagramas de interacción entre objetos

[http://groups.diigo.com/group/pdoo\\_ugr](http://groups.diigo.com/group/pdoo_ugr)



# Lección 2.1

## **Clases y Objetos: Conceptos Básicos**

# Objetivos de aprendizaje



- Comprender el **concepto de objeto**.
- Apreciar la utilidad de las **clases** como **mecanismos de abstracción de objetos**.
- Conocer la **estructura interna** de una clase: atributos y métodos.
- Diferenciar correctamente entre **estado e identidad** de un objeto.
- Comprender la diferencia entre un **valor primitivo y un objeto**.
- Entender la diferencia entre los **atributos/métodos de clase** y los **atributos/métodos de instancia**.
- Conocer los **elementos de agrupación**.
- Diferenciar el uso de los métodos y variables según sus **especificadores de acceso**.
- Diferenciar los **tipos de colecciones**.
- Conocer la diferencia entre **mensaje y método**.

# Objetivos de aprendizaje (cont)



- Entender el **envío de mensajes** entre objetos como el mecanismo básico de ejecución en un programa orientado a objetos.
- Conocer las **pseudovariables**
- Conocer el **ciclo de vida** de un objeto.
- Ver que hay métodos **consultores y modificadores** de objetos.
- Entender cómo se **construyen y destruyen** objetos.
- Saber que las clases se pueden agrupar en **unidades organizativas** denominadas **paquetes**.
- Comprender la relevancia de establecer **especificadores de acceso** adecuados para los métodos.
- Entender que un **atributo** de un objeto puede ser a su vez otro objeto.
- Comprender la utilidad de las **colecciones** de objetos.

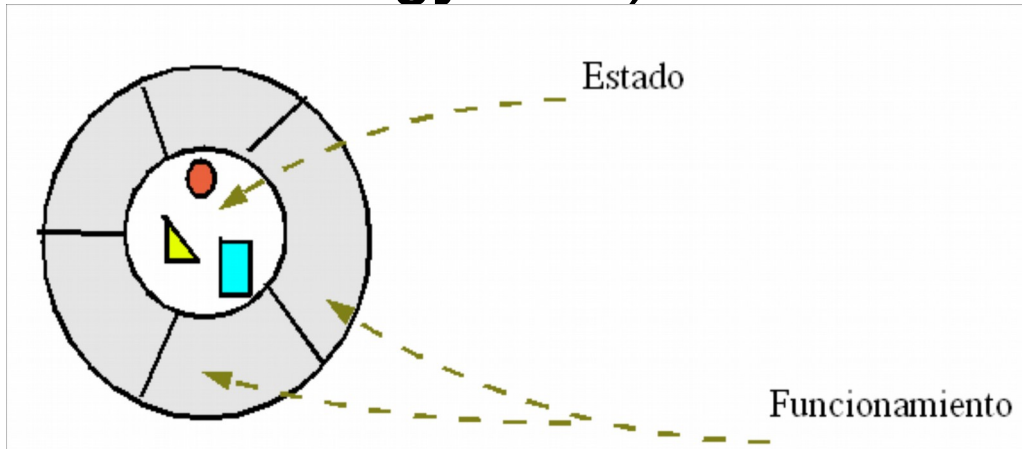
# Contenidos



1. Concepto de objeto.
2. Estado e identidad de objetos.
3. Concepto de clase.
4. Ámbito de los atributos.
5. Ámbito de los métodos.
6. Ciclo de vida de un objeto.
7. Constructores.
8. Destrucción.
9. Consultores y modificadores.
10. Elementos de agrupación.
11. Especificadores de acceso.
12. Agregaciones de objetos.
13. Colecciones.
14. Pseudovariables.
15. Envío de mensajes entre objetos.
16. Envío de mensajes a self en Ruby.

# 1. Concepto de objeto

- Entidad perfectamente delimitada, que encapsula **estado** y **funcionamiento** y posee una **identidad** (OMG 2001).
- Elemento, unidad o entidad individual e identificable, real o abstracta, con un papel bien definido en el dominio del problema (**Dictionary of Object Technology 1995**).



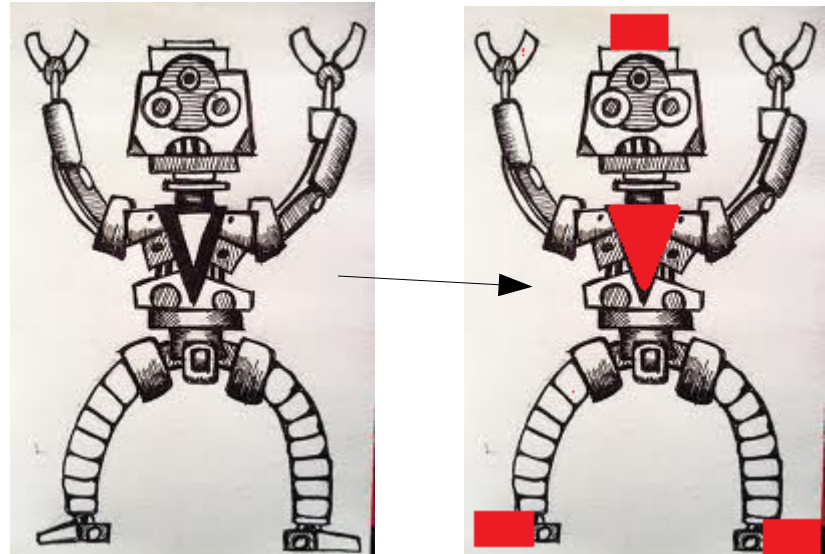


# 1. Concepto de objeto

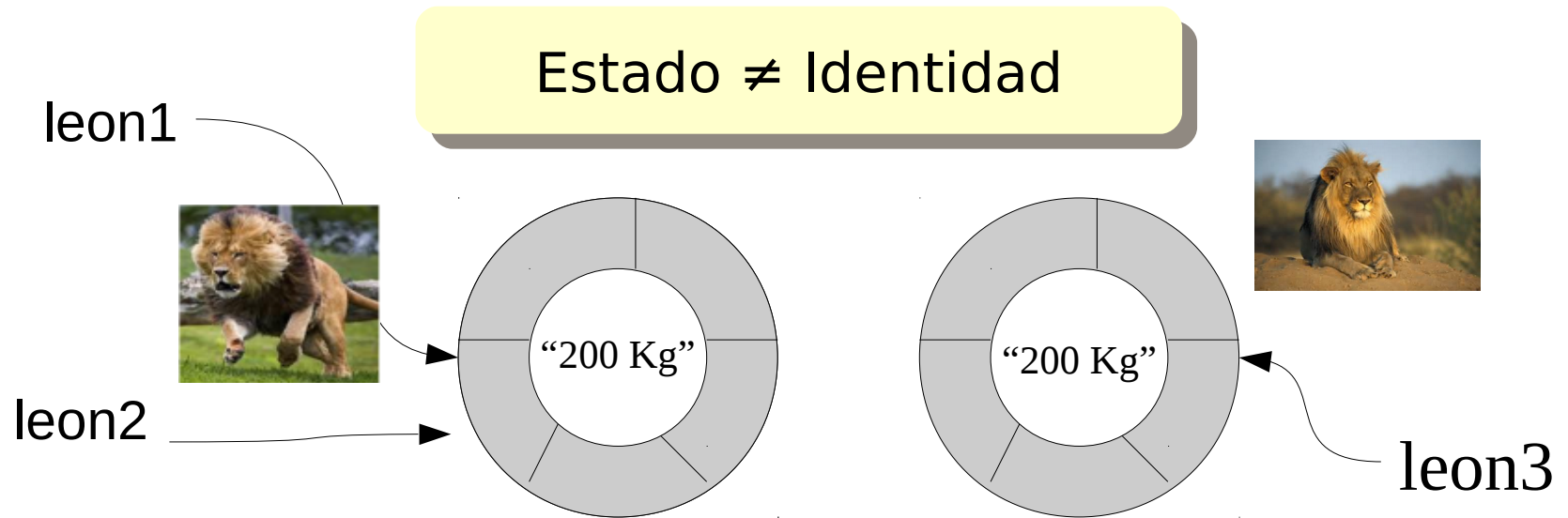
- La **identidad** es la propiedad que permite distinguir a un objeto de los demás.
  - Un objeto tiene identidad por el mero hecho de existir.
  - Aunque un objeto tenga características que lo pueden identificar de forma única, éstas no constituyen su identidad.
    - P.ej. el DNI de una persona lo identifica de forma única, pero no es su identidad.
    - Si hubiera dos personas con el mismo DNI, serían dos personas diferentes.

# 1. Concepto de objeto

- El **estado** del objeto lo determinan características observables o que pueden ser consultadas.
  - Dos objetos diferentes pueden tener el mismo estado.
  - El estado de un objeto puede variar a lo largo del tiempo.



## 2. Estado e identidad de objetos



### Identidad

leon1 **es idéntico a** leon2 (cierto)  
leon1 **es idéntico a** leon3 (falso)

### Estado

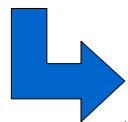
leon1 **es igual a** leon2 (cierto)  
leon1 **es igual a** leon3 (cierto)  
leon2 **es distinto de** leon3 (falso)

# 3. Concepto de clase

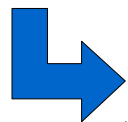
## Definición de la RAE:

Clase: Orden en que, con arreglo a determinadas condiciones o calidades, se consideran comprendidas diferentes personas o cosas.

Las clases son un mecanismo de abstracción sobre el **estado** y el **funcionamiento** de un conjunto de objetos.



Atributos



Métodos

Una clase es una especie de “molde” o plantilla para crear objetos.

Decimos que un objeto es una *instancia* de una determinada clase, o que *pertenece* a dicha clase.

# 4. Ámbito de los atributos

- **Atributos de instancia:** Almacenan los valores que configuran el estado de un objeto en particular.

Cada objeto tiene una copia distinta de cada atributo de instancia, las cuales se almacenan en posiciones de memoria diferentes. Así, cada objeto puede darle sus propios valores.

- **Atributos de clase:** Almacenan un único valor para todos los objetos de una clase.

El atributo está asociado a la clase, de forma que toma un valor único para todos los objetos de la clase y está almacenado en una única posición de memoria.

# 4. Ámbito de los atributos

Un ejemplo ilustrativo:

Clase: León

Atributos de instancia: peso, tamaño, edad

Atributos de clase: hábitat



Cada objeto de la clase León tendrá sus propios atributos *peso*, *tamaño* y *edad* a los que puede dar valores distintos a otros objetos. Sin embargo, todos los objetos comparten el atributo de clase *hábitat* que tomará el valor “sabana” para todos.



Un atributo de clase también podría ser un contador de ejemplares (*numeroLeones*), el cual se actualizaría con cada nacimiento o pérdida.

# 5. Ámbito de los métodos

Determinan el **comportamiento**

- **Métodos de instancia:**
  - Son métodos **asociados a los objetos** cuya invocación se realiza mediante **envío de mensajes** entre objetos. Pueden modificar los atributos de instancia. Pueden consultar los atributos de instancia y de clase (\*).
- **Métodos de clase:**
  - Son métodos **asociados a la clase** que se pueden invocar sin necesidad de crear ninguna instancia (ningún objeto). Pueden modificar los atributos de clase.



# 5. Ámbito de los métodos



## Un ejemplo realista:

Variables de  
instancia

Variable  
de clase

Uso de los  
métodos  
de clase

Métodos de  
instancia

Métodos  
de clase

```
public class Bicicleta {  
    private int marchas;  
    private int color;  
    private int numeroSerie;  
    private static int numeroDeBicicletas = 0;  
  
    public Bicicleta(int numeroMarchas, int unColor){  
        marchas = numeroMarchas;  
        color = unColor;  
        numeroSerie = Bicicleta.getNumeroDeBicicletas();  
        Bicicleta.incrementarNumeroDeBicicletas();  
    }  
    public int getColor() {  
        return color;  
    }  
    public int getNumeroSerie() {  
        return numeroSerie;  
    }  
    public static int getNumeroDeBicicletas() {  
        return numeroDeBicicletas;  
    }  
    public static void incrementarNumeroDeBicicletas(){  
        numeroDeBicicletas++;  
    }  
    ...  
}
```



# 5. Ámbito de los métodos

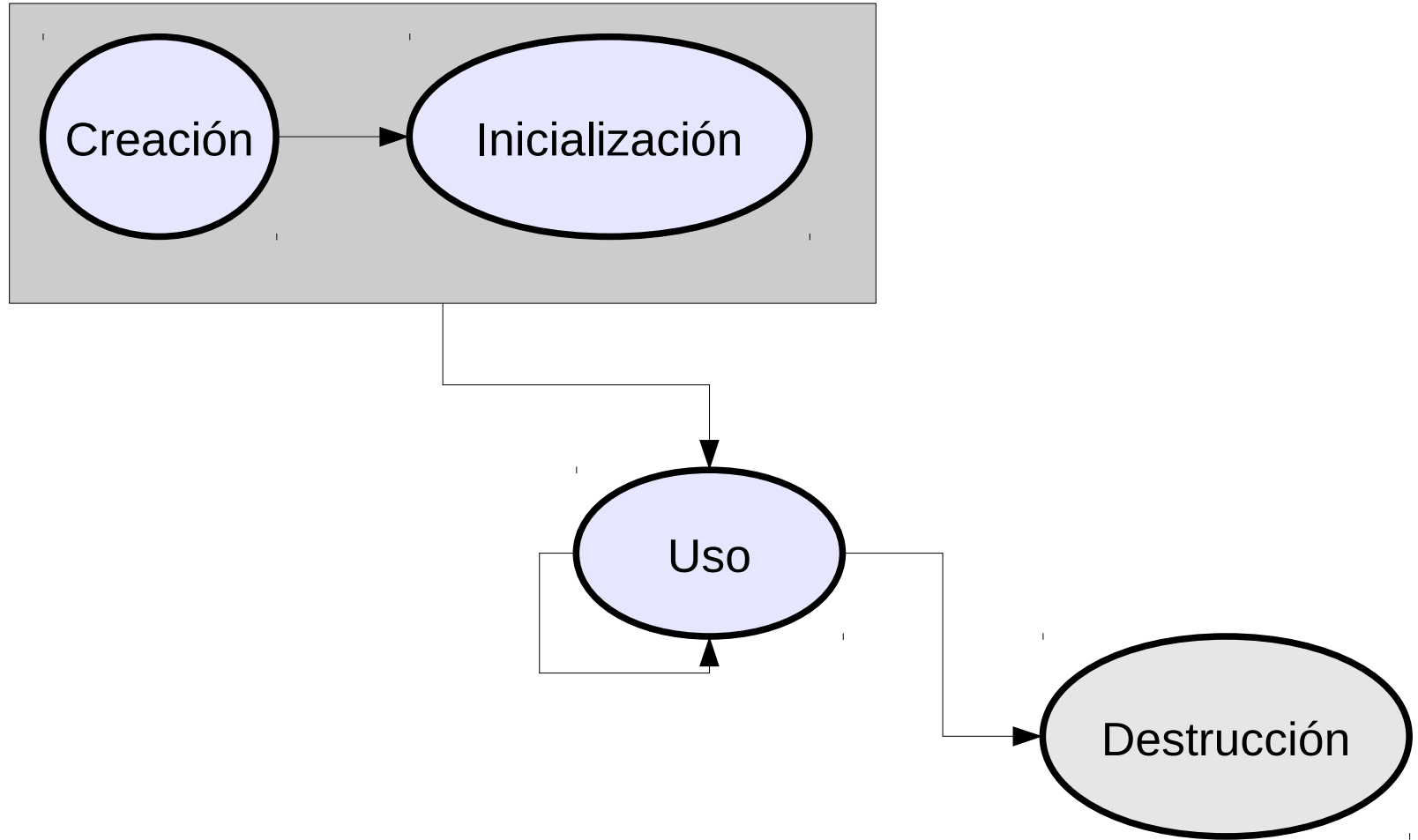


Probar código de ejemplo de Bicicleta en Ruby y Java

```
class Bicicleta
  @@numeroDeBicicletas = 0

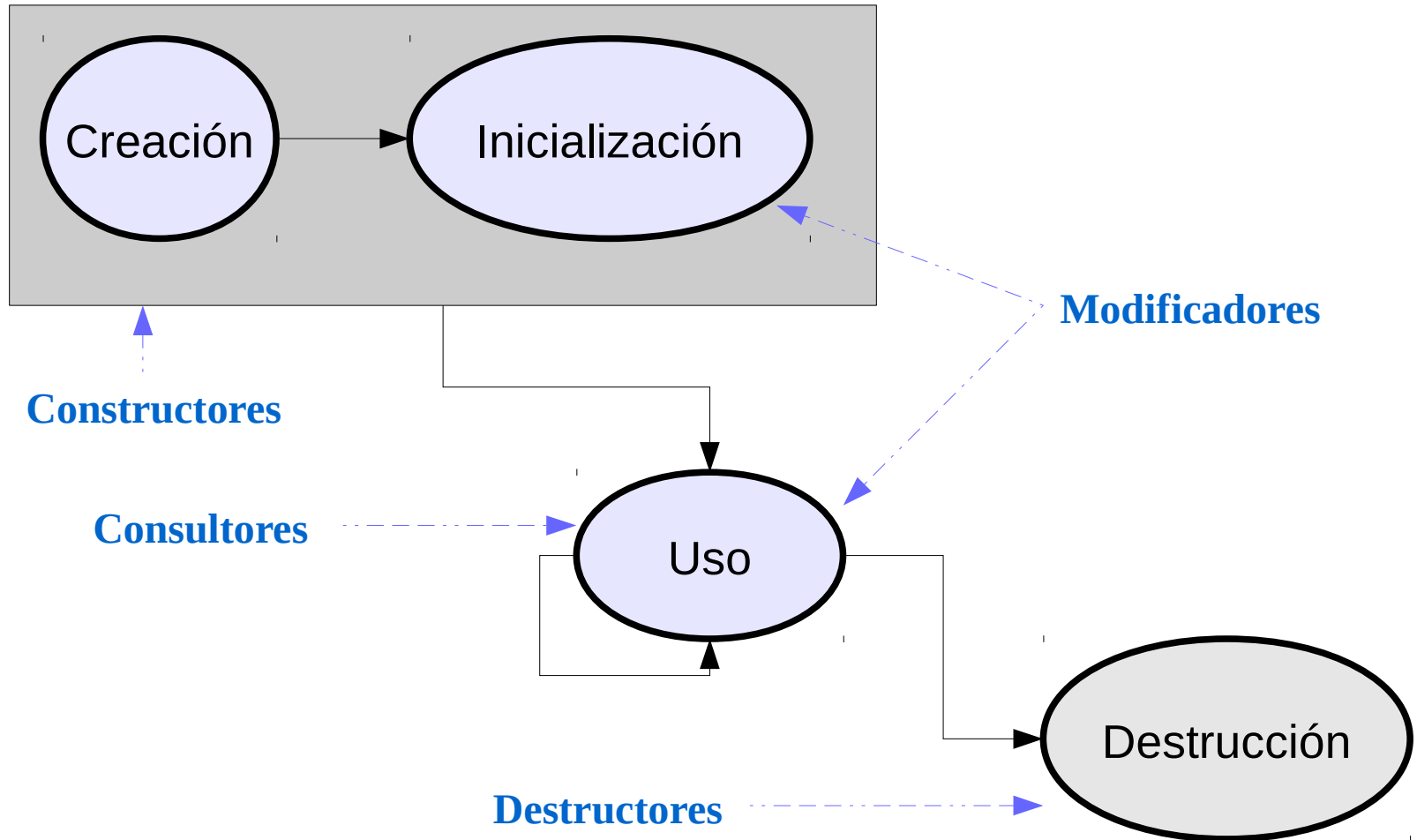
  def initialize(numeroMarchas, unColor)
    @marchas = numeroMarchas
    @color = unColor
    @numeroSerie = @@numeroDeBicicletas
    Bicicleta.incrementarNumeroDeBicicletas
  end
  def color
    @color
  end
  def numeroSerie
    @numeroSerie
  end
  def self.numeroDeBicicletas
    @@numeroDeBicicletas
  end
  def self.incrementarNumeroDeBicicletas
    @@numeroDeBicicletas = @@numeroDeBicicletas+1
  end
  ...
end
```

# 6.Ciclo de vida de un objeto



# 6.Ciclo de vida de un objeto

Tipos de métodos que intervienen:



# 7. Constructores

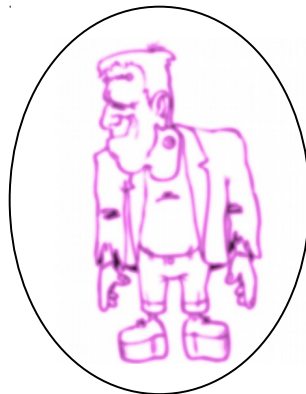
Tienen dos **propósitos específicos**:

- Crear instancias de la clase a la que pertenecen (invocando internamente a un método, que generalmente es *new*).
- Inicializar el estado del objeto recién creado.
- **Características comunes** de los constructores:
  - No son métodos de instancia.
  - No pueden especificar un valor de retorno (¡ni siquiera void!).

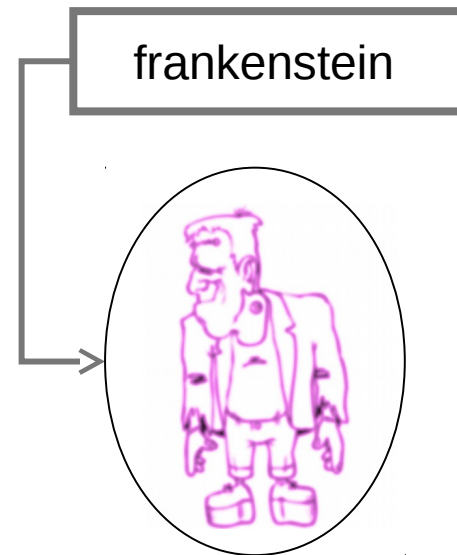
# 7. Constructores

El constructor se ejecuta (paso 1) antes de asignar una referencia al objeto (paso 2).

1. Crear un objeto e inicializar sus atributos:



2. Enlazar la referencia con el objeto:



# 7. Constructores

Características diferenciadoras de los constructores:

- Según el *nombre*:
  - En las clases-plantilla suelen tener el **mismo nombre que la clase** (p.ej. en Java).
  - En las clases-objeto pueden tener otro nombre, incluso **new**.
- Según *responsable*:
  - En las clases-plantilla, al constructor se le suele invocar usando la palabra reservada **new**.
  - En clases-objeto, al constructor se le suele invocar como **un método de instancia de la clase más**.

Ojo: no es lo mismo

# 7. Constructores

---

## Tipos de constructores:

- Constructor predeterminado
- Constructores definidos por el programador
  - Sin argumentos
  - Con argumentos

# 7. Constructores

## Constructor predeterminado:

No lo define el programador. Los atributos se inicializan al valor por defecto. Se invocan con *new*.

Ejemplo Java:



```
class MuertoViviente {  
  
    private float dedos_de_frente;  
  
    public void asustar () {  
        System.out.println("uuuhhh");  
    }  
}
```

Creación y uso de un objeto en Java:

```
MuertoViviente vampiro = new MuertoViviente();  
vampiro.asustar();
```

// dedos\_de\_frente tiene el valor 0.0

Ejemplo Ruby:



```
class MuertoViviente  
  
    attr_accessor :dedos_de_frente  
  
    def asustar  
        puts "uuuhhh"  
    end  
end
```

• Creación y uso de un objeto en Ruby

```
vampiro = MuertoViviente.new  
vampiro.asustar
```

# dedos\_de\_frente tiene el valor nil



# 7. Constructores

## Constructor predeterminado, peculiaridad de C++:

Ejemplo Java:

```
class MuertoViviente {
    private float dedos_de_frente;
    public void asustar () {
        System.out.println("uuuhhh");
    }
}
```



Ejemplo C++:

```
class MuertoViviente {
    private:
        float dedos_de_frente;
    public:
        void asustar() {
            cout<<"uuuhhh";
        }
}
```

Creación y uso de un objeto en Java:

```
MuertoViviente vampiro;
vampiro = new MuertoViviente();
vampiro.asustar();
```

Creación y uso de un objeto en C++:

```
MuertoViviente vampiro;
vampiro.asustar();
```

**Peculiaridad C++:** el objeto vampiro queda construido y puede ser usado inmediatamente

Sólo se ha declarado una referencia que no apunta a ningún objeto porque no se ha creado. Hay que crearlo expresamente o hacer que la referencia apunte a un objeto ya creado antes de poder usarlo.

**Ojo: no ocurre así con las referencias:**

```
MuertoViviente *vampiro_puntero; //Correcto, los punteros pueden declararse sin inicializarse
MuertoViviente &vampiro_referencia; //Error, las referencias hay que inicializarlas obligatoriamente
```

```
vampiro_puntero->asustar(); // error, no existe el objeto
vampiro_puntero = new MuertoViviente(); vampiro_puntero->asustar(); // correcto
MuertoViviente vampiro; // correcto, se usa el constructor por defecto
MuertoViviente &vampiro_referencia = vampiro; //correcto, la referencia queda inicializada
```

# 7. Constructores

Constructor sin argumentos creado por el programador:

Asigna a los objetos creados un mismo estado inicial, dando valores a sus atributos. Sustituye al constructor proporcionado por el lenguaje.



Ejemplo de constructor en **clase-plantilla** Java:

```
public class MuertoViviente {
    private float dedos_de_frente;

    public MuertoViviente(){
        setDedosDeFrente(4.5);
    }
    public void setDedosDeFrente(float ddf){
        dedos_de_frente = ddf;
    }
}
```

*Nombre igual que la clase*

Ejemplo de constructor en **clase-objeto** Ruby:

```
class MuertoViviente
    attr_accessor :dedos_de_frente

    def initialize()
        @dedos_de_frente=4.5
    end
end
```

*Initialize no es el constructor, pero se invoca cada vez que se construye un objeto*

*Invocación con new*

Invocación:

Java: MuertoViviente frankenstein = **new MuertoViviente()**;

Ruby: frankenstein = MuertoViviente.new



# 7. Constructores

Constructores con argumentos creados por el programador:

Cuando son invocados, el estado del objeto viene dado por el valor de los argumentos que se asignan a los atributos.

Ejemplo de constructor en **clase-plantilla** Java:



```
public class MuertoViviente {  
    private float dedos_de_frente;  
  
    public MuertoViviente(float ddf){  
        setDedosDeFrente(ddf);  
    }  
    public void setDedosDeFrente(float ddf){  
        dedos_de_frente = ddf;  
    }  
}
```

*Nombre  
igual que  
la clase*

Ejemplo de constructor en **clase-objeto** Ruby:



```
class MuertoViviente  
    attr_accessor :dedos_de_frente  
  
    def initialize(unFloat)  
        @dedos_de_frente=unFloat  
    end  
end
```

*Initialize no es el  
constructor, pero se  
invoca cada vez que  
se construye un  
objeto*

*Invocación con new*

Invocación:

Java: MuertoViviente frankenstein = **new MuertoViviente(2.0);**

Ruby: frankenstein = MuertoViviente.new(2.0)

# 7. Constructores

## Constructores: Peculiaridad de Ruby: solo un initialize



valor por defecto en parámetro y condiciones dentro del initialize:

```
class MuertoViviente
  attr_accessor :dedos_de_frente

  def initialize(unFloat=nil, unaEdad=4)
    if unFloat.nil?
      @dedos_de_frente=4.5
    else
      @dedos_de_frente=unFloat
    end
    @edad=unaEdad
  end
end
```

otros métodos de clase:

```
class MuertoViviente
  attr_accessor :dedos_de_frente

  def initialize(unFloat, unaEdad)
    @dedos_de_frente=unFloat
    @edad=unaEdad
  end

  def self.jovencito(unFloat)
    self.new(unFloat, 3)
  end
end
```

Invocación en Ruby:

```
frankenstein = MuertoViviente.new
zombi_1 = MuertoViviente.new(2.0)
zombi_2 = MuertoViviente.jovencito(3.3)
```



En Ruby solo puede definirse un *initialize*.

Pueden incluirse condiciones dentro del método o usar métodos de clase para invocar al constructor con diferentes parámetros.

¿quién tiene más “dedos\_de\_frente” ?

# 7. Constructores

Constructores creados por el programador. Ejemplo en Smalltalk.

Ejemplo de constructor en **clase-plantilla**  
Java:



```
public class MuertoViviente {

    private float dedos_de_frente;

    public MuertoViviente(float ddf){
        setDedosDeFrente(ddf);
    }

    public void setDedosDeFrente(float ddf){
        dedos_de_frente = ddf;
    }

}
```

*Nombre  
igual que  
la clase*

Ejemplo de constructor en **clase-objeto**  
SmallTalk:

```
class MuertoViviente
----- método de clase -----
    nuevoMV: unFloat
    ^ (self new) setDedosDeFrente: unFloat
-----
----- método de instancia de la clase -----

    setDedosDeFrente: unFloat
    dedos_de_frente = unFloat
-----
```

*Nombre del  
constructor  
distinto al  
de la clase*

*Invocación con "new"*

*Invocación como un  
método de clase*

Invocación:

Java: MuertoViviente frankenstein = **new MuertoViviente(2.0);**

Smalltalk: frankenstein := **MuertoViviente nuevoMV:2.0.**

# 7. Constructores



¿Qué **ventajas** aporta crear un constructor nuevo que dé valor a los atributos frente a usar el constructor proporcionado por el lenguaje, que inicializa a valores por defecto?

# 7. Constructores

Diferentes formas de asignar memoria para la creación de objetos:

- **Memoria dinámica:** en el montículo (heap)
  - Es la única posibilidad para muchos lenguajes OO (Java, ST, Ruby, PHP, O-C) y la más usual en C++.

Ejemplo en C++:

```
Frankenstein *franky=new Frankenstein(3.25);
```

- **Memoria estática:** en la pila (stack)
  - Lo permite C++, realizándose la ligadura de todos los métodos de forma estática.

Ejemplo en C++:

```
Frankenstein franky(3.25);
```

# 8. Destrucción de Objetos

Cuando un objeto **deja de ser referenciado** pueden ocurrir **dos situaciones**, dependiendo del lenguaje:

- **Recolector automático de basura:** Se destruye el objeto automáticamente, liberando el espacio en memoria (p.ej. Java, Smalltalk, Ruby, PHP).

Ejemplo Java: `MuertoViviente frankenstein = new MuertoViviente(2.0);  
frankenstein = new MuertoViviente(3.2);`

- **Liberación manual:** Aunque el objeto deje de estar referenciado, es necesario emplear un destructor para eliminar el objeto de la memoria (p.ej. C++, Objective C).

Ejemplos:	C++ , memoria dinámica	<code>delete frankenstein</code>
	Objective C	<code>frankenstein release</code>



# 9. Consultores y modificadores

- Se emplean para **conocer (consultores)** y **cambiar (modificadores)** el estado de los objetos.
- Existen los denominados **consultores/modificadores básicos**, que consultan o modifican directamente un único atributo.
  - **Java**: su nombre suele ser *getNombreAtributo* (consultores) y *setNombreAtributo* (modificadores), por lo que también se les conoce como métodos *get/set* o *getter/setter*.
  - **Ruby**: se les suele llamar con el mismo nombre que los atributos. No suelen definirse explícitamente sino que se hace uso de *attr\_accessor*, *attr\_writer* y *attr\_reader*

# 9. Consultores y modificadores

//Get en Java:



```
tipoAtributo getAtributo()
{
    return miAtributo;
}
```

//Set en Java:

```
void setAtributo(TipoAtributo valor)
{
    miAtributo=valor;
}
```

#Get en Ruby



attr\_reader :miAtributo

#Set en Ruby:

attr\_writer :miAtributo

#Get y Set en Ruby

attr\_accessor :miAtributo

(Detrás del nombre de la clase, cuando se crea)

# 9. Consultores y modificadores



//Ejemplo declaración en Java:



```
public class MuertoViviente {  
  
    private float dedos_de_frente;  
    int edad;  
  
    public MuertoViviente(float ddf){  
        setDedosDeFrente(ddf);  
        edad=0;}  
  
    public void setDedosDeFrente(float ddf){  
        dedos_de_frente = ddf;}  
  
    public float getDedosDeFrente(){  
        return dedos_de_frente;}  
}
```

// Ejemplo de uso de get y set en Java:

```
MuertoViviente zombi = new MuertoViviente(2.6)  
  
float var = zombi.getDedosDeFrente();  
zombi.setDedosDeFrente(8.3);
```

//Ejemplo declaración en Ruby:

```
class MuertoViviente  
    attr_accessor :dedos_de_frente  
  
    def initialize(unFloat)  
        @dedos_de_frente=unFloat  
        @edad=0  
    end  
end
```

// Ejemplo de uso de get y set en Ruby:

```
zombi= MuertoViviente.new(2.6)  
  
var= zombi.dedos_de_frente  
zombi.dedos_de_frente = 8.3
```



¿Cómo se sabe qué atributos tiene una clase en Ruby?

# 9. Consultores y modificadores

- Los consultores/modificadores básicos se deben utilizar en conjunción con los especificadores de acceso (punto 11 de esta lección) para garantizar que el acceso a los atributos se realiza siempre a través de estos métodos.
- CUIDADO: Los modificadores, e incluso los consultores superfluos, pueden permitir accesos no deseados al estado del objeto.



Modifica el ejemplo de las transparencias 18 y 19 (definición clase Bicicleta) para que todos los atributos tengan su método get/set y éstos se usen en el constructor.

# 10. Elementos de agrupación

- Las agrupaciones son otro elemento de encapsulamiento.
- Son de gran utilidad para la reutilización de código:
  - Existen agrupaciones estándar ya definidas con las clases, variables o métodos más utilizados.
  - Con la misma idea es posible hacer nuevas agrupaciones con las propias implementaciones.
- Las agrupaciones pueden anidarse entre sí (\*).
- Permiten además tener un nivel más de control de acceso (lo veremos a continuación).
- Las agrupaciones de Java se llaman **paquetes** y las de Ruby **módulos**.

# 10. Elementos de agrupación



## Paquetes de Java:

- Permiten agrupar **clases**.
- El dominio de los paquetes es plano. No se pueden anidar paquetes (solo a nivel de nombrado)
- **import** permite acceder de forma directa a clases de otro paquete.

```
package animales;
```

```
class Leon {...}
```

```
class Gallina {...}
```

---

```
package animales.reptiles;
```

```
class Serpiente {...}
```

---

```
package granja;
```

```
import animales.Gallina;
```

```
class Comedero{...}
```

# 10. Elementos de agrupación



## Módulos de Ruby:

- Permiten agrupar **clases, métodos, variables y trozos de código** que pueden ser usados por otras clases o módulos y/o ejecutarse directamente.
- Pueden agrupar a otros módulos.

```
module Animales
```

```
class Leon ... end
```

```
class Gallina ... end
```

```
def comer ... end
```

```
PLANETA= "Tierra"
```

```
module Reptiles
```

```
class Serpiente ... end
```

```
end
```

```
leon1=Leon.new
```

```
var=leon1.met(2)+8
```

```
end
```

# 10. Elementos de agrupación

## Módulos de Ruby:

- Para indicar el path del **fichero** donde está el código que necesitamos cargar: `require`
- `require_relative`: si el **fichero** a cargar está en la carpeta actual.
- `include` permite importar un **módulo** dentro de una clase.

```
module Cronometro  
  def empezar...end  
  def parar ... end  
end
```



crono.rb

```
-----  
require_relative "crono"  
class Reloj  
  include Cronometro  
  def mostrarHora...end  
end  
rel=Reloj.new  
rel.mostrarHora  
rel.empezar
```

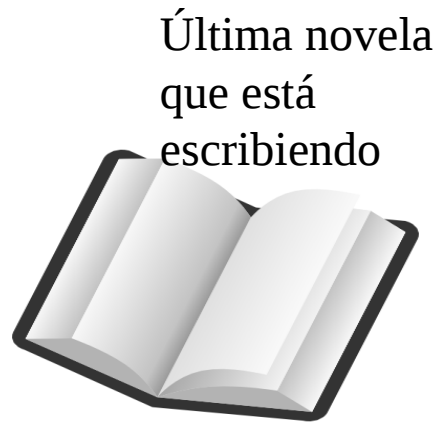
reloj.rb



# 11. Especificadores de acceso

Los especificadores de acceso sirven para restringir la visibilidad de clases, atributos y métodos.

Hay 3 especificadores de acceso: público, protegido y privado. En Java también está el especificador de paquete.



¿Con qué  
visibilidad  
tratarías los estos  
recursos de un  
escritor?



**Buena práctica de programación:** usar siempre el nivel de acceso más restrictivo posible en cada momento.

# 11. Especificadores de acceso

Especificadores de acceso en Java para atributos y métodos:

<i>Visible en:</i>	Mismo paquete		Otro paquete
	Clase	Otra	Otra
private	✓		
package	✓	✓	
protected	✓	✓	
public	✓	✓	✓

Si no se indica nada, la visibilidad por defecto en Java es de paquete (package) tanto en variables como en métodos.

# 11. Especificadores de acceso

Especificadores de acceso en Ruby para **métodos**:



<i>Visible en:</i>	Desde el propio objeto	Clase	Otra
private	✓		
protected	✓	✓	
public	✓	✓	✓

- La visibilidad de las **atributos de instancia y de clase** es **privada** en Ruby. La de las constantes es pública. No puede cambiarse.
- Por defecto los métodos son públicos.
- Cuando un método es privado solo puede ser invocado sin un receptor explícito (tampoco self).



# 11. Especificadores de acceso



```
class MuertoViviente
  attr_reader :dedos_de_frente
  def initialize(unFloat)
    @dedos_de_frente=unFloat
    @edad=30
  end
  def asustar
    'uuuuuuuuuuuuuu1'
  end
  def presentar
    "tengo " + self.dedos_de_frente.to_s +
    " dedos y " + miEdad.to_s + " lustros " + asustar
  end
  protected
  def comer
    'rico rico'
  end
  def miEdad
    @edad-5
  end
  private :miEdad, :asustar
end
```

```
otroZombi= MuertoViviente.new(2.2)
otroZombi.presentar
otroZombi.comer
otroZombi.asustar
```



¿Hay algún error arriba? Justifica dependiendo de dónde esté ese código

¿Qué visibilidad tienen los siguientes métodos?:

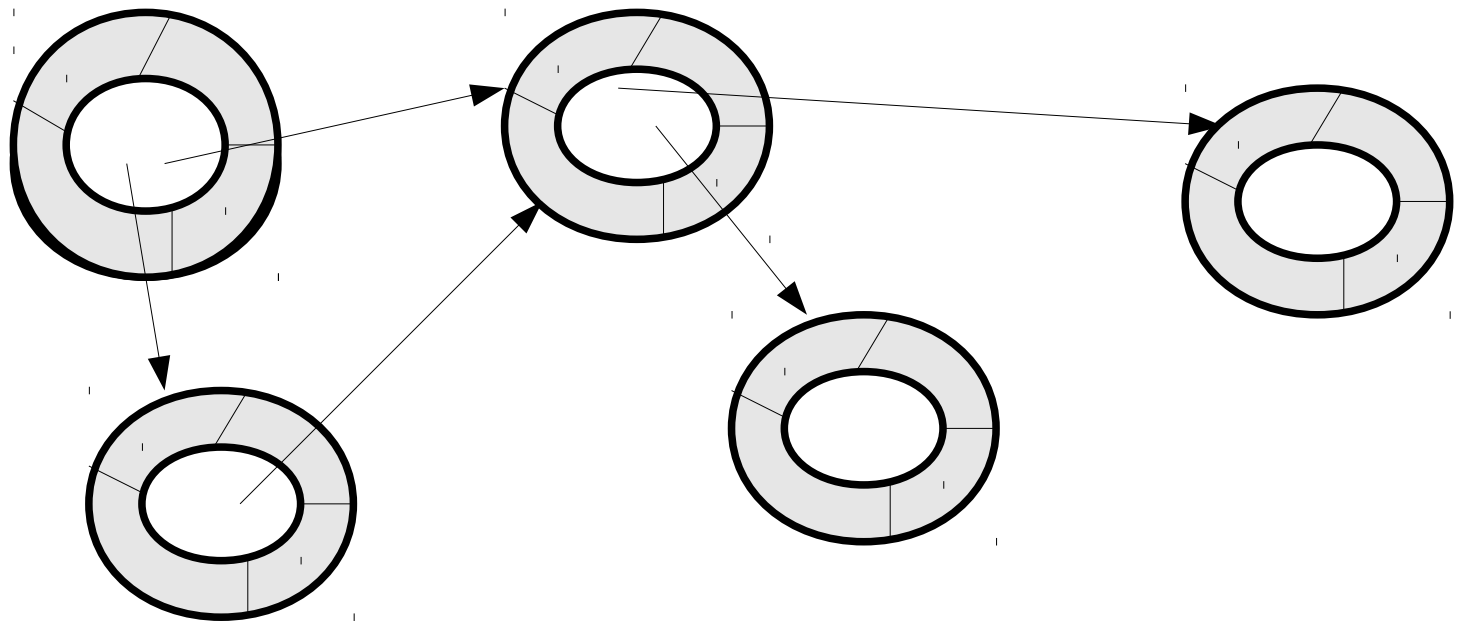
- presentar
- miEdad
- dedos\_de\_frente

Dos formas alternativas de especificar la visibilidad:

- Antes de los métodos afectados
- Una vez ya definidos

# 12. Agregación de Objetos

El estado de un objeto puede estar formado por un conjunto de objetos y éstos a su vez estar compuestos por otros objetos y así sucesivamente, hasta llegar a objetos cuyo estado está formado por valores simples, formando un grafo dirigido de relaciones entre objetos.



# 12. Agregación de Objetos

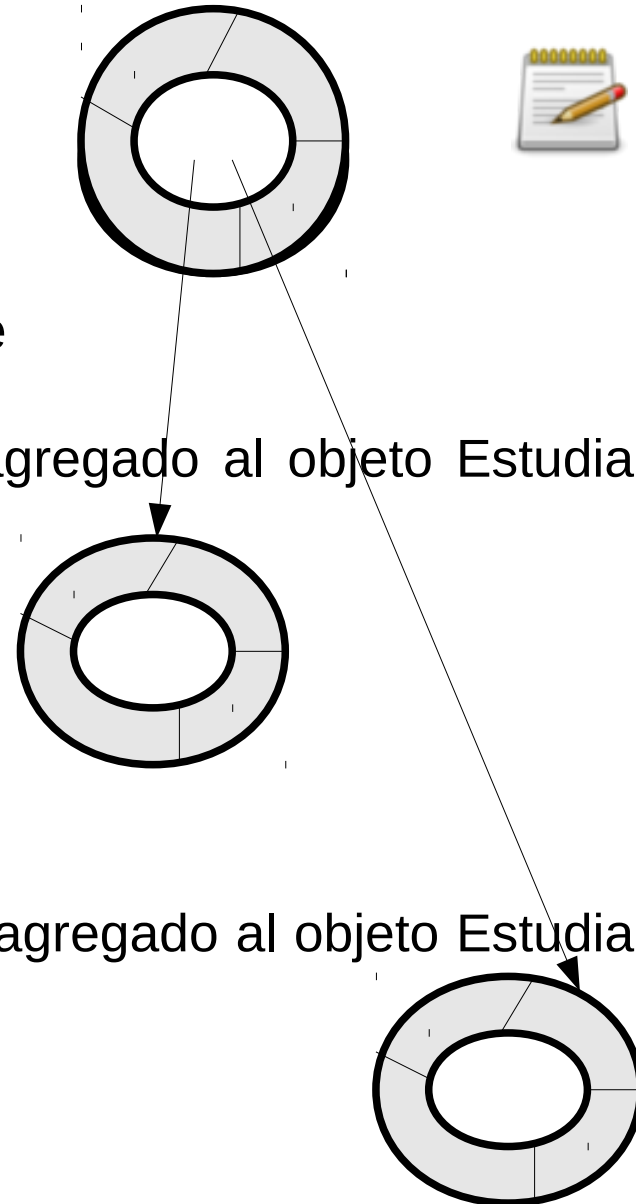
- Ejemplo de objeto Estudiante:  
nombre: "Juan"  
edad: 21  
direccion: objeto Direccion  
expediente: objeto Expediente

- Ejemplo de objeto Direccion agregado al objeto Estudiante anterior

calle: "Santa Tecla"  
numero: 99  
codigo\_postal: 18014  
Ciudad: "Granada"

- Ejemplo de objeto Expediente agregado al objeto Estudiante anterior

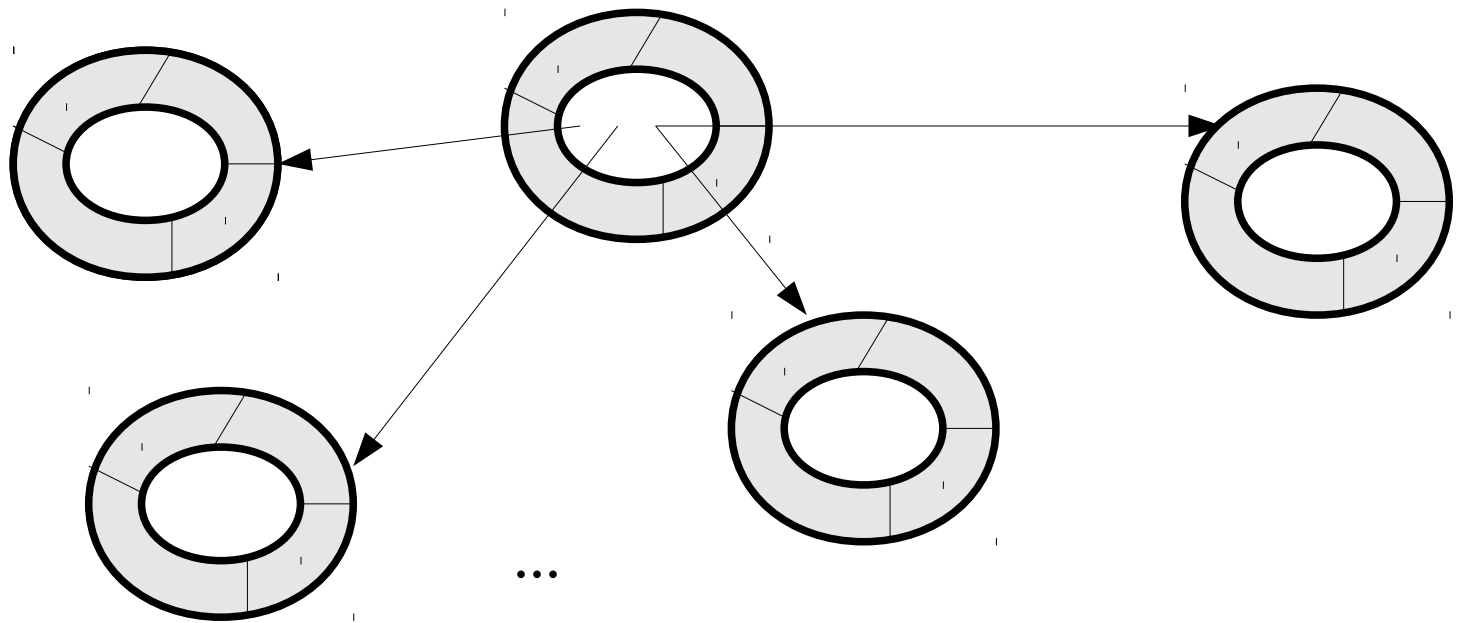
estudios: "Grado Informática"  
nota\_media: 6,3



Cuál es su estado?

# 13. Colecciones de objetos

- Cuando el estado de un objeto viene determinado por un conjunto de objetos iguales o parecidos, se dice que ese objeto es una **colección de objetos**.



# 13. Colecciones de Objetos

- **Tamaño**: fijas o variables.
- **Contenido**: homogéneas o heterogéneas.
- **Orden de elementos**: sin relación de orden (si acaso posición) o con orden.

Fijas y homogéneas -----> Eficientes

Variables y heterogéneas -----> Flexibles



# 13. Colecciones de objetos

- La **funcionalidad** general de una colección de objetos es:
  - Incluir uno o varios objetos.
  - Eliminar uno o varios objetos.
  - Comprobar la existencia de un determinado objeto.
  - Obtener un determinado elemento.
  - Obtener el número de elementos.
  - Iterar sobre todos sus elementos. Ejemplos:

## Iterador en Java para Listas:

```
ArrayList<MiClase> miLista = new ArrayList();  
for (MiClase elemento:miLista){ ....}
```

## Iterador en Ruby para Listas:

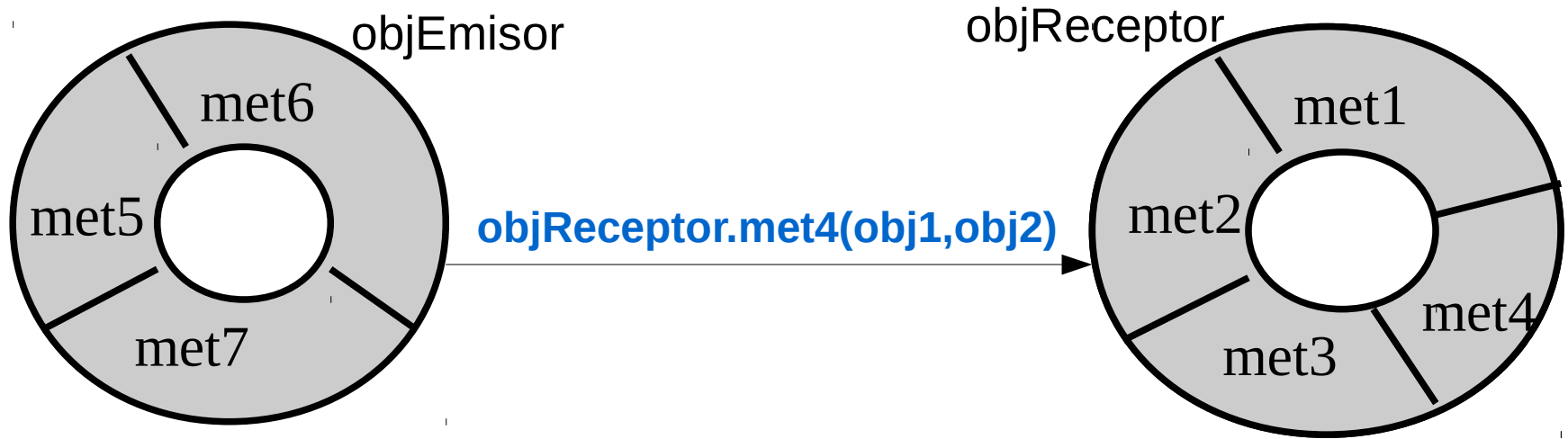
```
miLista = Array.new  
miLista.each {|elemento| ...}
```

# 14. PseudoVariables

- Una **pseudovariable** es una variable porque puede cambiar su valor u objeto que referencia, pero recibe el nombre de “pseudo” porque el programador no puede manipular ese valor (es decir, nunca podrá estar en la parte izquierda de una asignación: `this = a;`)
- En todos los lenguajes de programación existen **pseudovariables** que **referencian a un objeto especial: el objeto que tiene el control de la ejecución** en ese momento.
  - Se denominan de forma diferente en función del lenguaje de programación: Ejem: Java, c++: **this** y **super**; Ruby: **self** y **super**.
  - Cuando el objeto receptor toma el control, pasa a ser referenciado por esa variable. A lo largo de la ejecución, esa variable va cambiando de valor en función del objeto que tenga el control de la ejecución.

# 15. Envío de mensajes entre objetos

Mecanismo de comunicación entre objetos para realizar un funcionamiento.



objEmisor	objeto emisor del mensaje
objReceptor	objeto receptor del mensaje
met4()	operación requerida por objEmisor del objReceptor
obj1 y obj2	objetos necesarios para realizar la operación

# 15. Envío de mensajes entre objetos

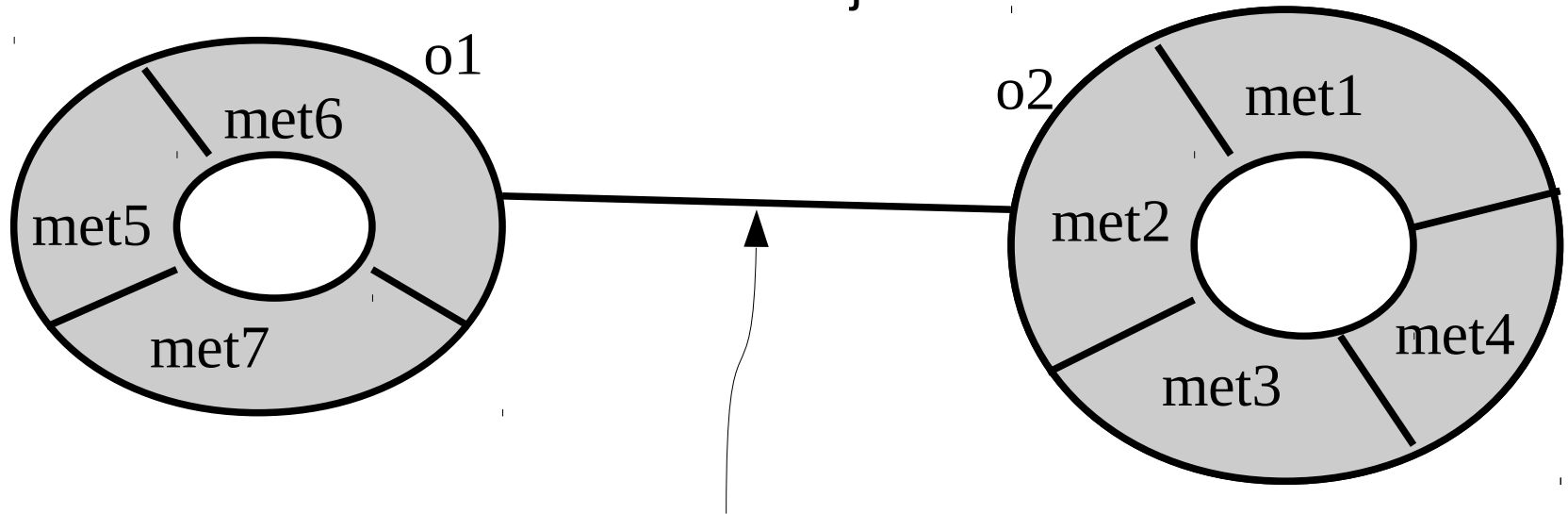
## Diferencias y semejanzas entre Java y Ruby en el envío de mensaje

`objReceptor.met4(obj1,obj2)`

	Java	Ruby
Receptor del mensaje	objReceptor	
Dónde está el código a ejecutar	En la clase a la que pertenece objReceptor en el momento del envío de mensaje en ejecución.	
Qué código se ejecuta	El que se corresponda con : <ul style="list-style-type: none"><li>- el nombre (met4),</li><li>- el número de parámetros (2) y</li><li>- los tipos de parámetros (tipo de obj1 y tipo de obj2)</li></ul> que intervienen en el envío de mensaje. Si en la clase no hay ningún método que se corresponda con el mensaje, se produce un error.	El que se corresponda con el nombre (met4).  Se produce error si en la clase no hay ningún método con ese nombre o existe pero con distintos parámetros definidos, a no ser que haya uno con más parámetros y éstos estén inicializados por defecto en la cabecera.
Cuándo se informa del error	En compilación y en ejecución.	En ejecución.

# 15. Envío de mensajes entre objetos

- Existen **canales de comunicación** entre un objeto *obj* y aquellos objetos que sean accesibles por él (ámbito y visibilidad adecuada).
- Cuando existe ese canal de comunicación, se dice que el objeto *obj* **conoce** esos objetos y puede comunicarse con ellos a través de envíos de mensaje.



Si ese canal no existiera o1 y o2 nunca podrían comunicarse a través de un envío de mensaje.

# 15. Envío de mensajes entre objetos

## Posibles canales de **comunicación/conocimiento** en Java:

```
public class Ejemplo {
```

```
    private static ClaseA variableGlobal=new ClaseA();
```

```
    private ClaseB variableAsociacion=new ClaseB();
```

```
    public void metodo(ClaseC variableParametro) {
```

```
        ClaseF variableLocal = new ClaseF();
```

```
        variableGlobal.operacion1();
```

```
        variableAsociacion.operacion2();
```

```
        variableParametro.operacion3();
```

```
        variableLocal.operacion4();
```

```
        this.operacion5();
```

```
    }
```

```
}
```

**Conocimiento global**

**Conocimiento asociación**

**Conocimiento parámetro**

**Conocimiento local**

**Conocimiento self**

¿Con qué objetos se comunica un objeto de la clase Ejemplo?

¿Cómo se inicia la comunicación?



# 15. Envío de mensajes entre objetos

Posibles canales de **comunicación/conocimiento** en Ruby:



```
$variableGlobal
```

```
class Ejemplo
```

```
  @@variableSemiGlobal
```

```
  def initialize(v)
```

```
    @variableAsociacion=v
```

```
  end
```

```
  def metodo(variableParametro)
```

```
    $variableGlobal.operacion0
```

```
    @@variableSemiGlobal.operacion1
```

```
    @variableAsociacion.operacion2
```

```
    variableParametro.operacion3
```

```
    variableLocal.operacion4
```

```
    self.operacion5
```

```
  end
```

```
end
```

**Conocimiento global**

**Conocimiento asociación**

**Conocimiento parámetro**

**Conocimiento local**

**Conocimiento self**

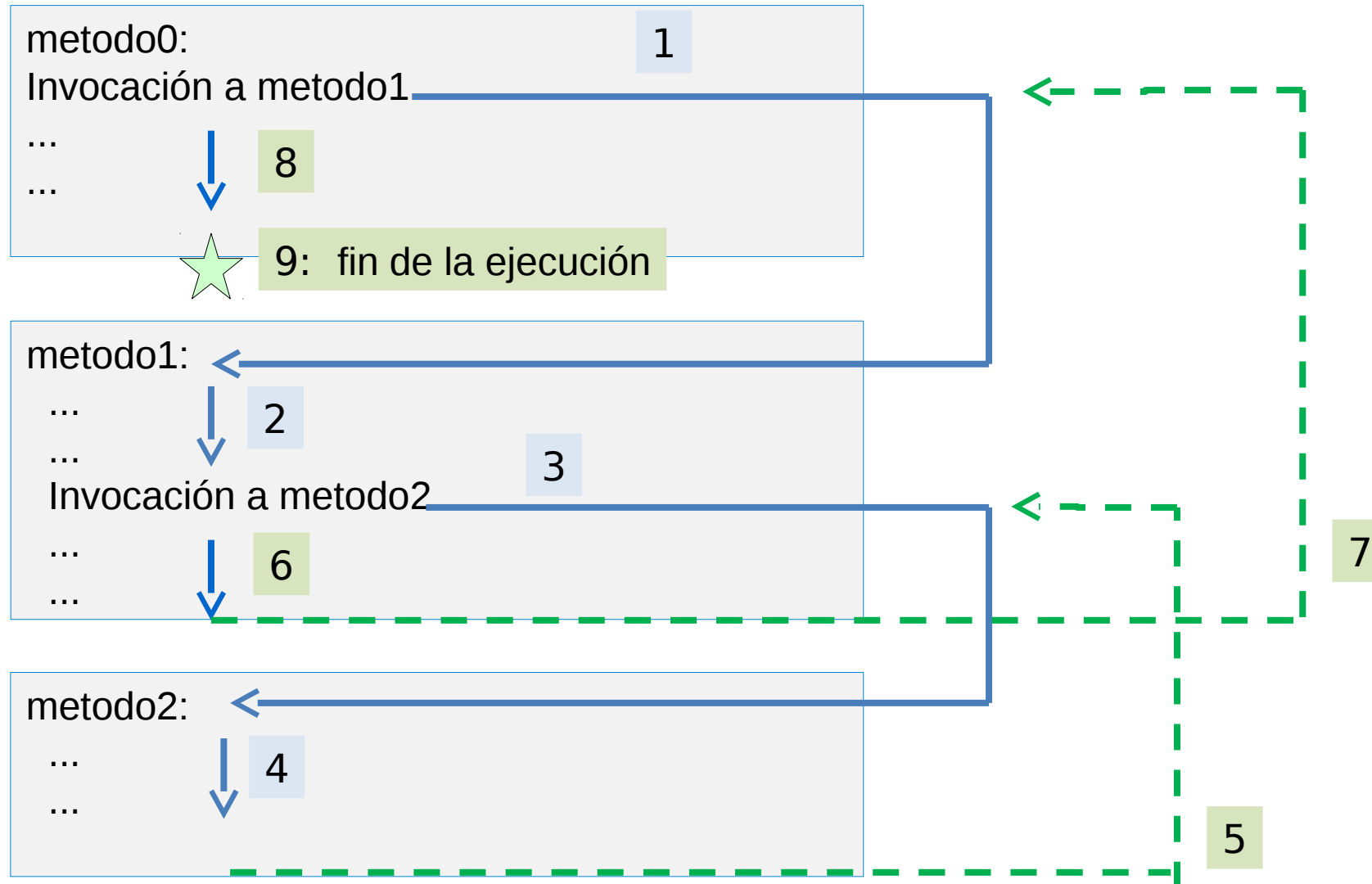
¿Con qué objetos se comunica un objeto de la clase Ejemplo?

¿Cómo se inicia la comunicación?



# 15. Envío de mensajes entre objetos

Flujo de control que se produce durante el envío de mensaje





# 15. Envío de mensajes entre objetos

Diferentes formas de ligar un mensaje al método que lo resuelve, según el momento en el que se realiza:

**Ligadura Estática:** Cuando ocurre antes de la ejecución.

**Ligadura Dinámica:** Cuando ocurre durante la ejecución.



Eficiencia

Flexibilidad

Tanto Java como Ruby utilizan la ligadura dinámica. C++ utiliza los dos tipos de ligadura, por defecto la estática. La dinámica sólo si se indica explícitamente.

# 16. Envío de mensajes a self en Ruby

main.rb

```

module M
  class A
    def metodo1
      puts self.inspect
    end

    def self.metodo2
      puts self.inspect
    end
  end

  puts self.inspect
  a = A.new
  a.metodo1
  A.metodo2
end

puts self.inspect

```

self en un método de instancia: objeto receptor del mensaje ligado a metodo1 (objeto actual)

self en un método de clase: clase actual

self en un módulo: módulo actual

#Resultado: M

#Resultado: #<M::A:0x2955ffe4>

#Resultado: M::A

self fuera de clases o módulos: archivo actual

#Resultado: main

