

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 los 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	Nº horas
2.1	Clases y Objetos: Conceptos básicos	6
2.2	Diagramas estructurales para la representación de clases	4
2.3	Diagramas de interacción entre objetos	4



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: paquetes y módulos**
- Diferenciar el uso de los métodos y variables según sus **especificadores de acceso**.
- Conocer las **pseudovariables**.

Objetivos de aprendizaje (cont.)



- Saber que hay métodos **consultores y modificadores** de objetos.
- Conocer el **ciclo de vida** de un objeto.
- Entender cómo se **construyen y destruyen** objetos.
- Diferenciar los **tipos de colecciones**.
- Aprender a **comparar objetos** según su identidad y estado.
- 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.
- Comprender el **tratamiento de excepciones** y aprender a manejarlas en Java y Ruby.

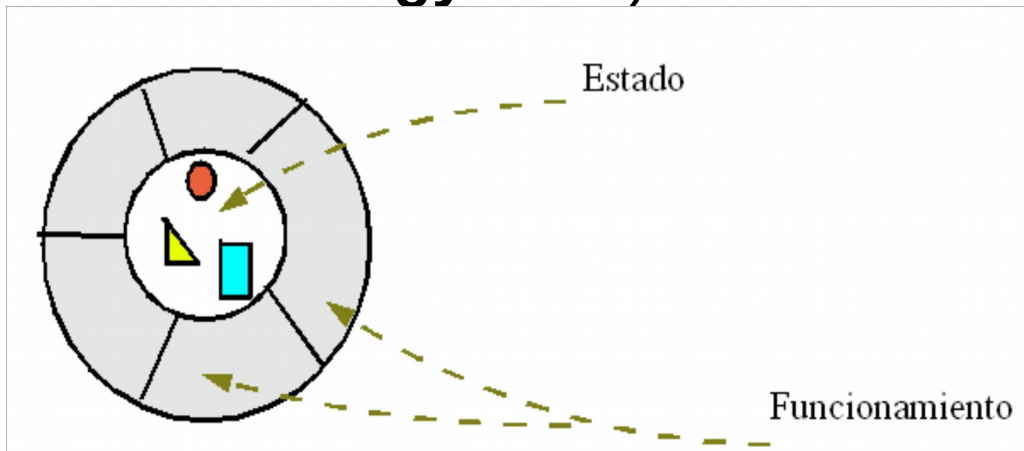
Contenidos



1. Concepto de objeto.
2. Estado e identidad de objetos.
3. Concepto de clase.
4. Atributos: tipos y ámbito.
5. Métodos: tipos y ámbito.
6. Ciclo de vida de un objeto.
7. Constructores.
8. Destrucción.
9. Consultores y modificadores.
10. Elementos de agrupación.
11. Especificadores de acceso.
12. Comparación de objetos: identidad y estado.
13. Agregaciones de objetos.
14. Colecciones.
15. Pseudovariantes.
16. Envío de mensajes entre objetos.
17. Excepciones.

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**).

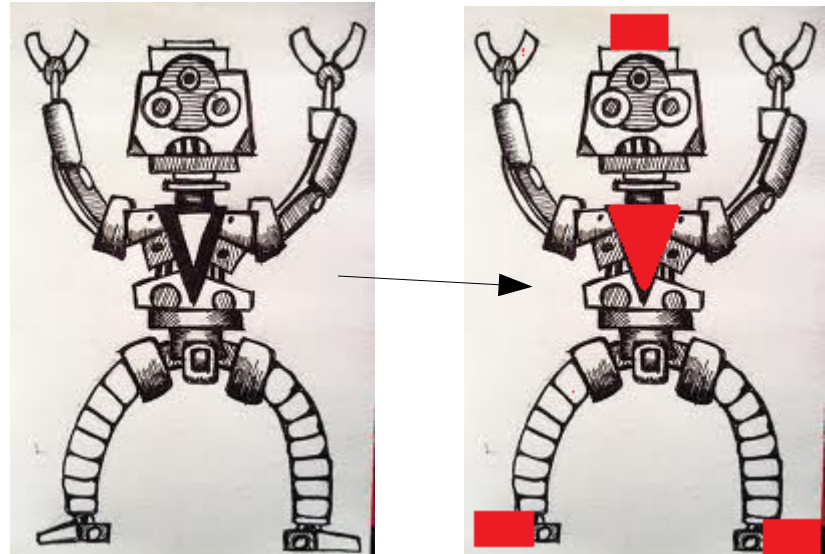


1. Concepto de objeto

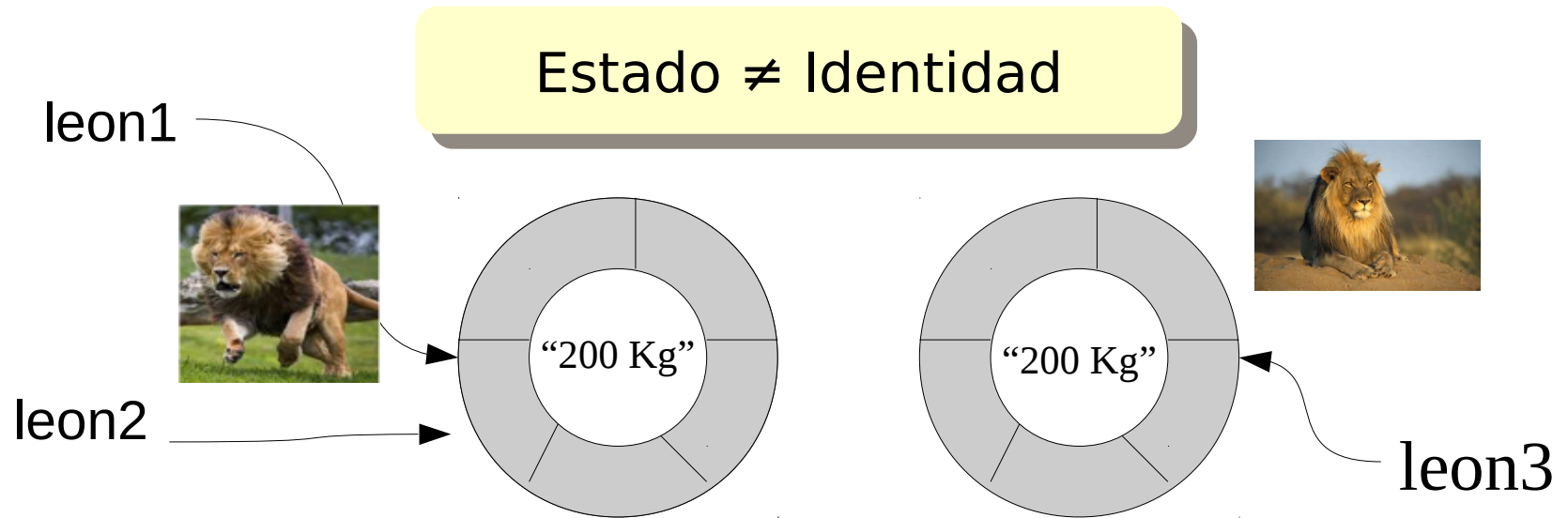
- La **identidad** es la propiedad que permite distinguir a un objeto de los demás.
 - Un objeto tiene identidad por el mero hecho de existir.
 - Aunque un objeto tenga características que lo pueden identificar de forma única, éstas no constituyen su identidad.
 - P.ej. el DNI de una persona lo identifica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

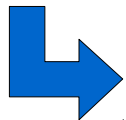
leon1 **es igual a** leon2 (cierto)
leon1 **es igual a** leon3 (cierto)
leon2 **es distinto de** leon3 (falso)

3. Concepto de clase

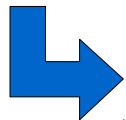
Definición de la RAE:

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. Atributos

- Un **atributo** es una característica que se representa mediante un valor almacenado en una variable.
- Existen dos tipos de atributos:
 - **Atributo de instancia**: Representa una característica de un objeto particular.
 - **Atributo de clase**: Representa una característica compartida por un conjunto de objetos y la propia clase.
- El **estado** de un objeto es definido por el conjunto de sus atributos de instancia.

4. Atributos en Java

- **Atributo de instancia:** Cada objeto tiene su propia copia de cada atributo de instancia. De ese modo, objetos diferentes pueden dar valores diferentes a un mismo atributo.

```
public class Bicicleta { // Se definen a continuación dos atributos de instancia.  
    private int marchas;  
    private int color;  
    ...  
}
```

- **Atributo de clase:** El atributo se almacena en la propia clase. De este modo, su valor es el mismo para todas las instancias de dicha clase.

```
public class Bicicleta { //Se define a continuación un atributo de clase.  
    private static int numeroDeBicicletas;  
}
```

4. Atributos en Ruby

- **Atributo de instancia:** Igual que en Java, cada objeto tiene su propia copia de los atributos de instancia definidos en la clase.

```
class Bicicleta # Se definen a continuación dos atributos de instancia.  
  # Los atributos de instancia se definen con @ dentro de cualquier método de instancia.  
  def initialize(numero_marchas, un_color)  
    @marchas = numero_marchas  
    @color = un_color  
  end  
  ...
```

- **Atributo de clase:** Igual que en Java, el atributo de clase es compartido por la clase y todas sus instancias.

```
class Bicicleta # Se define a continuación un atributo de clase.  
  # Los atributos de clase se definen con @@ fuera de cualquier método.  
  @@numero_de_bicicletas
```

4. Atributos en Ruby

- En Ruby, como las clases son objetos, existen también atributos de instancia de la clase.
- **Atributo de instancia de la clase:** El atributo define una característica de la clase (no de sus objetos), a la que solo la clase tiene acceso.

```
class Bicicleta # Se define a continuación un atributo de instancia de la clase.  
  # Los atributos de instancia de la clase se definen con @ en cualquier lugar fuera de los  
  # métodos de instancia.  
  @manual_ciclismo  
  ...
```


5. Métodos

- Un **método** es un trozo de código que define un comportamiento.
- Existen tres tipos de métodos:
 - **Método de instancia:** Dicho comportamiento es realizado por un objeto de la clase.

```
// Método de instancia en Java
public int getColor() {
    return color;
}
```

```
# Método de instancia en Ruby
def numero_serie
    @numero_serie
end
```

- **Método de clase:** El comportamiento es ejecutado por la clase y no requiere que exista ninguna instancia.

```
// Método de clase en Java
public static int getNumeroDeBicicletas() {
    return numeroDeBicicletas;
}
```

```
# Método de clase en Ruby
No tiene
```

- **Método de instancia de la clase:** Dicho comportamiento es realizado por el objeto clase.

```
# Método de instancia de la clase en Java
No tiene ¿por qué?
```

```
# Método de instancia de la clase en Ruby
def self.set_manual_ciclismo(manual)
    @manual_ciclismo = manual
end
```

5. Ámbito de atributos y métodos

El ámbito de un atributo es el contexto donde puede usarse y dependerá de su tipo y el lenguaje de programación.

La siguiente tabla indica los tipos de métodos desde dónde es accesible cada tipo de atributo en Java y Ruby:

TIPO DE ATRIBUTO	MÉTODOS donde puede usarse	
	Java	Ruby
De instancia	De instancia	De instancia
De instancia de la clase	No existe este tipo de atributo	De instancia de la clase
De clase	De instancia y de clase	De instancia y de instancia de la clase

5. Ámbito de atributos y métodos



Un ejemplo:

Variables de
instancia

Variable
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 atributos y métodos

Variable de clase

Variable de instancia de la clase

Variables de instancia

Métodos de instancia



Probar código de ejemplo de Bicicleta en Ruby y Java

```
class Bicicleta
  @@numero_de_bicicletas = 0
  @manual_ciclismo

  def initialize(numero_marchas, un_color)
    @marchas = numero_marchas
    @color = un_color
    @numero_serie = @@numero_de_bicicletas
    Bicicleta.incrementar_numero_de_bicicletas
  end

  def color
    @color
  end

  def numero_serie
    @numero_serie
  end

  def self.numero_de_bicicletas
    @@numero_de_bicicletas
  end

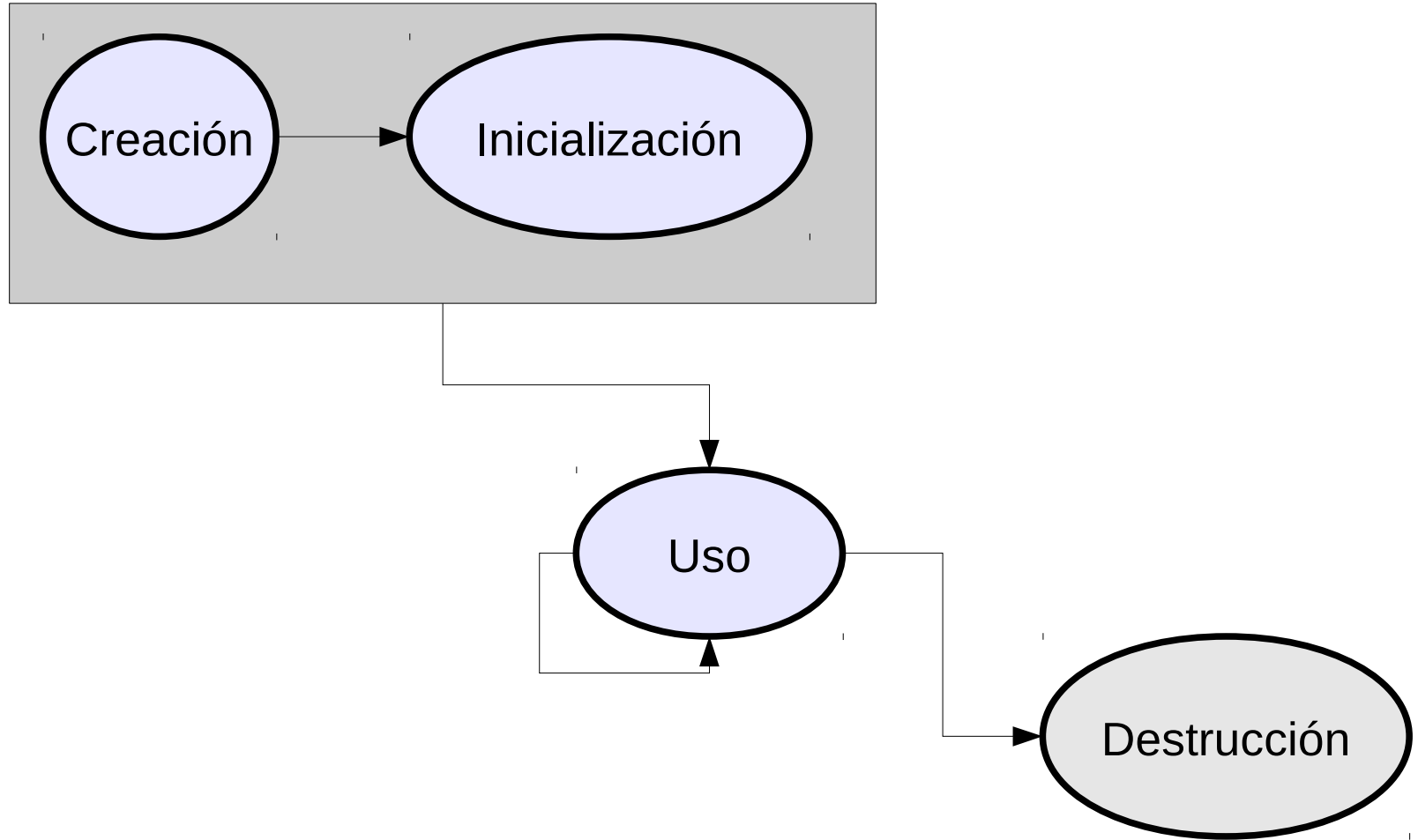
  def self.incrementar_numero_de_bicicletas
    @@numero_de_bicicletas = @@numero_de_bicicletas+1
  end

  def self.set_manual_ciclismo(manual)
    @manual_ciclismo = manual
  end
end
```



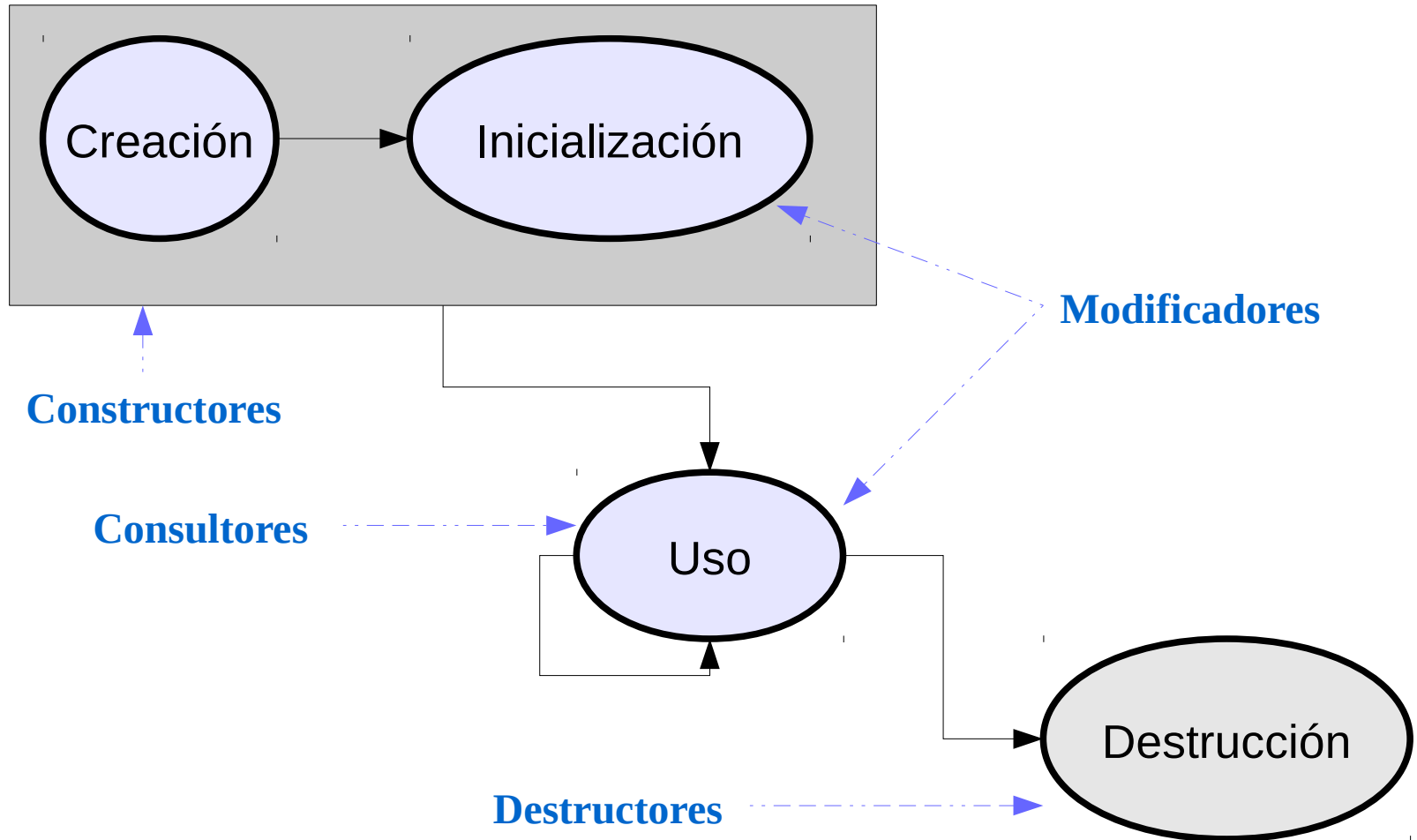
Métodos de instancia de la clase

6.Ciclo de vida de un objeto



6.Ciclo de vida de un objeto

Tipos de métodos que intervienen:



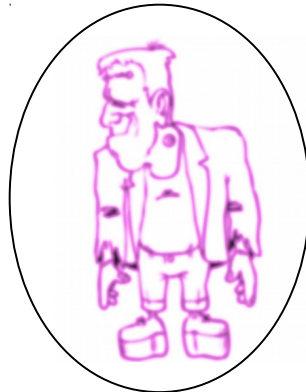
7. Constructores

- 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!).

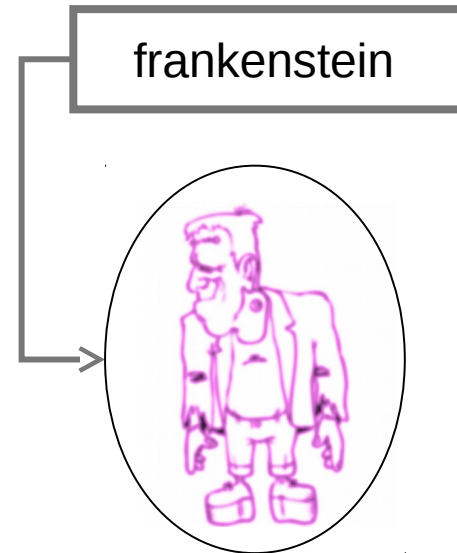
7. Constructores

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:

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

Ruby: frankenstein = MuertoViviente.new(2.0)

Invocación con new

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
```

Invocación en Ruby:

```
zombi_1 = MuertoViviente.new(2.0)
zombi_2 = MuertoViviente.new
zombi_3 = MuertoViviente.new(2.5, 8)
zombi_4 = MuertoViviente.crearZombi(3.3)
```

otros métodos de clase:

```
class MuertoViviente
  attr_accessor :dedos_de_frente

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

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



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"?
¿Que repercusión tendría declarar la clase MuertoViviente según una forma u otra de las indicadas en los recuadros anteriores?

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** puede dar lugar a **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	<i>delete frankenstein</i>
	Objective C	<i>frankenstein release</i>

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 de 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 de 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.



Modifica el ejemplo de las transparencias 19 y 20 (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:
 - Para cada lenguaje existen agrupaciones estándar ya definidas con las clases, variables o métodos más utilizados.
 - Con la misma idea, es posible hacer nuevas agrupaciones con las implementaciones propias.
- 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**.
- Pueden agrupar otros paquetes.
- **import** permite acceder de forma directa a clases de otro paquete.

```
package animales;
```

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

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

```
-----  
package animales.reptiles;
```

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

```
-----  
import animales.Gallina;
```

```
package granja;
```

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

10. Elementos de agrupación



Módulos de Ruby:

- Permiten agrupar **clases, métodos, constantes 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

Uso de módulos de Ruby:

- **require:** para indicar la ruta del **fichero** donde está el código que necesitamos cargar.
- **require_relative:** si el **fichero** a cargar está en la carpeta actual.
- **require** o **require_relative** suelen ponerse solo en el módulo de la clase “main”.
- **include** permite importar un **módulo** dentro de una clase.

```
module cronometro  
  def empezar...end  
  def parar ... end  
end
```



crono.rb

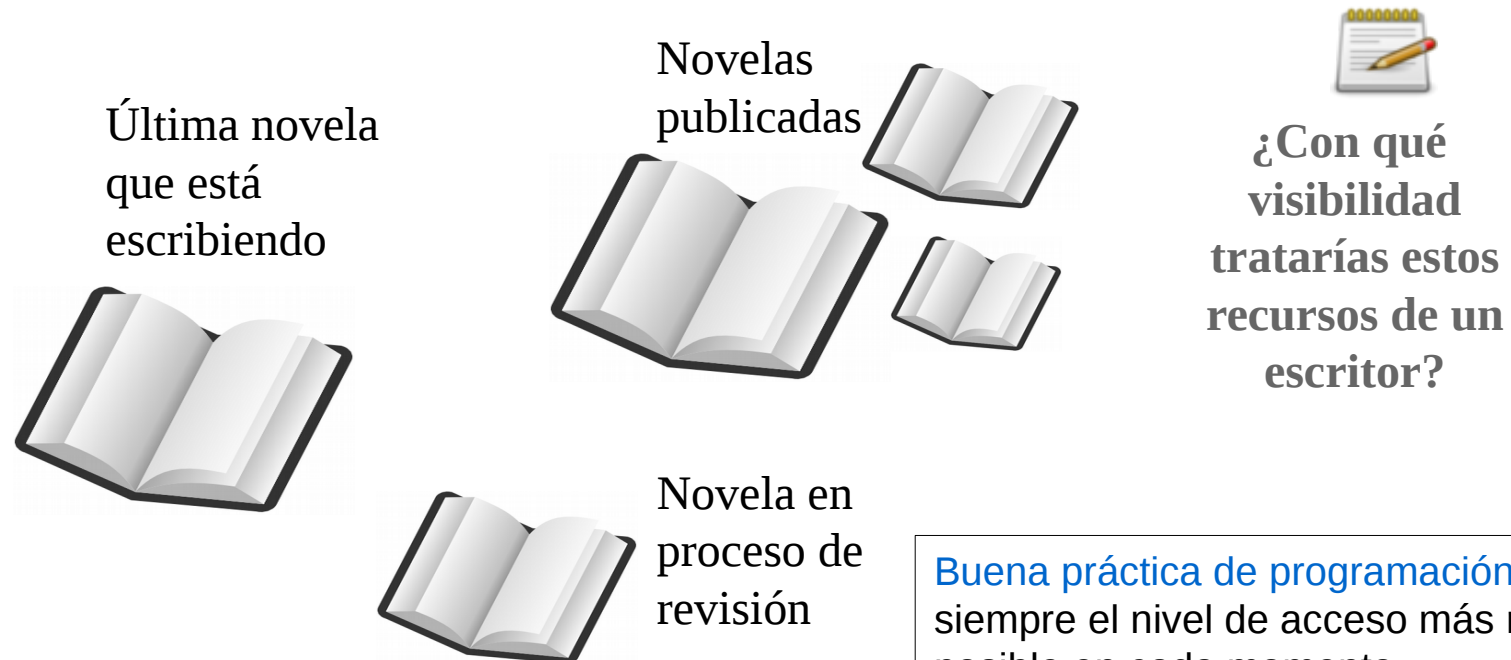
```
-----  
require_relative "crono"  
class Reloj  
  include cronometro  
  def mostrarHora...end  
end  
rel=Reloj.new  
rel.mostrarHora  
rel.empezar
```

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.



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
	Esa clase	Otra clase	Otra clase
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	Desde la propia clase	Desde otra clase
private	✓		
protected	✓	✓	
public	✓	✓	✓

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



<http://www.skorks.com/2010/04/ruby-access-control-are-private-and-protected-Methods-only-a-guideline/>

11. Especificadores de acceso

```
class MuertoViviente
  attr_reader :dedos_de_frente
  def initialize(unFloat)
    @dedos_de_frente=unFloat
    @edad=30
  end
  def asustar
    'uuuuuuuuuuuuuu'
  end
  def presentar
    "tengo " + self.dedos_de_frente+
    " dedos y " + miEdad + " lustros" +
    + asustar
  protected
  def comer
    'rico rico'
  end
  def miEdad
    return 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. Comparación de objetos: identidad y estado

- A la hora de comparar objetos podemos comparar identidad o estado.
- Dependiendo de cómo se hayan construido los objetos a comparar, así será el resultado de la comparación:

	Comparando identidad ¿objetos idénticos?	Comparando estado ¿objetos iguales?
Cuando a es idéntico a b	true	true
Cuando a es copia de b	false	true
Cuando a no es ni idéntico ni copia de b	false	true o false

- Todos los lenguajes de programación proporcionan funcionalidad para comparar identidad e igualdad (estado) de objetos.

12. Comparación de objetos: identidad y estado



obj1 == obj2, obj1 != obj2 y obj1.equals(obj2)

- Comparan identidad por defecto y devuelven `true` o `false`
- Para comparar estado se redefine `equals(obj)`
- `obj1 != obj2` es equivalente a `!(obj1 == obj2)`

Ejemplo

```
class MiClase{
    private String saludo;
}
MiClase mc1 = new MiClase("hola");
MiClase mc2 = new MiClase("hola");
MiClase mc3 = mc1;

mc1 == mc2 // false
mc1 == mc3 // true
mc2 == mc3 // false
mc1.equals(mc2) // false
mc1.equals(mc3) // true
mc2.equals(mc3) // false
```



Entender y manipular el
código en ComparacionJava

12. Comparación de objetos: identidad y estado



Código recomendado para redefinir equals(obj)

@Override

```
public boolean equals(Object obj) {
```

```
    if (obj == null)
        return false;
```

```
    if (obj == this)
        return true;
```

```
    if (!(obj.getClass().getSimpleName().equals("MiClase")))
        return false;
```

```
    MiClase mc = (MiClase) obj;
    if (!saludo.equals(mc.saludo))
        return false;
```

```
    // compara según el atributo saludo de obj
    return true;
```

```
}
```

Cabecera ya proporcionada y que no podemos cambiar (@Override)

Consulta el nombre de la clase a la que pertenece obj

Estudiaremos con detalle la redefinición en el tema 3

Referenciar obj por una variable, mc, que sea de tipo MiClase

12. Comparación de objetos: identidad y estado

obj1 == obj2, obj1 != obj2, obj1.equal?(obj2) y obj1.eql?(obj2)



- Comparan identidad por defecto y devuelven true o false
- *obj1 != obj2* equivalente a *!(obj1 == obj2)*
- Recomendaciones para la redefinición de estos métodos:
 - Para comparar estado redefinir *obj1 == obj2*
 - No redefinir *equal?(obj2)* ya que se usa internamente para determinar identidad
 - Mantener el mismo significado de *==* y *eql?(obj2)*: para Hash compara keys

```
class MiClase
  attr_reader :saludo
end

mc1 = MiClase.new("hola")
mc2 = MiClase.new("hola")
mc3 = mc1
```

```
mc1 == mc2 # false
mc1 == mc3 # true
mc2 == mc3 # false

mc1.equal?(mc2) # false
mc1.equal?(mc3) # true
mc2.equal?(mc3) # false
```

Entender y manipular el
código en `comparacion_ruby`



[Lectura interesante: comparando objetos en Ruby](#)

12. Comparación de objetos: identidad y estado

Código recomendado para redefinir ==(obj)



```
def ==(obj)

  if (obj == nil)
    return false
  end

  if obj.class.name.split('::').last != 'MiClase'
    return false
  end

  if @saludo != obj.saludo
    return false
  end

  # compara según el atributo saludo de obj
  return true

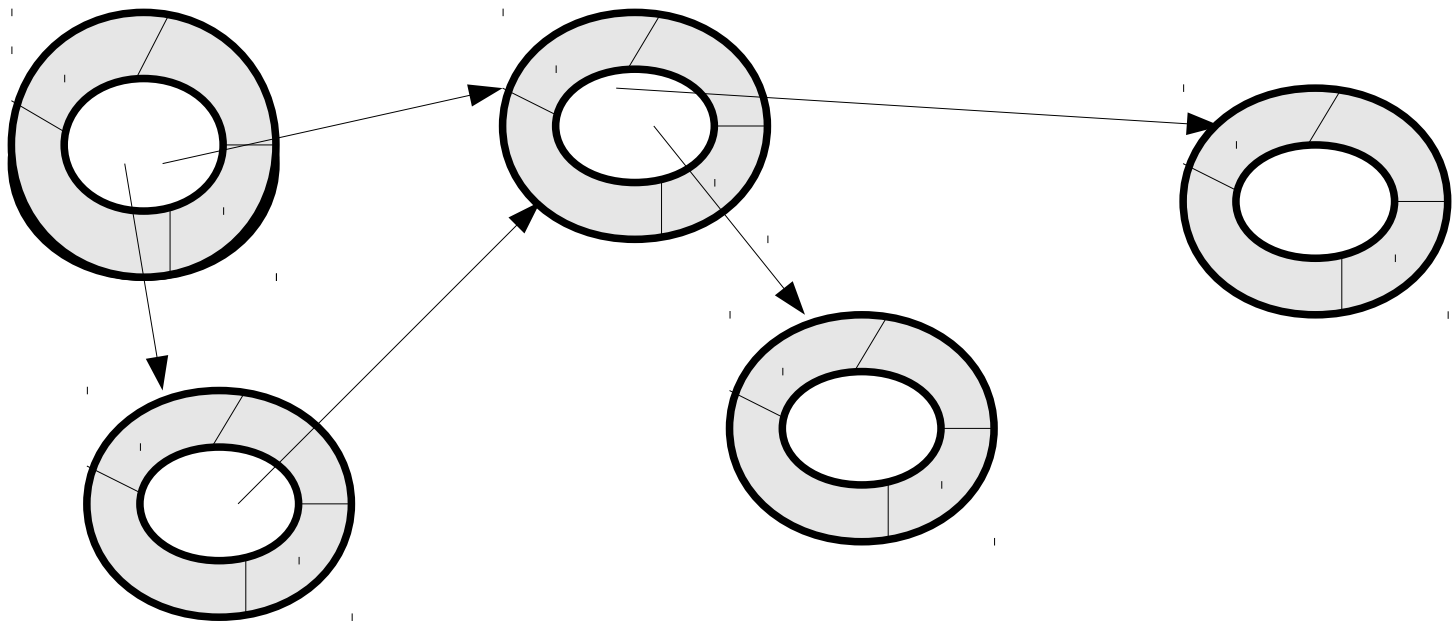
end
```

Consulta el nombre
de la clase a la que
pertenece obj

Estudiaremos con detalle la
redefinición en el tema 3

13. Agregación de objetos

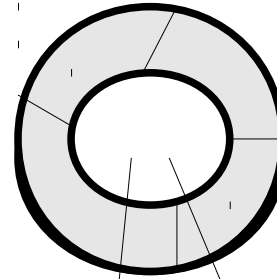
- El estado de un objeto puede estar formado por un conjunto de objetos y éstos a su vez estar compuestos por otros objetos y así sucesivamente, hasta llegar a objetos cuyo estado está formado por valores simples, formando un grafo dirigido de relaciones entre objetos.



Probar código de ejemplo del ClubCiclista en Java y Ruby

13. Agregación de Objetos

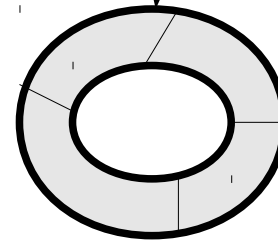
- Ejemplo de objeto Estudiante:
nombre: "Juan"
edad: 21
direccion: objeto Direccion
expediente: objeto Expediente



¿Cuál es el estado del objeto Estudiante?

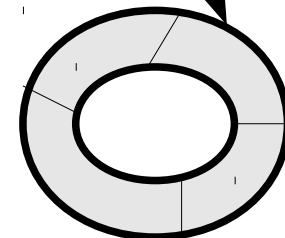
- Ejemplo de objeto Direccion agregado al objeto Estudiante anterior

calle: "Santa Tecla"
numero: 99
codigo_postal: 18014
Ciudad: "Granada"



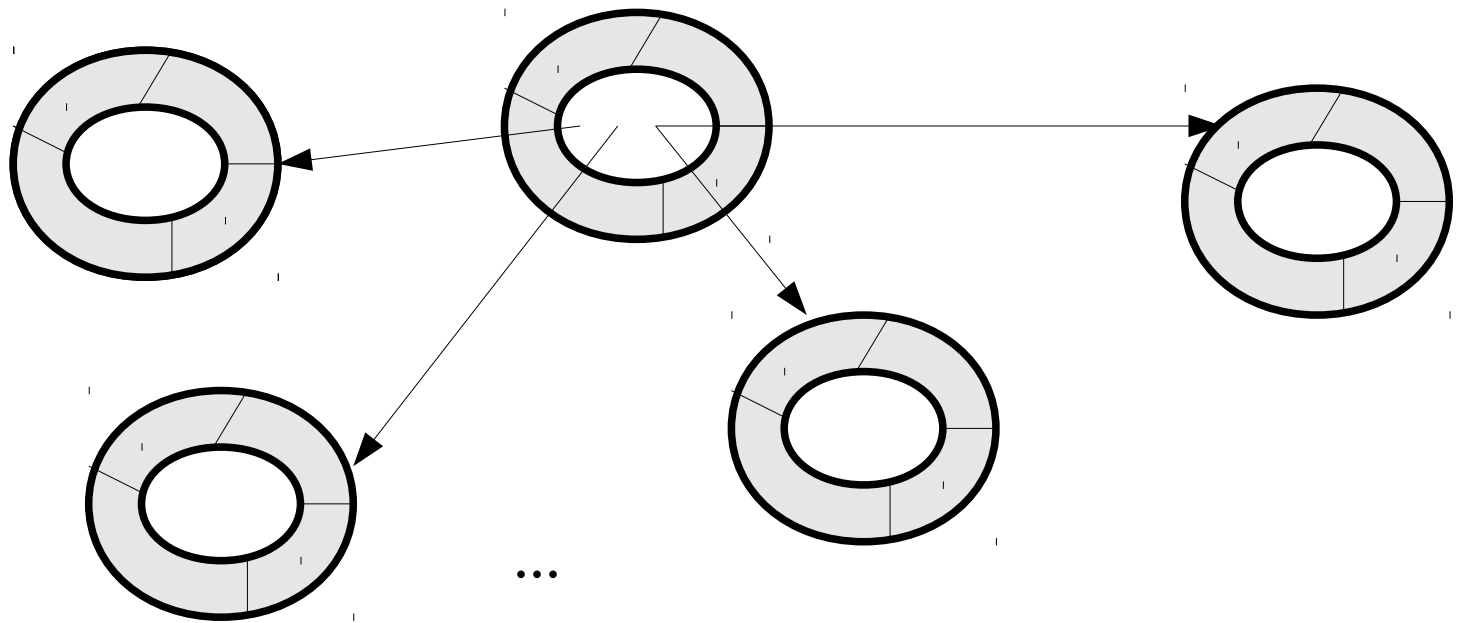
- Ejemplo de objeto Expediente agregado al objeto Estudiante anterior

estudios: "Grado Informática"
nota_media: 6,3



14. 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**.



14. Colecciones de Objetos

	Listas	Conjuntos	Diccionarios
Propiedades	Sin orden Con duplicados	Sin duplicados	Cada elementos es un par: (key, value) Sin duplicados en su Key
Java	Clases: <ul style="list-style-type: none">• LinkedList• ArrayList	Clases: <ul style="list-style-type: none">• HashSet (sin orden)• TreeSet (ordenados según la relación de orden definida entre sus elementos)	Clases: <ul style="list-style-type: none">• HashMap (sin orden)• TreeMap (ordenado por la key y según la relación de orden definida en la clase a la que pertenece key)
Ruby	Clases: <ul style="list-style-type: none">• Array	Clases: <ul style="list-style-type: none">• Set (sin orden)	Clases: <ul style="list-style-type: none">• Hash (sin orden)

- **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

14. 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){ ....}
```

Iteradores en Ruby para Listas:

```
miLista = Array.new  
miLista.each {|elemento| ...}  
for elemento in miLista  
  ...  
end
```

15. PseudoVariables

- Una **pseudovariable** es una variable porque puede cambiar su valor o el objeto al 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.

16. 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

16. Envío de mensajes entre objetos

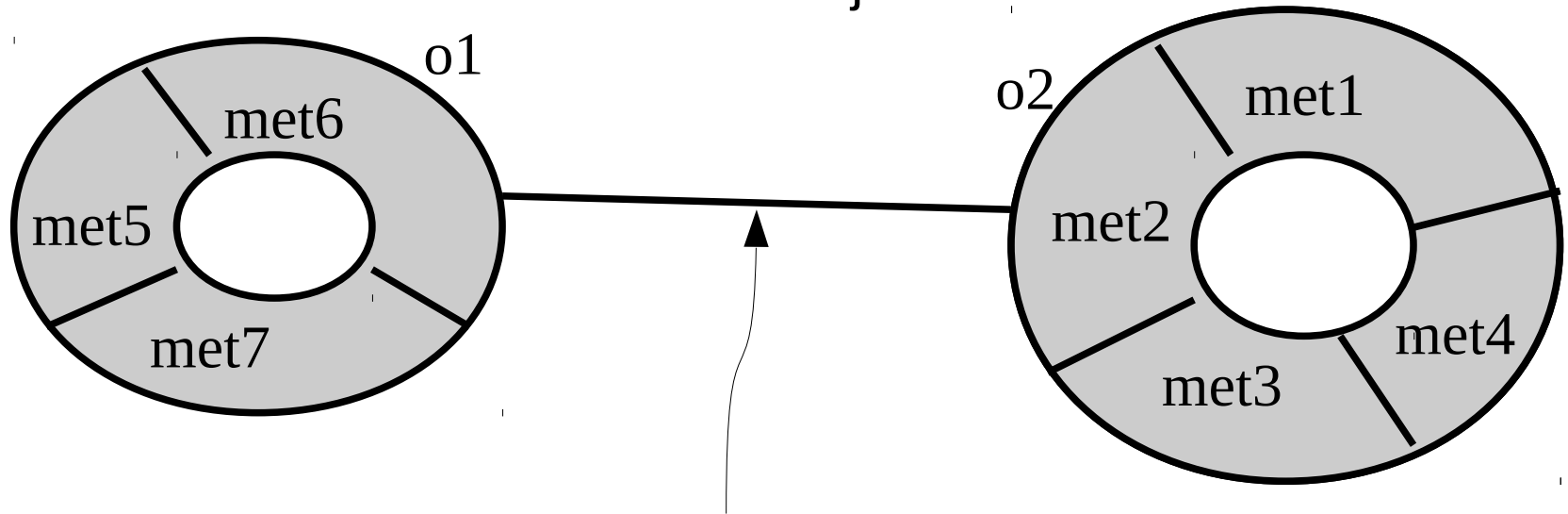
Diferencias y semejanzas entre Java y Ruby en el envío de mensaje

`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">- el nombre (met4),- el número de parámetros (2) y- los tipos de parámetros (tipo de obj1 y tipo de obj2) 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). Si en la clase no hay ningún método con ese nombre o existe pero con distintos parámetros definidos, se produce un error.
Cuándo se informa del error	En compilación y en ejecución	En ejecución

16. 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.

16. Envío de mensajes entre objetos

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

```
public class Ejemplo {
```

```
    private static ClaseA variableGlobal;
```

```
    private ClaseB variableAsociacion;
```

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

```
        ClaseF variableLocal;
```

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

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

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

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

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

```
    }
```

```
}
```

Conocimiento global

Conocimiento asociación

Conocimiento local

Conocimiento parámetro

Conocimiento self

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

¿Cómo se inicia la comunicación?



16. 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

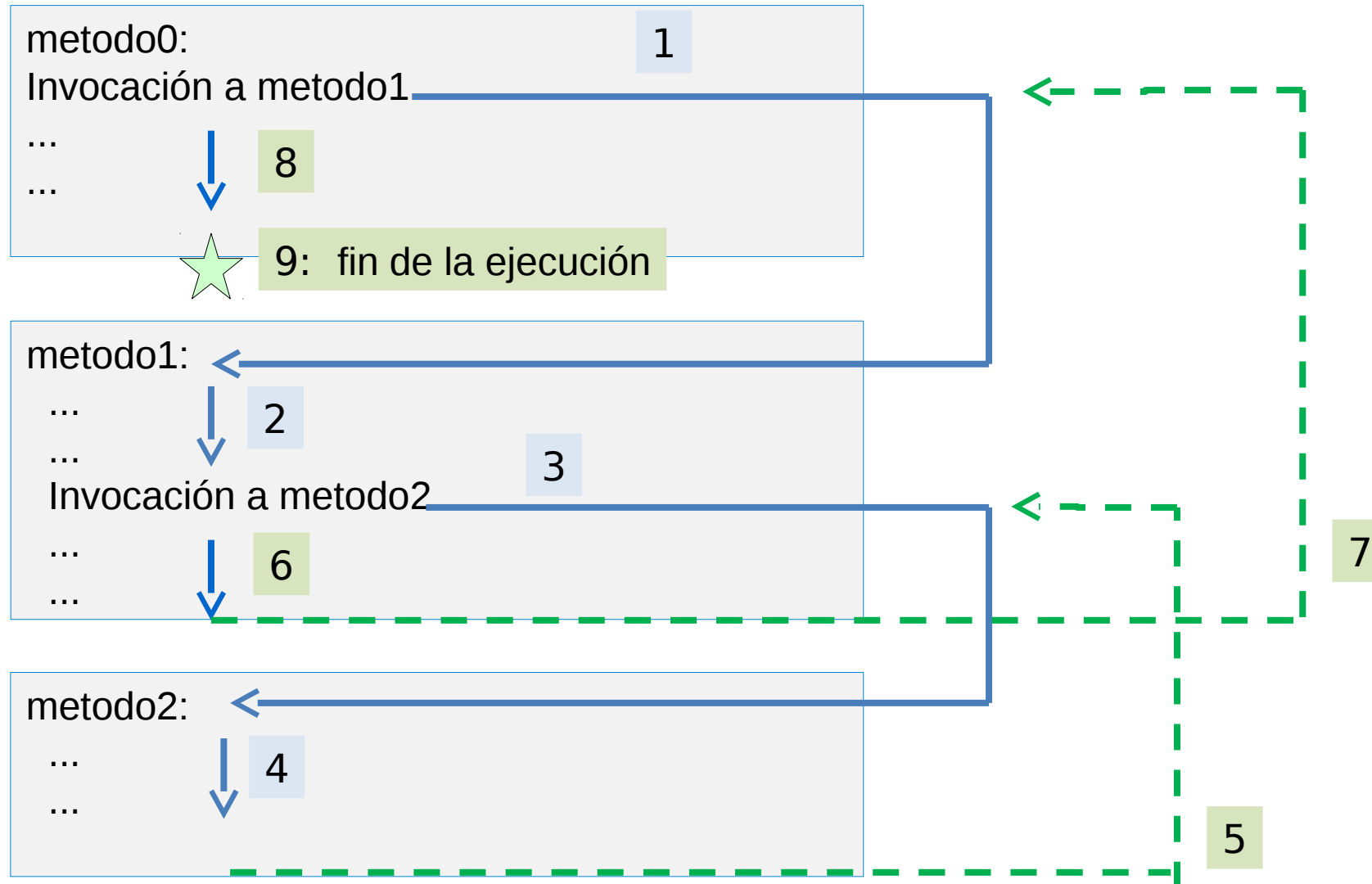
¿Con qué objetos se comunica un objeto de la clase Ejemplo?

¿Cómo se inicia la comunicación?



16. Envío de mensajes entre objetos

Flujo de control que se produce durante el envío de mensaje



16. Envío de mensajes entre objetos

Existen 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

self fuera de clases o módulos: archivo actual

#Resultado: M

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

#Resultado: M::A

#Resultado: main



17. Excepciones

Excepción:

Situación provocada por un problema durante la ejecución que implica que el programa termine de forma anormal, a no ser que se maneje la excepción.

Ejemplos:

- El usuario introduce un valor erróneo en una variable.
- Se necesita un fichero que no existe.
- Un objeto no puede responder un mensaje en su estado actual.

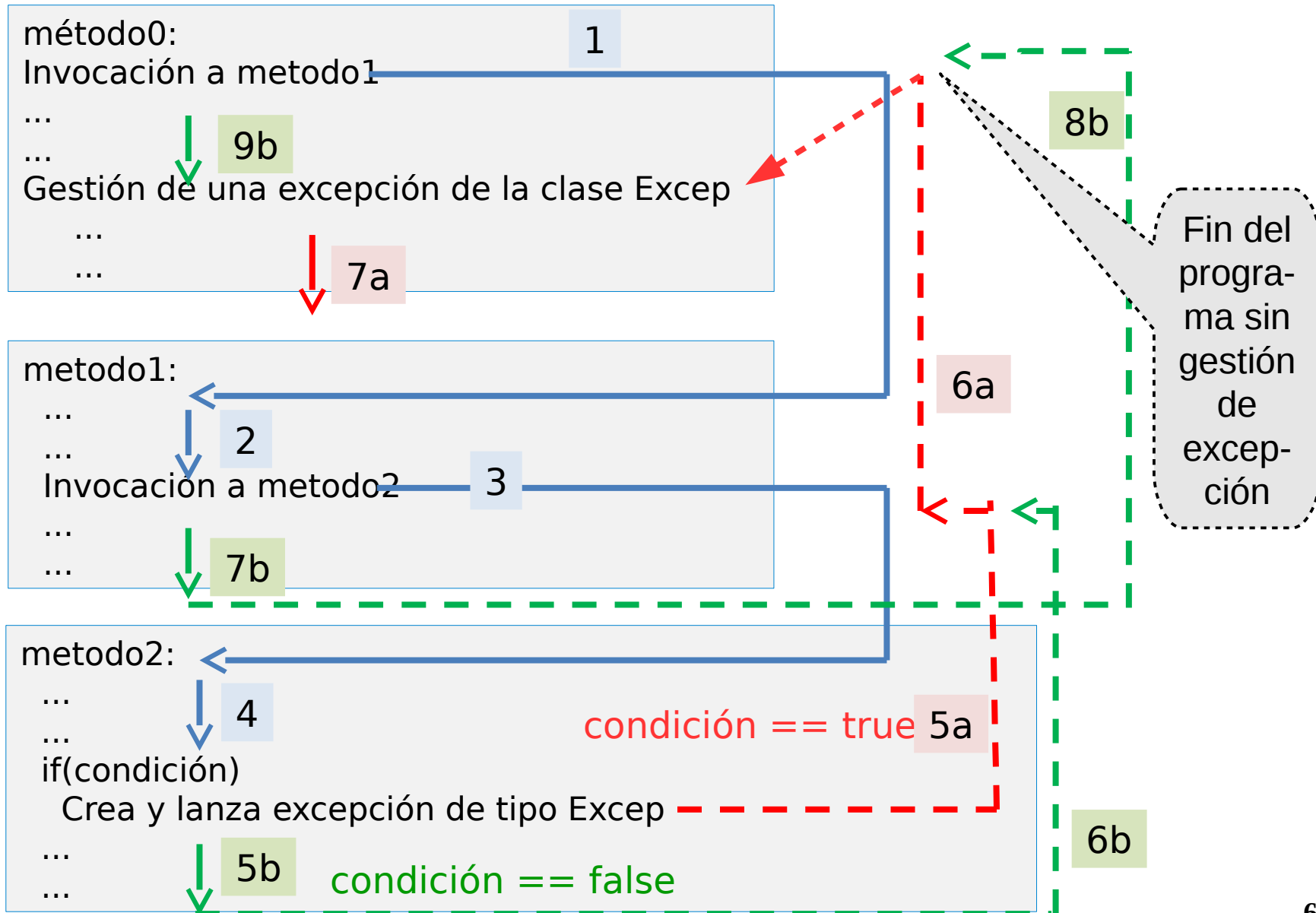
Tratamiento/manejo de la excepción:

Evita la terminación anormal, detectando el problema y dando una terminación alternativa para recuperarse del error.

Ejemplos:

- Se muestra de forma comprensible un mensaje de error.
- Se cambia el valor de la variable o el estado del objeto.


17. Excepciones: tratamiento



17. Excepciones:

Mecanismos en los lenguajes OO

- Los lenguajes de programación OO proveen de mecanismos para:
 - La definición e instanciación de clases/tipos de Excepciones.
 - El lanzamiento de excepciones, cuando se producen.
 - La captura y tratamiento de excepciones.
 - En algunos casos: avisar en la cabecera que un método propaga determinado tipo de excepciones.



Ejemplo de excepciones en los proyectos:
ExcepcionesJava y excepciones_ruby

17. Excepciones: Alternativas en Java

```
try{ ...  
    a.metodo();  
    ...  
} catch (UnTipoDeExcepcion e) {  
    // Código a ejecutar cuando se produzca una excepción de tipo  
    // UnTipoDeExcepcion  
    dentro del código try  
} catch (OtroTipoDeExcepcion o){  
    // Código a ejecutar cuando se produzca una excepción de tipo  
    // OtroTipoDeExcepcion dentro del código try  
} finally {  
    // Código que se ejecuta independientemente de que se lance o no una  
    excepción dentro del código try  
}
```



```
void metodo() throws UnTipoDeExcepcion, OtroTipoDeExcepcion {  
    ...  
    if (algopasa)  
        throw new UnTipoDeExcepcion("mensaje_error1");  
    ...  
    if (otracosapasa)  
        throw new OtroTipoDeExcepcion("mensaje_error2");  
    ...  
}
```

17. Excepciones: Alternativas en Ruby

```
begin
```

```
  ...
```

```
  a.metodo
```

```
  ...
```

```
rescue UnTipoDeExcepcion => e
```

```
  # Código a ejecutar cuando se produzca una excepción de tipo
```

```
  UnTipoDeExcepcion dentro del begin anterior
```

```
rescue OtroTipoDeExcepcion => o
```

```
  # Código a ejecutar cuando se produzca una excepción de tipo
```

```
  OtroTipoDeExcepcion dentro del begin anterior
```

```
else
```

```
  # Código que se ejecuta si no hay errores dentro del begin
```

```
ensure
```

```
  # Código que se ejecuta siempre
```

```
end
```



```
def metodo
```

```
  ...
```

```
  if (algopasa)
```

```
    raise UnTipoDeExcepcion, 'mensaje_error1'
```

```
  end
```

```
  if (otracosapasa)
```

```
    raise OtroTipoDeExcepcion, 'mensaje_error2'
```

```
  end
```

```
end
```

17. Excepciones: Tipos

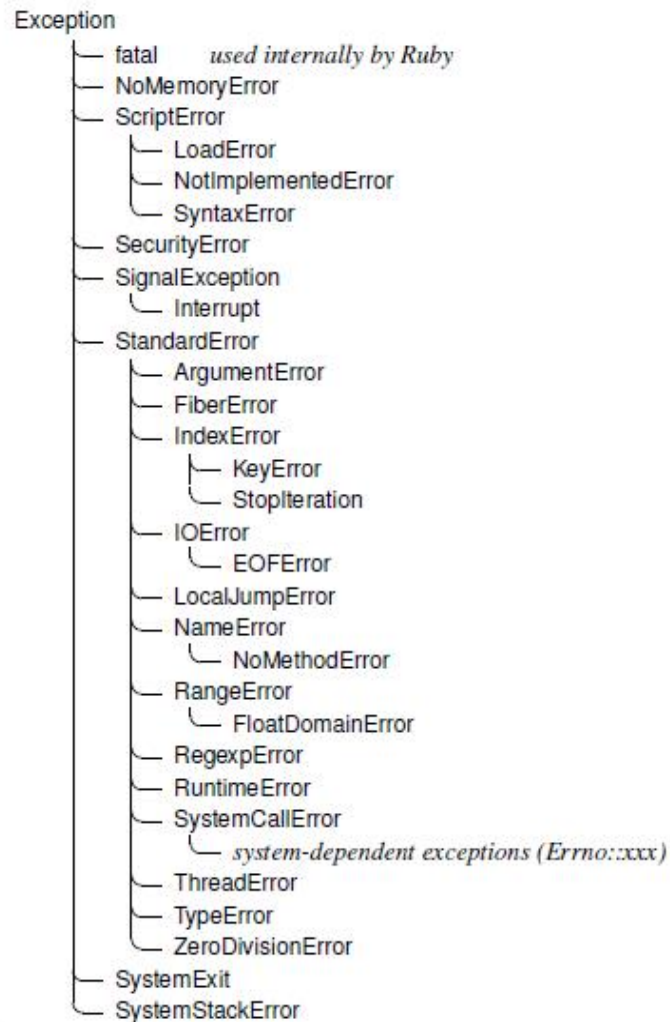
Además, los lenguajes de programación proveen clases correspondientes a los tipos de excepciones más comunes.

- Ejemplos de algunas clases de excepciones de Java:
 - `ArrayStoreException`
 - `NullPointerException`
 - `ArrayIndexOutOfBoundsException`
 - `NumberFormatException`
 - `ClassNotFoundException`

17. Excepciones: Tipos

Ejemplos de algunas clases de excepciones de Ruby:

- RuntimeError
- NoMethodError
- ArgumentError
- IOError



17. Excepciones: ejemplos Java



Código con excepción **sin** tratamiento definido:

```
int divisor = 0;
int resultado = 100/divisor;
System.out.println("Resultado: " + resultado);
/* Salida: Exception in thread "main"
    java.lang.ArithmeticException: / by zero */
```

Se detecta el error y se termina la ejecución con un mensaje por defecto

Código con excepción **con** tratamiento definido:

```
try {
    int divisor = 0;
    int resultado = 100/divisor;
    System.out.println("Resultado: " + resultado);
} catch (ArithmeticException ex) {
    System.out.println("Error, dividiendo por cero");
    // Salida: Error, dividiendo por cero
    System.out.print(ex.getMessage());
    // Salida: / by zero
}
```

Se detecta el error y se pasa a su tratamiento

17. Excepciones: Ejemplos en Ruby

Código con error **sin** tratamiento definido:

```
divisor = 0
resultado = 100/divisor
puts "Resultado: " + resultado
# Salida: ZeroDivisionError: divided by 0
```

Se produce el error y se termina la ejecución con un mensaje de error por defecto

Código con error **con** tratamiento definido:

```
begin
  divisor = 0
  resultado = 100/divisor
  puts "Resultado:" + resultado
rescue ZeroDivisionError => ex
  ERR.puts "Error, dividiendo por cero"
end

# Salida: Error, dividiendo por cero

puts ex.message
# Salida: divided by 0
```

Se detecta el error y se pasa a su tratamiento, al estar éste definido



Pruebas



Para afianzar y comprender mejor los conceptos aprendidos en esta lección haz pruebas con los siguientes ejemplos en Java y Ruby:

- Ejercicios básicos de Ruby: `basico_ruby`
- Ejemplos de uso de métodos y variables y pruebas de visibilidad: `LaBicicletaJava` y `la_bicicleta_ruby`
- Ejemplos de agregación de objetos: `ElClubCiclista_Java` y `el_club_ciclista_ruby`
- Ejemplos de colecciones: `ColeccionesJava` y `colecciones_ruby`
- Ejemplos de comparadores: `ComparacionJava` y `comparacion_ruby`
- Ejemplos de excepciones: `ExcepcionesJava` y `excepciones_ruby`