

**Curso**

# **Programación Orientada a Objetos (POO)**

**Instructor:** Alvar Velázquez de León Lavarrios

# Propósito del Curso

“Servir a los estudiantes como **complemento** a sus conocimientos sobre este importante paradigma de programación”.

“**Reforzar** (o enseñar) los temas que se ven en la asignatura Programación Orientada a Objetos, pero de una forma más **simplificada**.”

# Requerimientos mínimos del curso

- Tener computadora (Escritorio o Laptop)  
    ↳ Windows, MacOS o Linux.
- Conocimientos en **C** y/o **Java** (recomendado)  
    ↳ De todas formas, se enseñará en este curso
- **¡¡ Tener gusto por programar !!**  
    ↳ Practicar, practicar y más practicar...



# Temario del Curso

## 0.- Fundamentos Básicos

### 1.- Conceptos POO

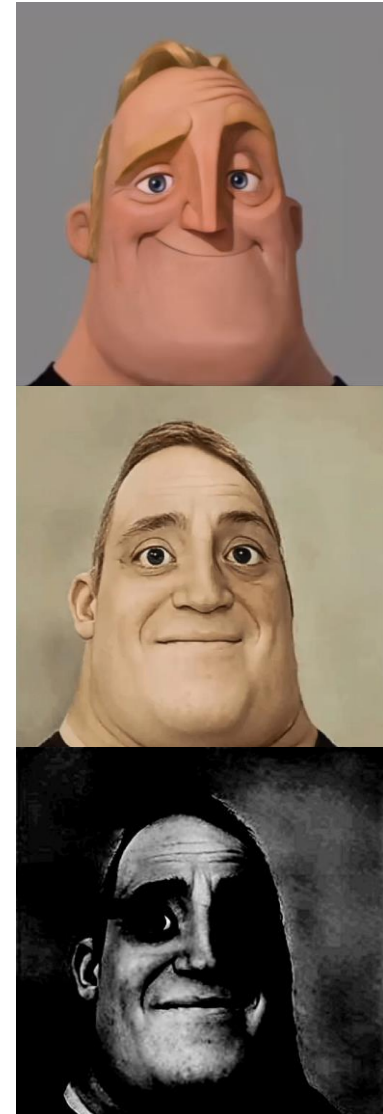
### 2.- Abstracción

### 3.- Encapsulamiento

### 4.- Herencia

### 5.- Polimorfismo

### 6.- Paquetes



**Tema 0**

# **Fundamentos Básicos**

...de la **PROGRAMACIÓN...**

- **Variables Primitivas**
- **Funciones**
- **Apuntadores**

# Variable Primitiva

## Definición Formal

“También llamado **tipo de dato primitivo**, es un término informático que se utiliza para describir un dato que existe dentro de un lenguaje de programación de computadoras de forma predeterminada.”

Fuente:

[¿Qué es el tipo de datos primitivo? - Spiegato](#)

## Otra Definición (no tan formal...)

Todo aquel **tipo de dato** que viene con el lenguaje y que almacena un **valor básico** o “**simple**” modificable.

**Int   Boolean   Float   Double**  
**Long   Short   Byte   Char**  
**¿String?**

# Tipos de Datos Primitivos

## Boolean

1 bit  
(0 o 1)



False

True

## Byte

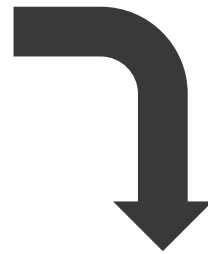
8 bits  
(1 byte)



00001111

10110010

[0, 255]



Microcomputadoras

Procesadores

Memorias

Lenguajes de Bajo  
Nivel

Diseño de  
Hardware

## Char

16 bits  
(2 bytes)



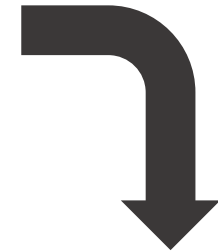
'A'

'2'

'd'

'='

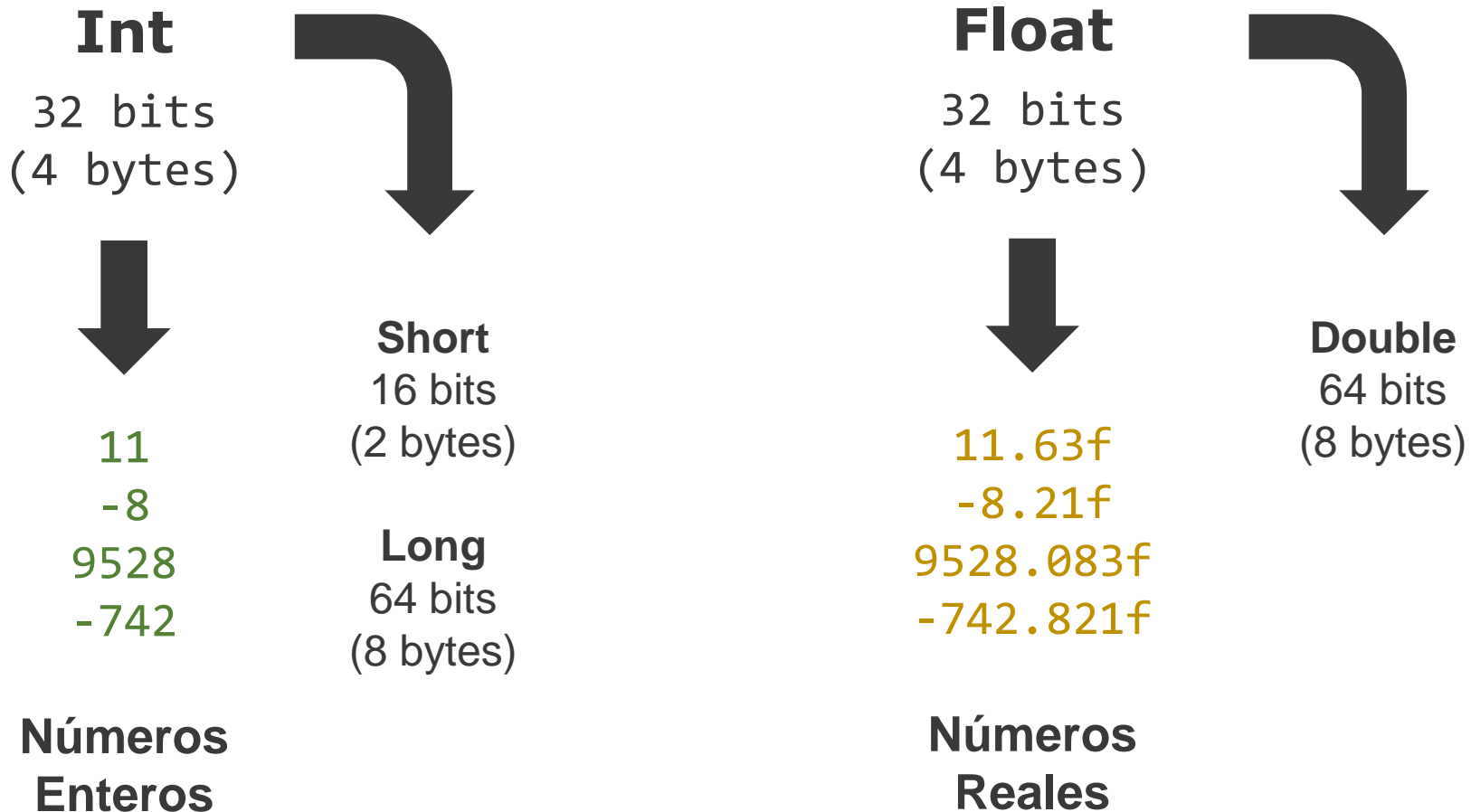
'#'



Caracteres  
UNICODE  
(dígitos, letras,  
números)

**Código ASCII**

# Tipos de Datos Primitivos





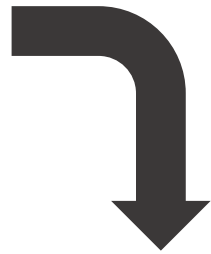
# Tipos de Datos Primitivos

**String**

? bits  
(? bytes)



“Hola”  
“ñandú\n”  
“Código”  
“Buen viaje...”



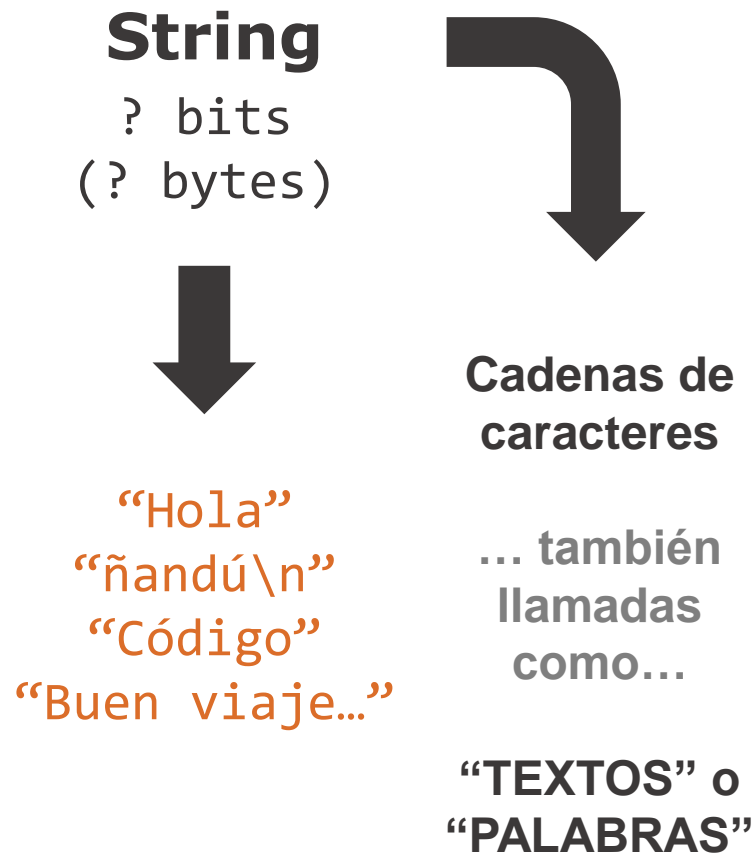
**Cadenas de  
caracteres**

... también  
llamadas  
como...

“TEXTOS” o  
“PALABRAS”

¿String, es un tipo de dato primitivo?

# Tipos de Datos Primitivos



¿String, es un tipo de dato primitivo?

**Formalmente...**

**No**, porque es un **tipo de dato complejo** de Java y otros lenguajes.

**Prácticamente...**

**Sí**, porque **su uso es tan común** que se asemeja a cualquier variable primitiva. Además de declararse como **Objeto**, puede declararse como una **variable**.

# Ejemplos: Declaración de Variables Primitivas

```
boolean mi_booleano = false;  
char mi_caracter = 'A';  
int mi_numero_entero = 15;  
float mi_numero_real = -14.56f;  
  
// Como variable  
String mi_texto_1 = "Hola Mundo";  
  
// Como Objeto  
String mi_texto_2 = new String("Hola Mundo");
```



Próximamente...

# Función

## Definición Formal

“Es un **bloque de código** que realiza una tarea específica y puede ser reutilizado en diferentes partes de un programa.”

Fuente:

[Funciones en programación: su definición y uso \(universodidactico.com.mx\)](http://universodidactico.com.mx)

## Otra Definición (no tan formal...)

**Conjunto de código** (instrucciones, variables, etc.) que, puede o no, tomar **argumentos** y/o, **retornar** o no, un tipo de dato (primitivo, complejo, vacío).

```
function MiFuncion():  
void MiFuncion() {}
```

# “Plantilla” Básica de una Función

```
TipoDato nombreFuncion(TipoDato d1, TipoDato d2, ...){  
  
    // Comentarios  
    [Bloque de Código]  
  
    return TipoDato;  
}
```

Donde:

TipoDato



Define el “tipo de función” y el tipo de dato que debe retornar (obligatorio si de definió un tipo distinto a **void**)

TipoDato  
TipoDato



(**Opcionales**) Conocidos como **argumentos**, son datos que se utilizarán dentro de la función. Cada vez que se llame a esta función, sí o sí se deben definir los argumentos con los que va a operar.

# Ejemplos: Tipos de Retorno de una Función

```
boolean miFuncion1(){  
    // Comentarios  
    [Bloque de Código]  
  
    return true;  
}
```

```
int miFuncion2(){  
    // Comentarios  
    [Bloque de Código]  
  
    return 22;  
}
```

```
float miFuncion3(){  
    // Comentarios  
    [Bloque de Código]  
  
    return 8.5f;  
}
```

```
char miFuncion4(){  
    // Comentarios  
    [Bloque de Código]  
  
    return 'A';  
}
```

```
String miFuncion5(){  
    // Comentarios  
    [Bloque de Código]  
  
    return "Hola";  
    return null;  
}
```

```
void miFuncion6(){  
    // Comentarios  
    [Bloque de Código]  
  
    return; // Opcional  
}
```

# Importancia de las Funciones

¿Por qué existen?

¿Tienen ventajas?

¿Cuál es la “función” de las funciones?

# Importancia de las Funciones

**¿Por qué existen?**

**¿Tienen ventajas?**

**¿Cuál es la “función” de las funciones?**

Definir funciones sirve para:

- Evitar repetir código
- Reutilizar código
- Ejecutar dicho bloque de código en cualquier momento del programa
- Escribir código más eficiente y fácil de identificar



# Apuntador

## Definición Formal

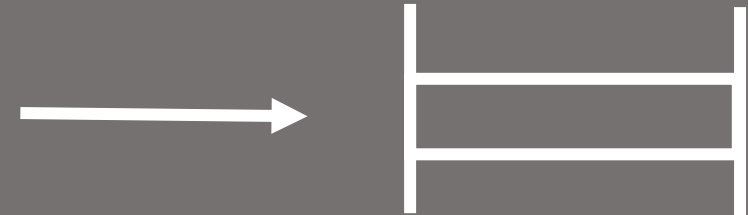
“También conocido como **puntero**, es un elemento de lenguaje de programación que almacena la **dirección de memoria** de otro valor ubicado en la memoria de la computadora.”

Fuente:

[Definición de Apuntador \(programación\) \(alegsa.com.ar\)](http://alegsa.com.ar)

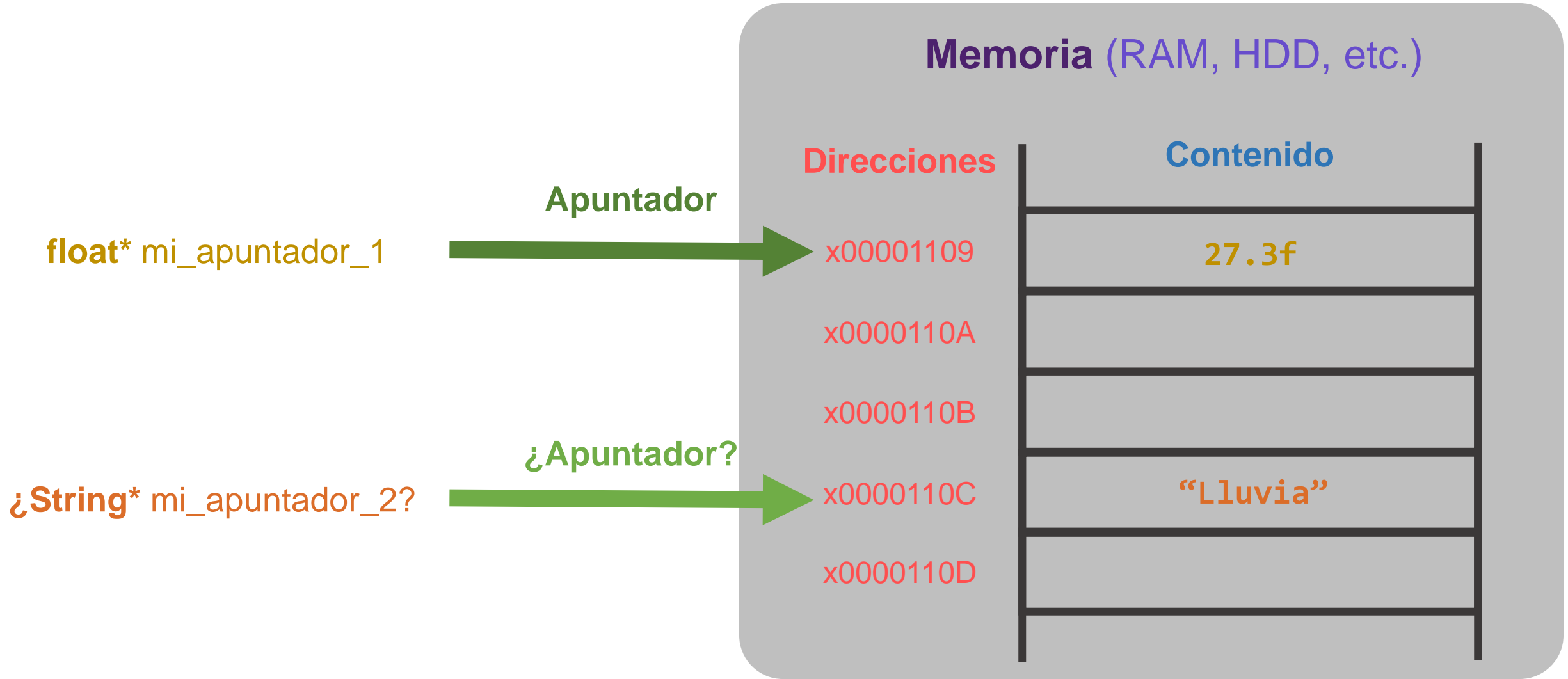
## Otra Definición (no tan formal...)

**Variable especial** (“flecha imaginaria”) que contiene una **dirección en memoria** de otra variable y que permite acceder a su **contenido**.





# Visualmente...





# ¿Apuntadores en Java?

En lenguajes de **medio nivel** (como C):

Definición de apuntadores:

```
boolean* mi_apuntador_bool;  
char* mi_apuntador_char;  
int* mi_apuntador_int;  
float* mi_apuntador_float;
```

En lenguajes de **alto nivel** (como Java):

¿Existen los apuntadores?

**Sí**, ya que... **internamente** todo lenguaje de programación hace uso de la **memoria** para almacenar y recuperar información.

¿El programador los declara como en C?

**No**, en su lugar declara algo conocido como **Referencias** (se verá en el siguiente tema...).

¿En este curso... programaremos apuntadores?

Alerta de *spoiler*..... **NOOOOOO!**

# Dicho de otra forma...



```
int* apuntador;
```



```
String referencia;
```





# Comandos Básicos de Java

Compilar un código



```
>> javac NombrePrograma.java
```

Ejecutar un código



```
>> java NombrePrograma
```

# https://github.com/alvarvelazquezdeleonlavarrios/Curso-FI-UNAM-POO

The screenshot shows a web browser displaying the GitHub repository page for 'Curso-FI-UNAM-POO' by the user 'alvarvelazquezdeleonlavarrios'. The browser's address bar shows the URL 'https://github.com/alvarvelazquezdeleonlavarrios/Curso-FI-UNAM-POO'. The repository page has a dark theme. At the top, there's a navigation bar with the GitHub logo, the repository name, and a search bar. Below this is a secondary navigation bar with links for 'Code', 'Issues', 'Pull requests', 'Actions', 'Projects', 'Wiki', 'Security', 'Insights', and 'Settings'. The repository name 'Curso-FI-UNAM-POO' is followed by a 'Public' badge. To the right of the repository name are buttons for 'Pin', 'Unwatch' (with a count of 1), 'Fork' (with a count of 0), and 'Star' (with a count of 0). Below the repository name, there's a section for branches and tags, showing 'main' as the selected branch, '1 branch', and '0 tags'. To the right of this section are buttons for 'Go to file', 'Add file', and 'Code'. The main content area displays a list of files and folders. The first item is a folder named 'Ejercicios Prácticos' with the description 'Material Inicial' and a timestamp of '1 minute ago'. The second item is a file named 'POO - Teoría.pdf' with the description 'Material Inicial' and a timestamp of '1 minute ago'. The third item is a file named 'README.md' with the description 'Initial commit' and a timestamp of '16 minutes ago'. Below the file list, there's a section for the 'README.md' file, showing the title 'Curso-FI-UNAM-POO' and a description: 'Repositorio con el material utilizado para el curso de la asignatura "Programación Orientada a Objetos" impartido en la FI UNAM.' On the right side of the repository page, there's an 'About' section with a description: 'Repositorio con el material utilizado para el curso de la asignatura "Programación Orientada a Objetos" impartido en la FI UNAM.' Below the 'About' section are links for 'Readme', 'Activity', '0 stars', '1 watching', and '0 forks'. At the bottom of the right sidebar, there's a 'Releases' section with the text 'No releases published' and a link to 'Create a new release'.

https://github.com/alvarvelazquezdeleonlavarrios/Curso-FI-UNAM-POO

Importar favoritos | Dell | Nueva pestaña

alvarvelazquezdeleonlavarrios / Curso-FI-UNAM-POO

Type to search

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

Curso-FI-UNAM-POO Public

Pin Unwatch 1 Fork 0 Star 0

main 1 branch 0 tags

Go to file Add file Code

alvarvelazquezdeleonlavarrios Material Inicial 0F6096a 1 minute ago 2 commits

Ejercicios Prácticos	Material Inicial	1 minute ago
POO - Teoría.pdf	Material Inicial	1 minute ago
README.md	Initial commit	16 minutes ago

README.md

## Curso-FI-UNAM-POO

Repositorio con el material utilizado para el curso de la asignatura "Programación Orientada a Objetos" impartido en la FI UNAM.

About

Repositorio con el material utilizado para el curso de la asignatura "Programación Orientada a Objetos" impartido en la FI UNAM.

Readme Activity 0 stars 1 watching 0 forks

Releases

No releases published

Create a new release

## Tema 1

# Conceptos POO

...FUNCIONAMIENTO BASE  
de este paradigma...

- Objeto
- Clase
- Referencia a Objeto

# ¿POO?

**POO** → **P**rogramación **O**rientada a **O**bjetos

del Inglés, *Object Oriented Programming* (OOP)

...pero...

¿Qué es un **Objeto**?



# Objeto

## Definición General

“Toda aquella **cosa física o conceptual** que tiene **propiedades y comportamientos**.”

Jugador



Arma



Coleccionable



Fuente:

[¿Qué es un Objeto? - Platzi](#)

# Objeto

## Definición para POO

“Es una **entidad** que representa **información** sobre una cosa **dentro del** código de un **programa**.”

Es una **instancia** de una **clase** definida.”



**Jugador**

```
<91a209d81ce9785  
297501f8164ba861  
4c978f178023ec2>
```

**¿Instancia?...**      **¿Clase?...**

Lo veremos a continuación...

Fuente:

[Qué es un objeto en Java, cuáles son sus elementos y cómo crearlo \(hubspot.es\)](https://www.hubspot.es/es/blog/que-es-un-objeto-en-java-cuales-son-sus-elementos-y-como-crearlo)

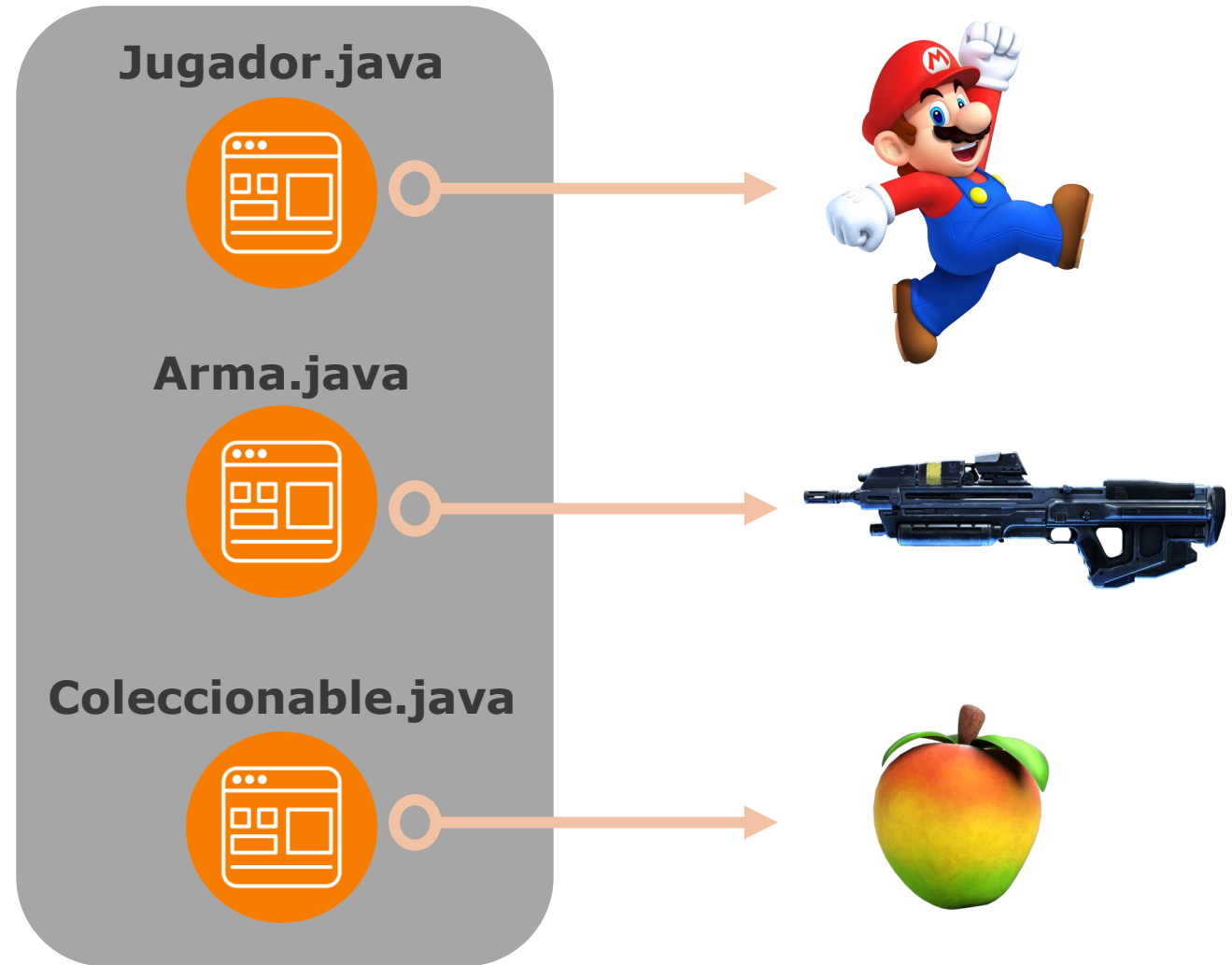
# Clase

## Definición para POO

“Es un modelo mediante el cual se construyen los objetos. Son los ‘**moldes**’ que permiten **generar nuevos objetos**.”

Fuente:

[Abstracción: ¿Qué es una Clase? - Platzi](#)



# Instancia

## Definición para POO

“Es un **objeto concreto** que tiene su **propio estado** (atributos) y su **propio comportamiento** (métodos).”

Coleccionable.java



(Instancia 1)

<91a209d81ce978  
5297501f8164ba8  
614c978f178023e  
c2>



(Instancia 2)

<26a19294bc66f0  
5c03478a070315b  
87409e8590219d5  
6e>

¿Atributos?... ¿Métodos?...

Lo veremos en el siguiente tema...

Fuente:

[Qué es una instancia de clase en Java y cómo crearla \(hubspot.es\)](https://www.hubspot.es/es/que-es-una-instancia-de-clase-en-java-y-como-crearla)

# En Código...

¿Se acuerdan de esto?



```
String mi_texto = new String("Hola Mundo");
```

# En Código...

¿Se acuerdan de esto?

The diagram illustrates the components of the Java code `String mi_texto = new String("Hola Mundo");` with the following annotations:

- Clase** (Class): A red bracket above the word `String` on the left.
- Nombre de variable** (Variable name): A green bracket above `mi_texto`.
- Clase** (Class): A red bracket above the word `String` on the right.
- (Pendiente...)** (Pending...): A grey bracket above the opening curly brace of the constructor.
- Objeto** (Object): A purple bracket below the entire constructor expression `new String("Hola Mundo")`.
- Instancia** (Instance): A blue arrow pointing from the `new` keyword to the word `Instancia`.
- ¡¡ Referencia !!** (Reference): An orange bracket below `mi_texto` with the text `¡¡ Referencia !!` below it.

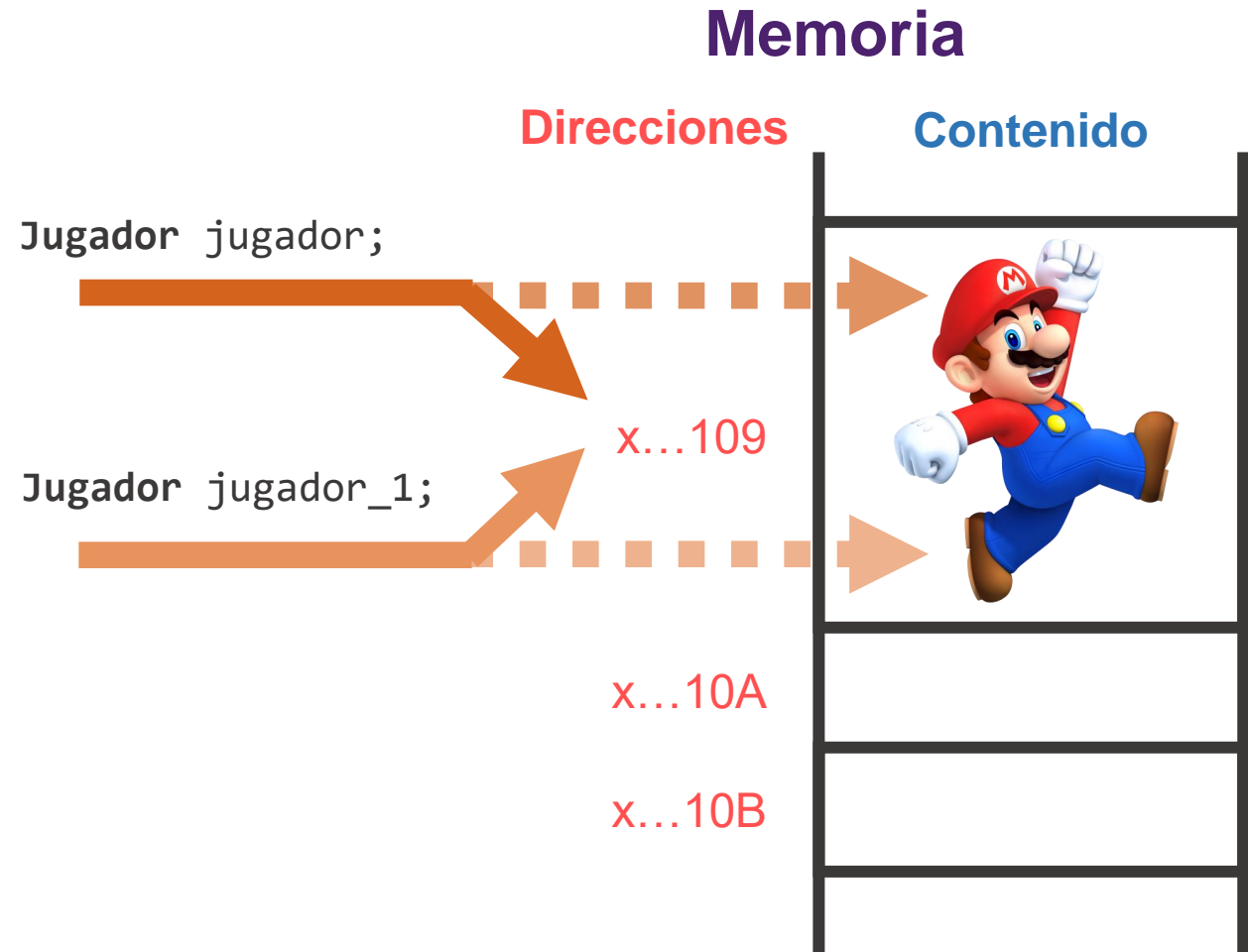
# Referencia a Objeto

## Definición para POO

“Es una **variable** que almacena la **dirección de memoria** de un **objeto**, y mediante la cual se puede acceder al **contenido** del mismo.

Fuente:

[¿Qué es una REFERENCIA en Programación } - Ejemplos en Unity \(gamedevtraum.com\)](http://gamedevtraum.com)



## En resumen...

**Objeto** → Cosa representada en código. Lo interpreta la computadora. A la derecha de una asignación.

**Clase** → “Plantilla” para generar objetos. Lo programamos nosotros.

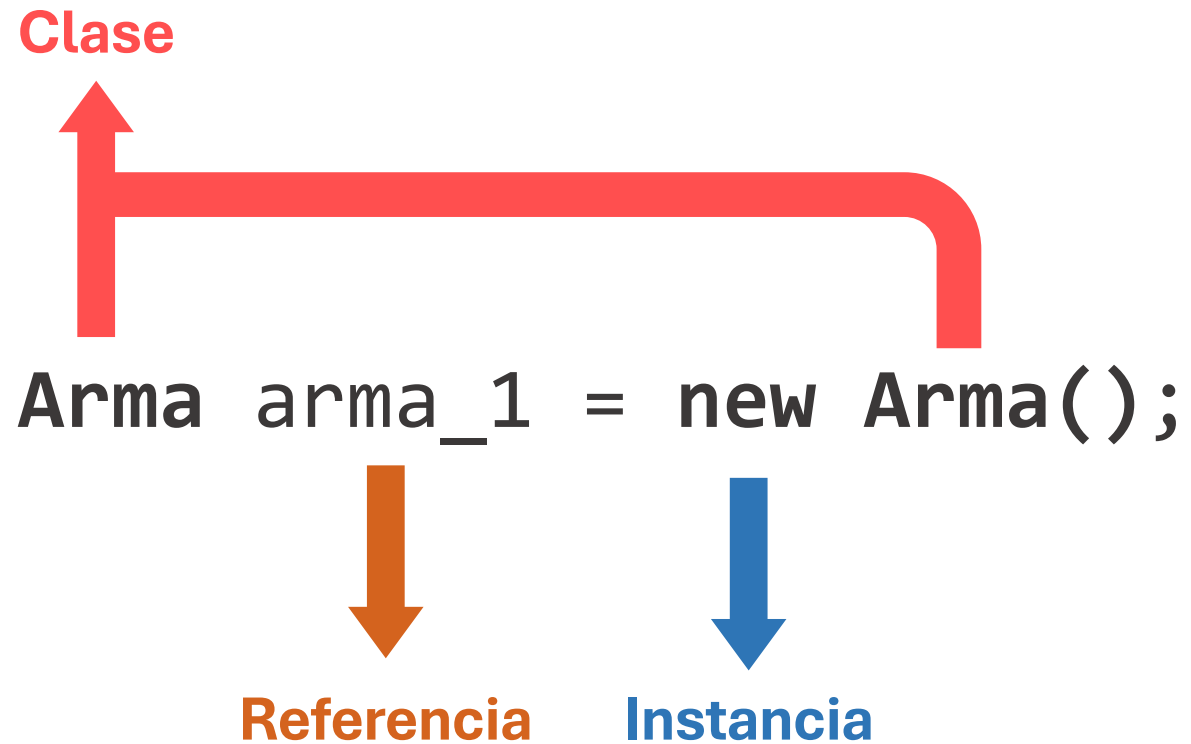


## En resumen...

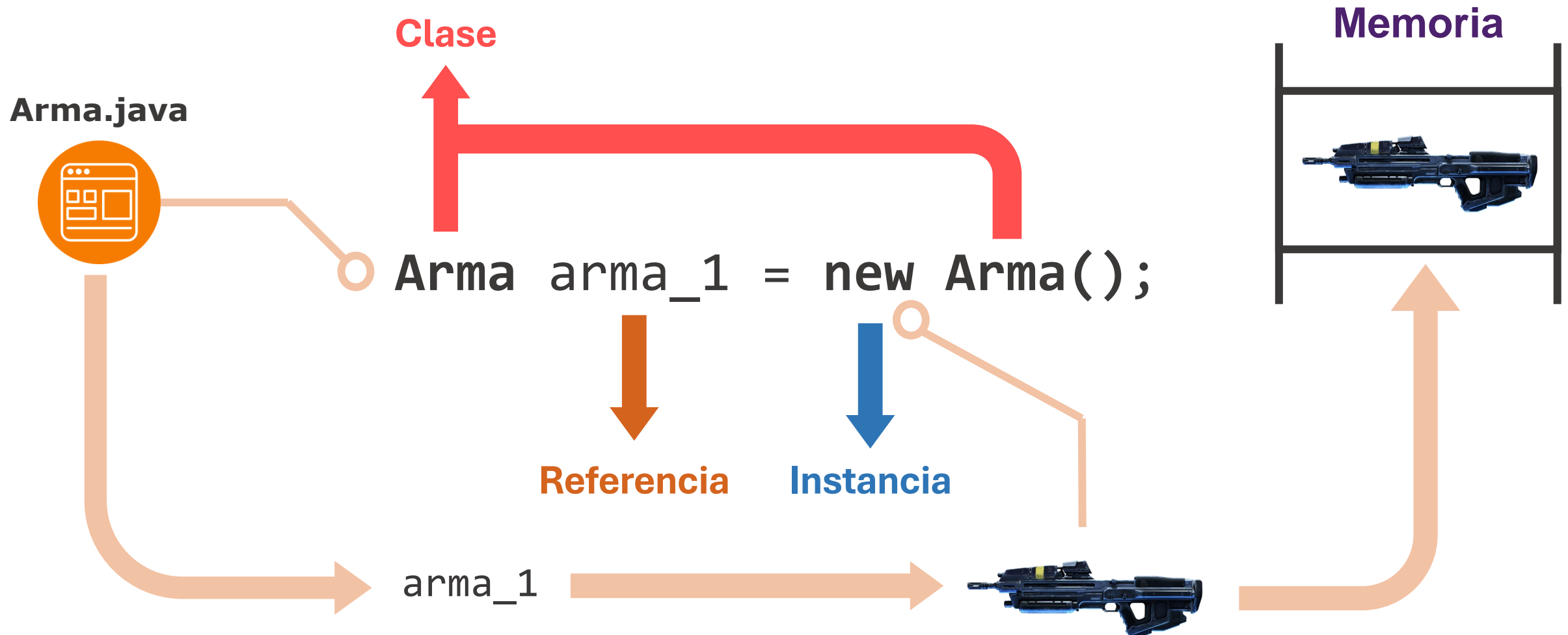
**Instancia** → Nuevo **objeto** creado desde la **clase**. Lo programamos nosotros.

**Referencia** → Variable que “apunta” a un **objeto** en concreto. A la izquierda de una asignación.

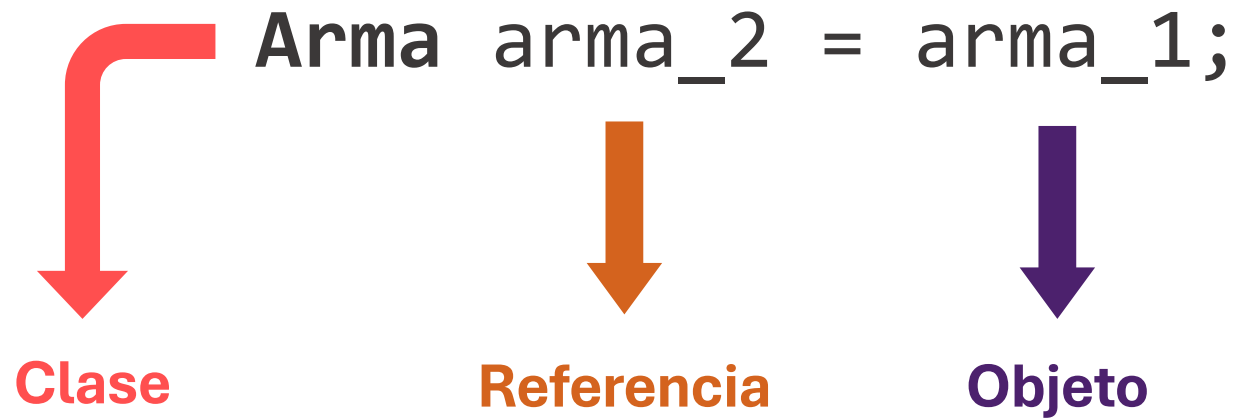
# Resumen... ¡pero visual!



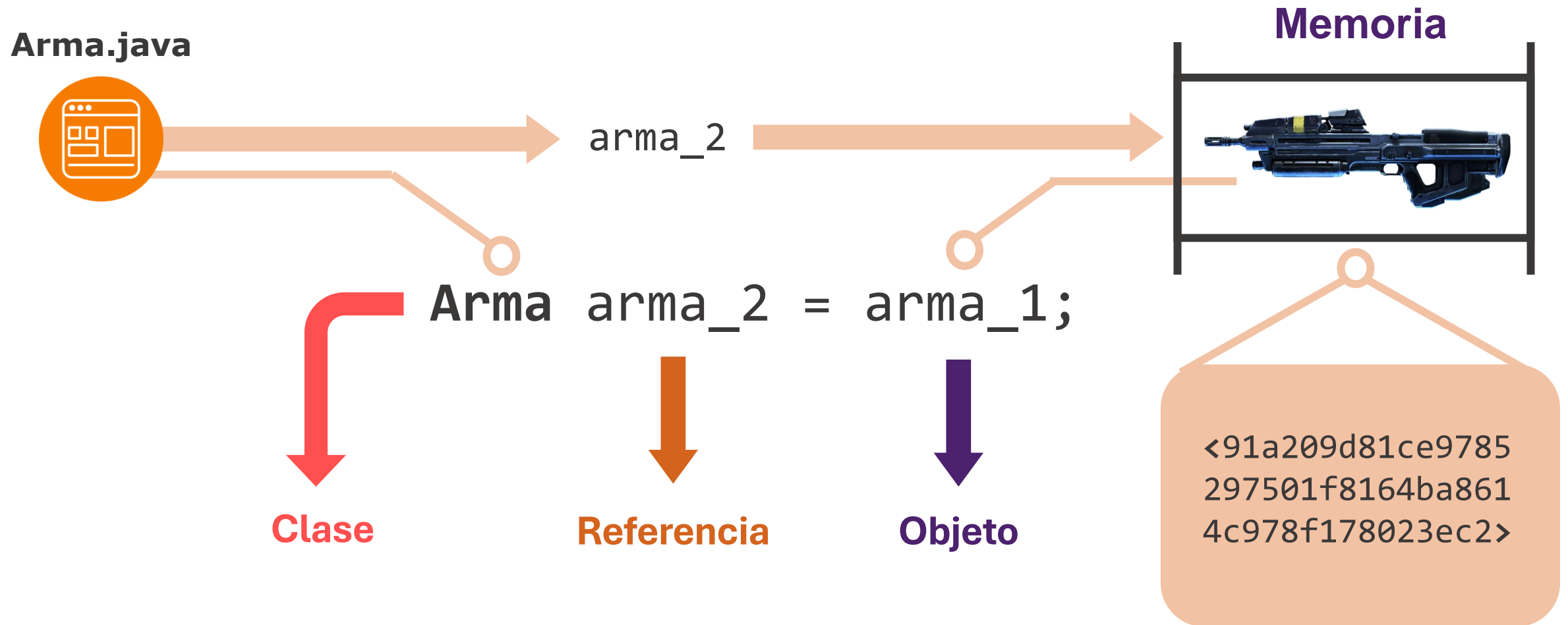
# Resumen... ¡pero visual!



# Resumen... ¡pero visual!



# Resumen... ipero visual!



## Tema 2

# Abstracción

...**"plantilla"** de una clase...

- **Identificador de Clase**
- **Atributos**
- **Métodos**

# Identificador de Clase

## Definición

Es el **nombre** que define a una clase.

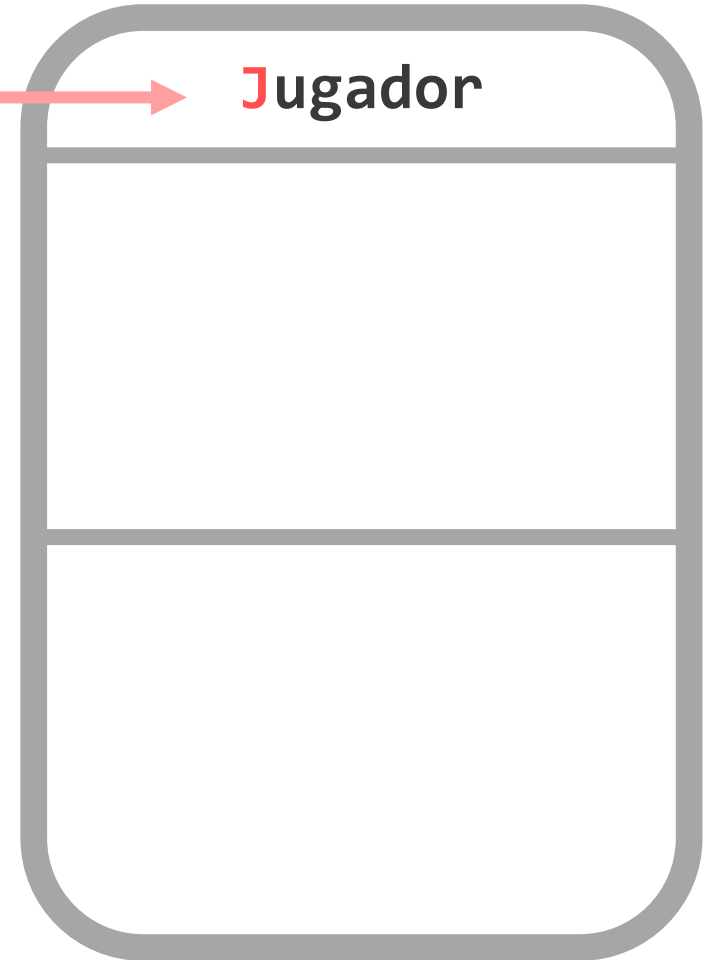
### [Obligatorio]

¡¡ Debe iniciar con letra mayúscula !!

Jugador.java



Jugador



# Atributo

## Definición

Es todo aquel **campo (variable o constante)** que posee una clase.

Define un **estado** de un **objeto** en concreto.

Jugador.java



```
String nombre = "Mario";  
int vidas = 3;  
float velocidad_mov = 2f;  
boolean en_suelo = true;
```



# Método

## Definición

Es toda aquella **función** que posee una clase.

Define un **comportamiento** de un **objeto**.

## Jugador.java



```
Jugador(String nombre) {}  
→ ¿?  
  
void saltar() {}  
void mover() {}  
boolean puedeMoverse() {}
```

# ¿¡Otra Vez!?

¿Se acuerdan de esto?



(Pendiente...)

```
String mi_texto = new String("Hola Mundo");
```

# ¿¡Otra Vez!?

¿Se acuerdan de esto?



¡¡ Constructor !!

```
String mi_texto = new String("Hola Mundo");
```

# Constructor

## Definición

“**Método (función) especial** usado para **inicializar objetos** de una clase”.

**this** → Palabra clave que refiere a un **objeto actual**, diferenciando entre los atributos (**campos**) de la clase y los **argumentos** de un método o constructor.

## Jugador.java



```
Jugador(String nombre){  
    this.nombre = nombre;  
}
```

j.nombre → “Mario”



```
Jugador j =  
new Jugador(“Mario”);
```

Fuentes:

[Java Constructors \(w3schools.com\)](https://www.w3schools.com/java/java_constructors.asp)

[Java this Keyword \(w3schools.com\)](https://www.w3schools.com/java/java_this.asp)

## ¡¡ Importante #1 !!

Para consultar el valor de un **atributo** o utilizar un **método** de un objeto, se utiliza el caracter **punto** ‘.’

Atributos



```
mi_jugador.nombre;
```

```
String nombre =  
"Mario";
```

Métodos



```
mi_jugador.saltar();
```

```
void saltar() {}
```

# Ejemplo

Primero, tenemos la clase **Jugador**...

**Jugador.java**



```
class Jugador {  
  
    String nombre = "Mario";  
  
    void saltar() {  
        print("El jugador ha saltado")  
    }  
}
```

# Ejemplo

Luego, tenemos el siguiente **código principal**...

```
public static void main(String[] args) {  
  
    Jugador mi_jugador = new Jugador();  
  
    print(mi_jugador.nombre);  
  
    mi_jugador.saltar();  
}
```

# Ejemplo

El resultado de la **ejecución** sería...

Para las variables...

```
print(mi_jugador.nombre);
```

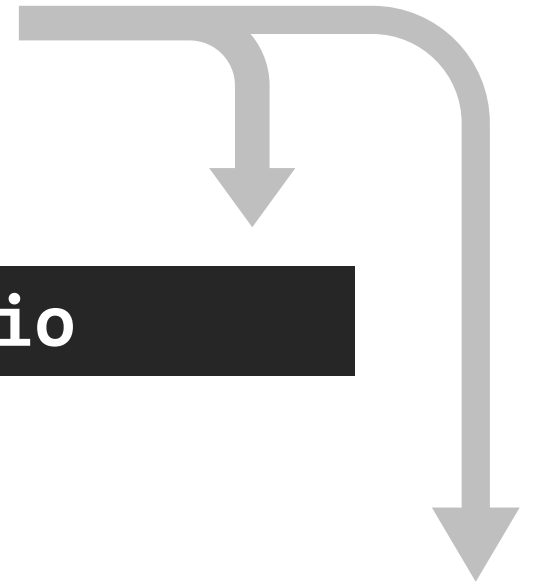
En la **Terminal**...

```
>> Mario
```

Para las funciones...

```
mi_jugador.saltar();
```

```
>> El jugador ha saltado
```





## ii Importante #2 !!

Previamente dijimos que...

```
Arma arma_2 = arma_1;
```



**Referencia**



**Objeto**

## ii Importante #2 !!

Arma.java



Previamente dijimos que...

arma\_2

Memoria



Arma arma\_2 = arma\_1;

Referencia

Objeto

<91a209d81ce9785  
297501f8164ba861  
4c978f178023ec2>

## ii Importante #2 !!

Si en lugar de eso, tenemos ahora...

```
arma_2.balas = arma_1.balas;
```



**Objeto #2**



**Objeto #1**

## ii Importante #2 !!

Si en lugar de eso, tenemos ahora...

Arma #2



Antes...

```
int balas = 15;
```

Ahora...

```
int balas = 36;
```

arma\_2.balas = arma\_1.balas;

Objeto #2

Objeto #1

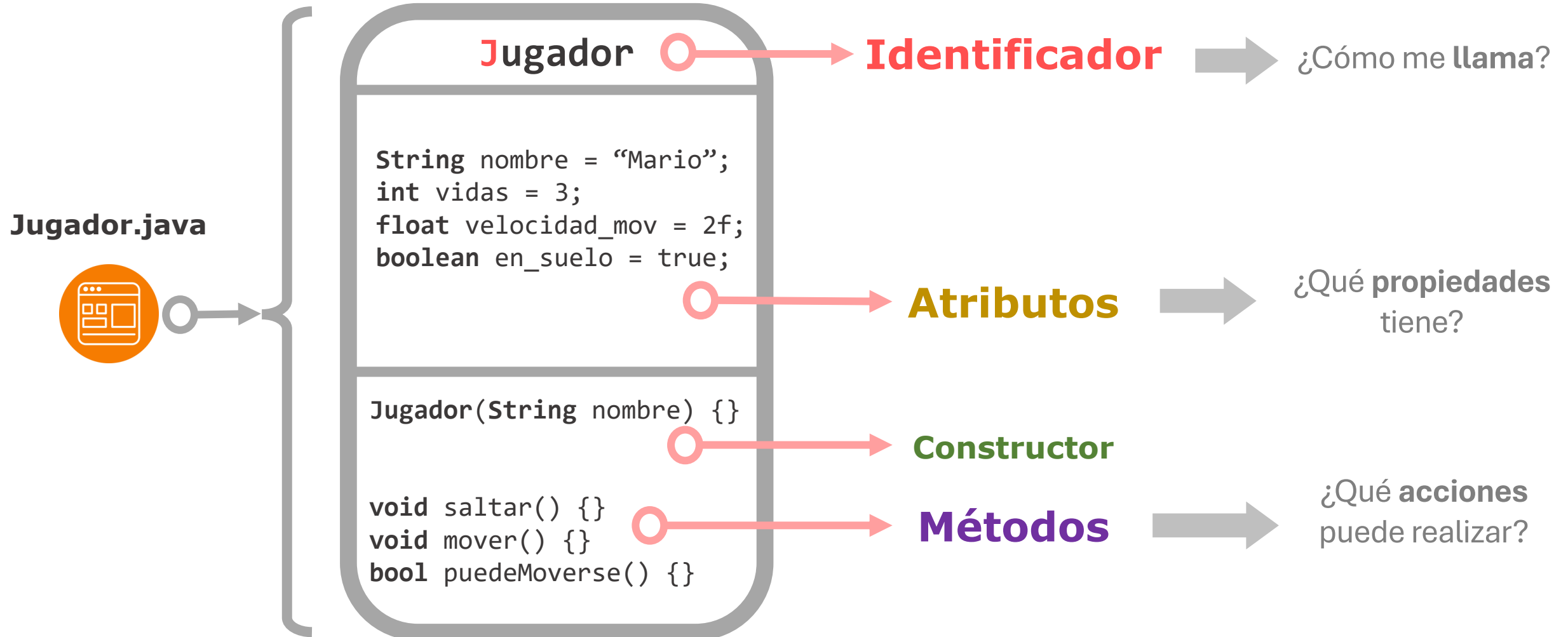
Arma #1



```
int balas = 36;
```

**¡OJO!** Está consultando un **dato primitivo**

# Recapitulando...



## Tema 3

# Encapsulamiento

... o debería decir...**escudo**  
de una clase...

- **Alcance**
- **Aspectos de Seguridad**
- **Funciones *Get* y *Set***

# Anteriormente...

Ya sabemos como declarar **variables**...

```
String nombre = "Mario";  
int vidas = 3;  
float velocidad_mov = 2f;  
boolean en_suelo = true;
```

... y también **funciones**.

```
void saltar() {}  
void mover() {}  
boolean puedeMoverse() {}
```

# Anteriormente...

También podemos crear múltiples **clases**.

**Jugador.java**



**Arma.java**



**Coleccionable.java**





# Anteriormente...

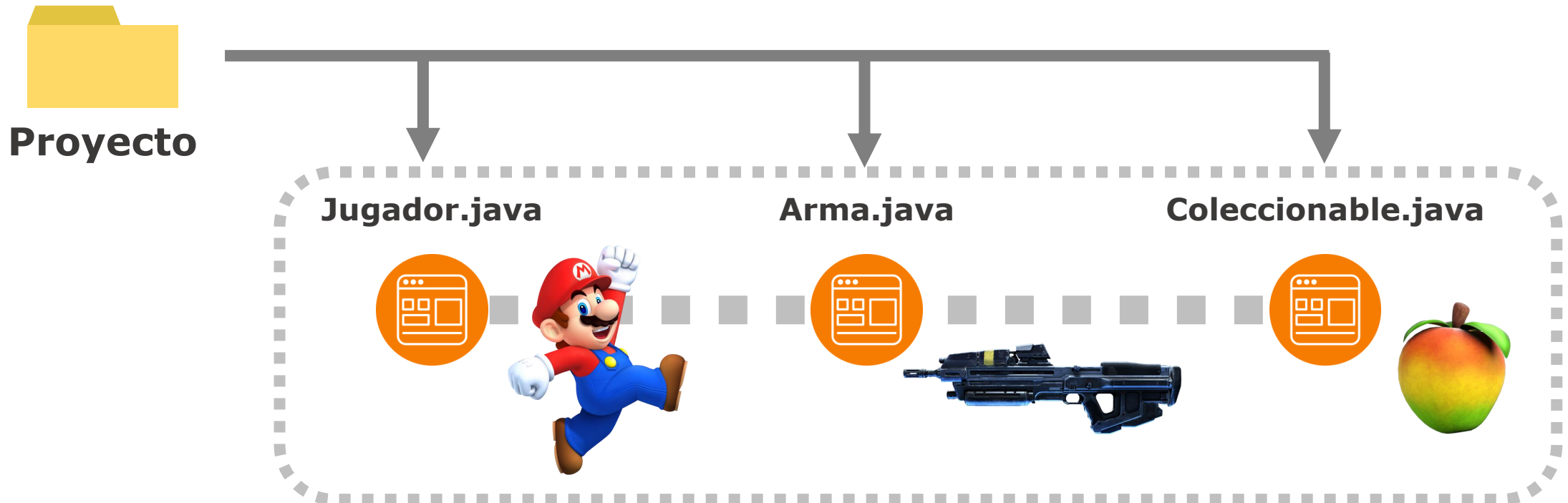
También podemos crear múltiples **clases**.



Pero.... ¿pueden **interactuar** entre sí?

# Respuesta...

**Sí** pueden. Solo ten en cuenta que las clases deben estar en una **misma carpeta**.



# Ejemplo...

## Jugador.java



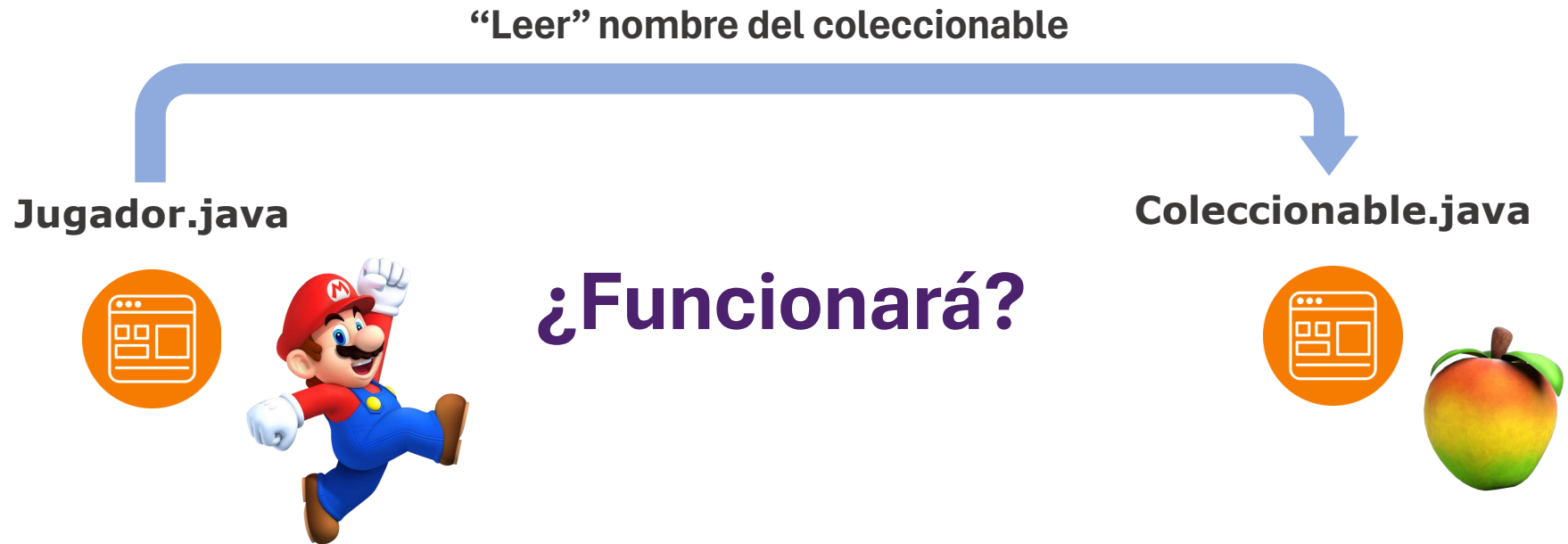
```
void funcion(){  
    Coleccionable fruta = new Coleccionable();  
  
    System.out.println(fruta.nombre);  
}
```

## Coleccionable.java



```
String nombre = "Fruta Wumpa";  
Coleccionable() {}
```

# Ejemplo...

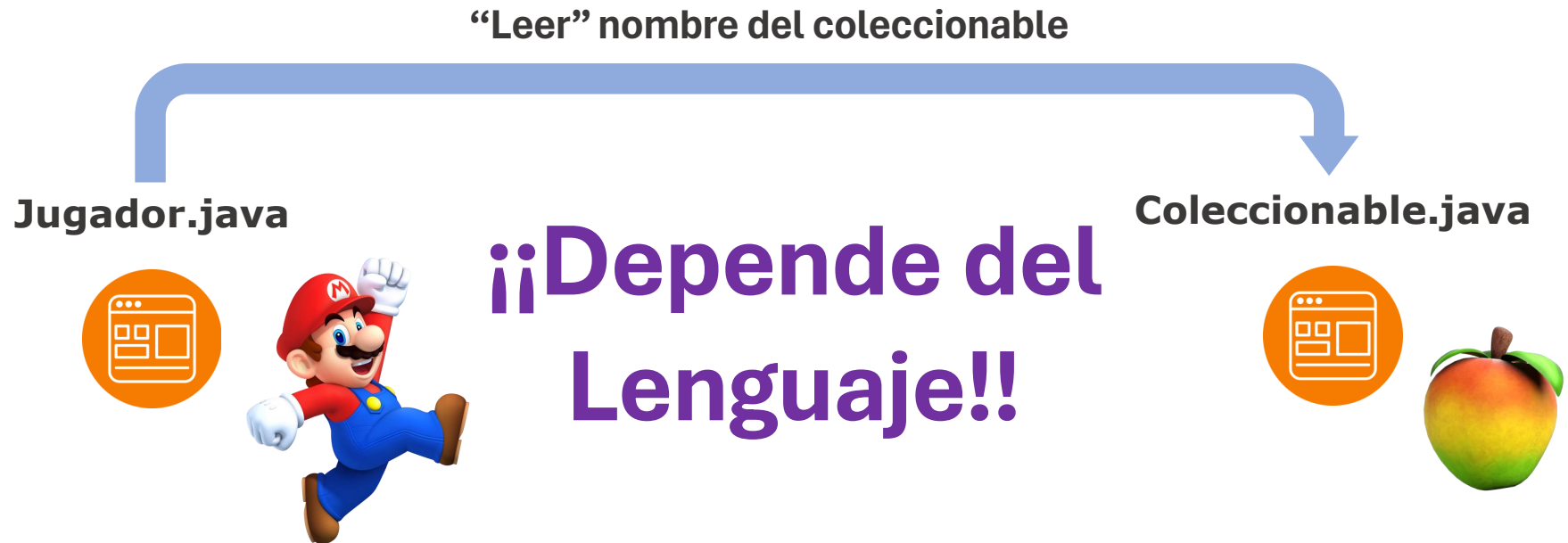


```
void funcion(){
    Coleccionable fruta = new Coleccionable();

    System.out.println(fruta.nombre);
}
```

```
String nombre = "Fruta Wumpa";
Coleccionable() {}
```

# Ejemplo...



```
void funcion(){  
    Coleccionable fruta = new Coleccionable();  
  
    System.out.println(fruta.nombre);  
}
```

```
String nombre = "Fruta Wumpa";  
Coleccionable() {}
```



# Ejemplo en Java

“Leer” nombre del coleccionable

Jugador.java



¡Sí  
funcionará!



Coleccionable.java



```
void funcion(){  
    Coleccionable fruta = new Coleccionable();  
  
    System.out.println(fruta.nombre);  
}
```

```
String nombre = “Fruta Wumpa”;  
Coleccionable() {}
```

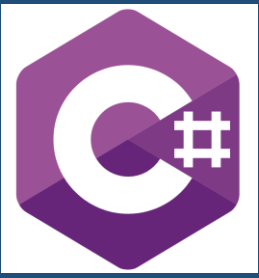


# Ejemplo en C#



```
void funcion(){  
    Coleccionable fruta = new Coleccionable();  
  
    System.out.println(fruta.nombre);  
}
```

```
String nombre = "Fruta Wumpa";  
Coleccionable() {}
```



# Ejemplo



¿Por qué **no**  
funcionará en **C#**?

Porque la variable no tiene el **alcance**  
necesario para que pueda “**ser vista**” desde  
**clases externas**.

Jugador.cs



Coleccionable.cs



```
String nombre =  
"Fruta Wumpa";
```



# Entonces...

¿Cómo hacerla  
“visible” para  
otras clases?

Con un modificador de acceso llamado....

**public**



# Public

## Definición

**Modificador de acceso** usado para declarar **atributos** o **métodos** de una clase y que puedan ser **accesibles**, por **referencia**, desde cualquier clase externa.

Crea variables y funciones que sean públicas para todas las clases.

### Jugador.java



### Arma.java



### Coleccionable.java



#### Coleccionable

```
public String nombre =  
"Fruta Wumpa";
```

```
public void utilizar()  
{}
```



## ii Importante !!

Sintaxis de declaración de **atributo**



```
public String nombre = "Fruta Wumpa";
```



**Alcance  
del  
atributo**



**Clase  
(tipo de  
atributo)**



**Atributo  
(variable)**



**Inicialización  
de la variable**

## ii Importante !!

Sintaxis de declaración de **método**



```
public void utilizar() {...}
```

The diagram illustrates the components of a method declaration. A grey arrow points from the text 'Sintaxis de declaración de método' to the code snippet. Below the code, four colored arrows point to their respective parts: a blue arrow for 'public', a green arrow for 'void', an orange arrow for 'utilizar()', and a grey arrow for '{...}'.

**Alcance  
del método**

**Tipo de  
retorno**

**Método  
(función)**

**Bloque de  
código de la  
función**



# Volviendo al Ejemplo en C#



“Leer” nombre del coleccionable

Jugador.cs



¿Funcionará?

Coleccionable.cs



```
void funcion(){  
    Coleccionable fruta = new Coleccionable();  
  
    System.out.println(fruta.nombre);  
}
```

```
public String nombre =  
    "Fruta Wumpa";  
  
Coleccionable() {}
```



# Volviendo al Ejemplo en C#



“Leer” nombre del coleccionable

Jugador.cs



¡Sí  
funcionará!



Coleccionable.cs



```
void funcion(){  
    Coleccionable fruta = new Coleccionable();  
  
    System.out.println(fruta.nombre);  
}
```

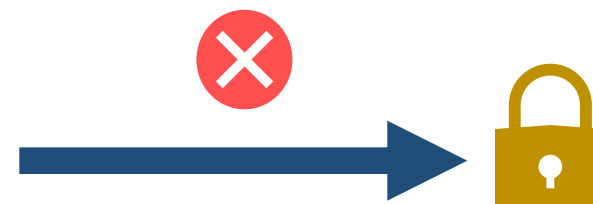
```
public String nombre =  
    "Fruta Wumpa";  
  
Coleccionable() {}
```

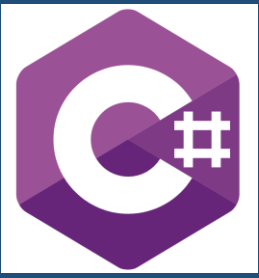
# Ahora...

Es pública... ¡se puede acceder!

Pero, otra pregunta...

¿Y si **NO quiero** que la “vean”  
otras clases?





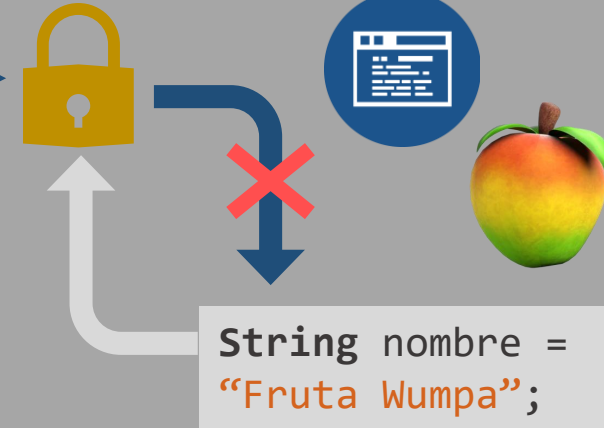
# ¿Solución?



Jugador.cs



Coleccionable.cs



Sin embargo.... ¡**NO** está garantizado!



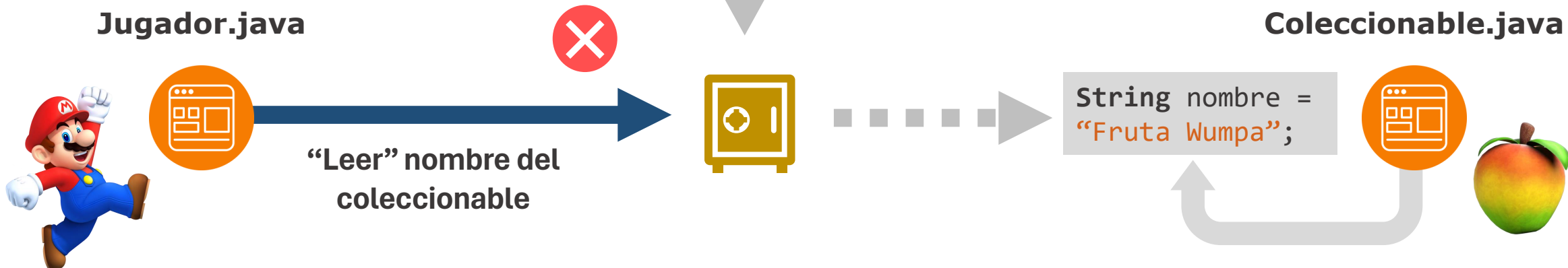


# Entonces...

¿Cómo hacerla  
“invisible” para  
otras clases?

Con una modificador de acceso llamado....

**private**



# Private

## Definición

**Modificador de acceso** usado para declarar **atributos** o **métodos** de una clase y que sean accesibles únicamente dentro de la **misma clase**.

Crea variables y funciones que **no** puedan ser consultadas por clases externas.

### Jugador.java



### Arma.java



### Coleccionable.java



#### Coleccionable

```
private String nombre =  
"Fruta Wumpa";
```

```
private void utilizar()  
{}
```



# De igual forma...

Sintaxis de declaración de **atributo**



```
private String nombre = "Fruta Wumpa";
```



**Alcance  
del  
atributo**



**Clase  
(tipo de  
atributo)**



**Atributo  
(variable)**



**Inicialización  
de la variable**

# De igual forma...

Sintaxis de declaración de **método**



```
private void utilizar() {...}
```

The diagram illustrates the syntax of a method declaration. A grey arrow points from the text 'Sintaxis de declaración de método' to the code snippet. Below the code, four colored arrows point to their respective parts: a blue arrow for 'private', a green arrow for 'void', an orange arrow for 'utilizar()', and a grey arrow for '{...}'.

**Alcance  
del método**

**Tipo de  
retorno**

**Método  
(función)**

**Bloque de  
código de la  
función**



# Volviendo al Ejemplo en Java

“Leer” nombre del coleccionable

Jugador.java



¿Funcionará?

Coleccionable.java



```
void funcion(){  
    Coleccionable fruta = new Coleccionable();  
  
    System.out.println(fruta.nombre);  
}
```

```
private String nombre =  
    "Fruta Wumpa";  
  
Coleccionable() {}
```



# Volviendo al Ejemplo en Java



```
void funcion(){  
    Coleccionable fruta = new Coleccionable();  
  
    System.out.println(fruta.nombre);  
}
```

```
private String nombre =  
    "Fruta Wumpa";  
  
Coleccionable() {}
```

# En general...

Existen **4 modificadores de acceso** para atributos y métodos:

**public**  `public void funcion()`

**private**  `private void funcion()`

*(sin modificador)*  `void funcion()`

**protected**  `protected void funcion()`



¡Tienen que ver con algo llamado **Paquetes**!

Modificador de Acceso	Nivel de Accesibilidad → ¿Qué clases pueden “ver” el atributo o método declarado?				
	Clase	Paquete	Subclase (mismo paquete)	Subclase (diferente paquete)	Todos
<b>public</b>	✓	✓	✓	✓	✓
<b>private</b>	✓	✗	✗	✗	✗
<i>(sin modificador)</i>	✓	✓	✓	✗	✗
<b>protected</b>	✓	✓	✓	✓	✗

Fuente:

[What is the difference between public, protected, package-private and private in Java? - Stack Overflow](#)

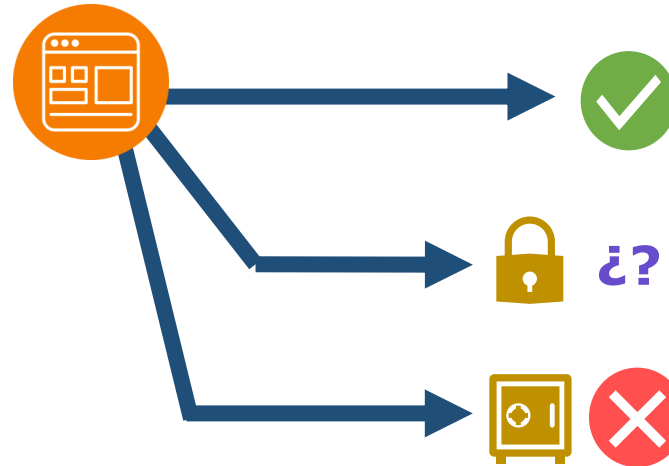


# Alcance

## Definición

**Propiedad** de una clase, atributo o método que define su **accesibilidad** (“visibilidad”) para **otras clases** y la clase misma.

### ClaseExterna.java



### Clase.java



```
public int a = 1;
```

```
int b = 2;
```

```
private int c = 3;
```

# Aspectos de Seguridad

**Public** permite acceder a una variable o función desde cualquier **clase externa**.



## Jugador

```
public String nombre = "Mario";  
public int vidas = 3;
```

```
public Jugador(String nombre)  
{}
```

```
public void saltar() {}  
public void mover() {}
```

Jugador.java



# Aspectos de Seguridad

Sin embargo...

¡ **No** es buena  
práctica manejar  
todo como **public** !

## Jugador

```
public String nombre = "Mario";  
public int vidas = 3;
```

```
public Jugador(String nombre)  
{}
```

```
public void saltar() {}  
public void mover() {}
```

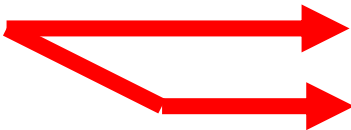
Jugador.java



# Aspectos de Seguridad

¿Por qué?...

Los hackers



Jugador.java

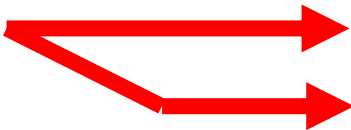


```
public String nombre = "Mario";  
public int vidas = 3;
```

# Aspectos de Seguridad

¿Por qué?...

Los hackers



Jugador.java



GAME  
OVER

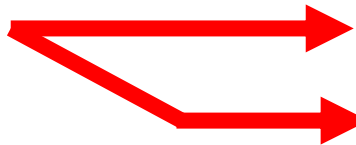


```
public String nombre = "Muerto";  
public int vidas = -1;
```

# Aspectos de Seguridad

¿Solución? Usar **private**

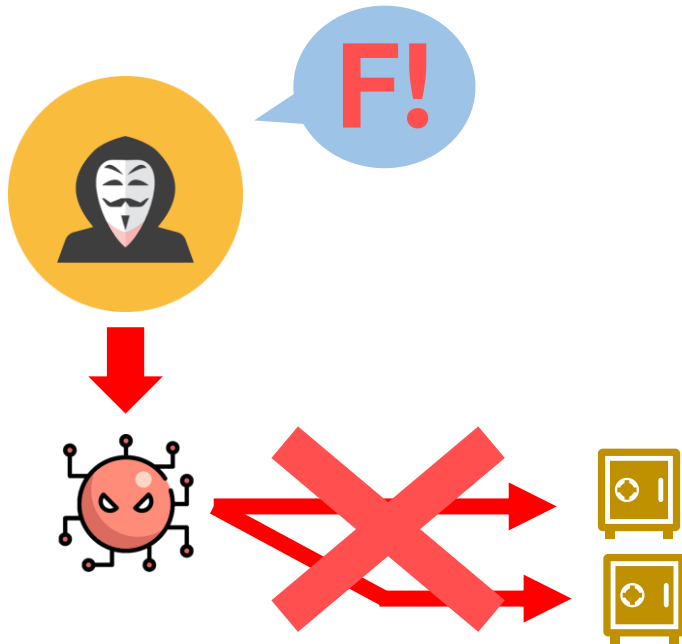
Jugador.java



```
private String nombre = "Mario";  
private int vidas = 3;
```

# Aspectos de Seguridad

¿Solución? Usar **private**



Jugador.java



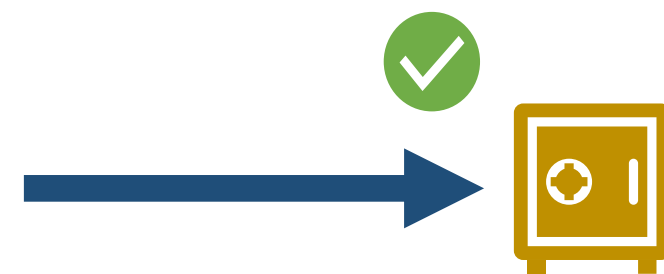
```
private String nombre = "Mario";  
private int vidas = 3;
```

# Ahora...

Es privada... ¡se protege!

Pero...

**¿Y si necesito “ver” la variable  
o función?**





# Función *Get*

Principal.java



```
void funcion1(){  
    Jugador j = new Jugador();  
    print(j.getVidas());  
}
```

¿Imprime algo aquí?

Sí

>> 3

Jugador.java



```
private int vidas = 3;  
  
public int getVidas(){  
    return vidas;  
}
```

# Función Set

Principal.java



¿Modifica el valor?

Sí

>> 6

Jugador.java



```
void funcion2(){  
    Jugador j = new Jugador();  
    j.setVidas(6);  
}
```

```
private int vidas = 3;  
  
public void setVidas(int vidas){  
    this.vidas = vidas;  
}
```

## Tema 4

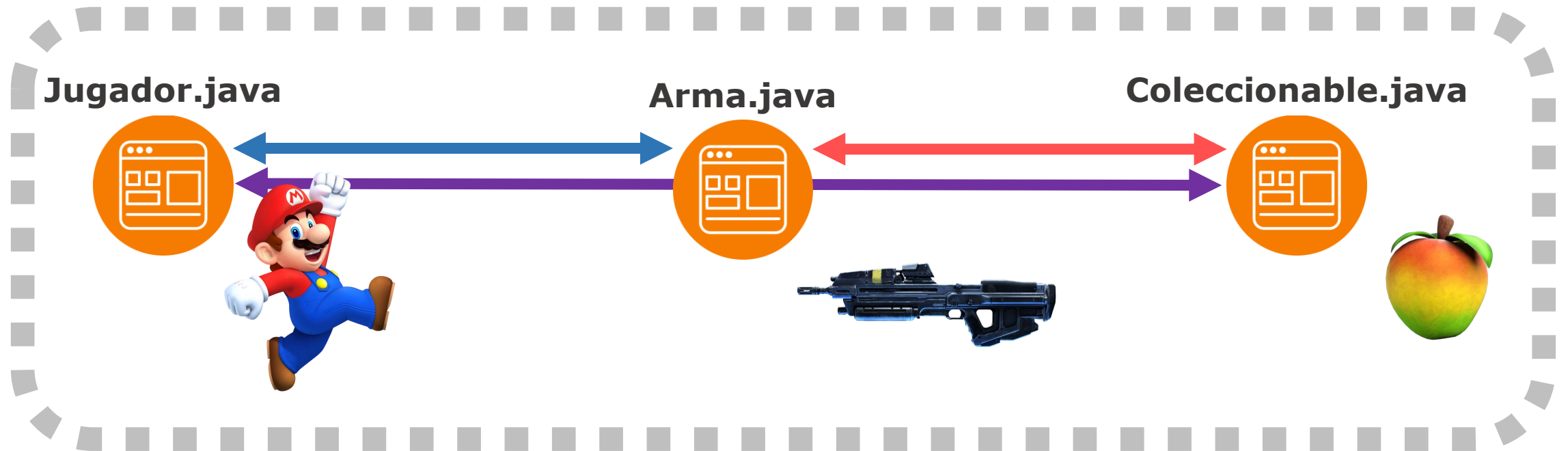
# Herencia

... mi **familia** y yo (de  
clases)...

- **Relaciones entre clases**
- **Transferencia de atributos y métodos**

# Anteriormente...

Interacciones entre clases...



¿otra forma de **relación** más “profunda”?

# Ejemplo

## ¿Comparten cosas en común?

Goomba.java



```
String nombre = "Goomba";  
float vida = 20f;  
  
void mover() {}  
void correr() {}
```

Koopa.java



```
String nombre = "Koopa";  
float vida = 40f;  
float tiempo_escondite = 5f;  
  
void mover() {}  
void rodar() {}
```

# Ejemplo

Goomba.java



¡ Así es !

Koopa.java



```
String nombre = "Goomba";  
float vida = 20f;  
  
void mover() {}  
void correr() {}
```

```
String nombre = "Koopa";  
float vida = 40f;  
float tiempo_escondite = 5f;  
  
void mover() {}  
void rodar() {}
```

# Entonces... se puede **generalizar**

Enemigos **específicos**

**Goomba.java**



```
String nombre = "Goomba";  
float vida = 20f;  
  
void mover() {}  
void correr() {}
```

**Koopa.java**



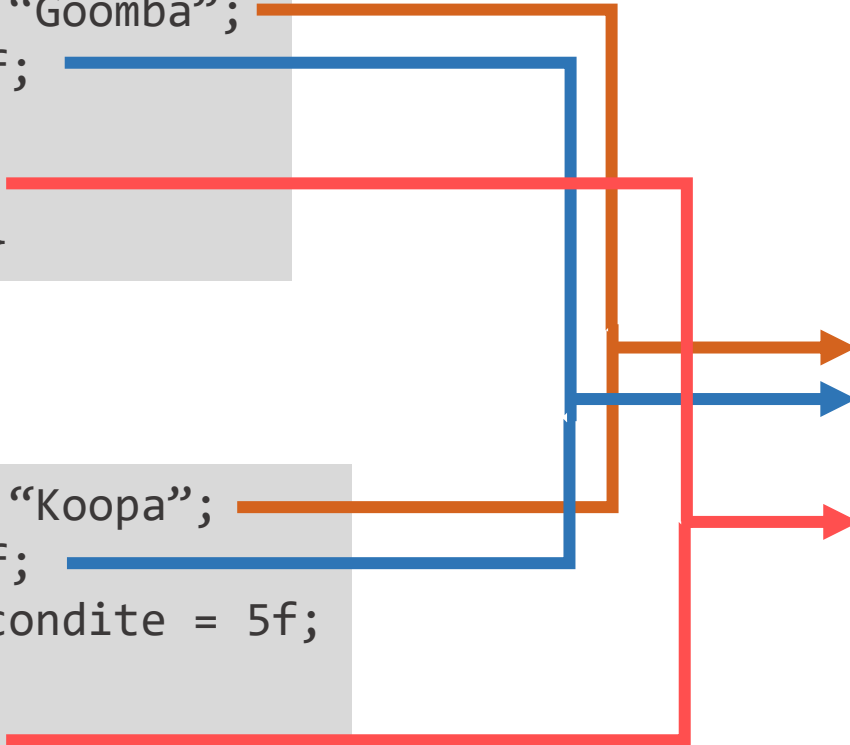
```
String nombre = "Koopa";  
float vida = 40f;  
float tiempo_escondite = 5f;  
  
void mover() {}  
void rodar() {}
```

Enemigo **generalizado**

**Enemigo.java**



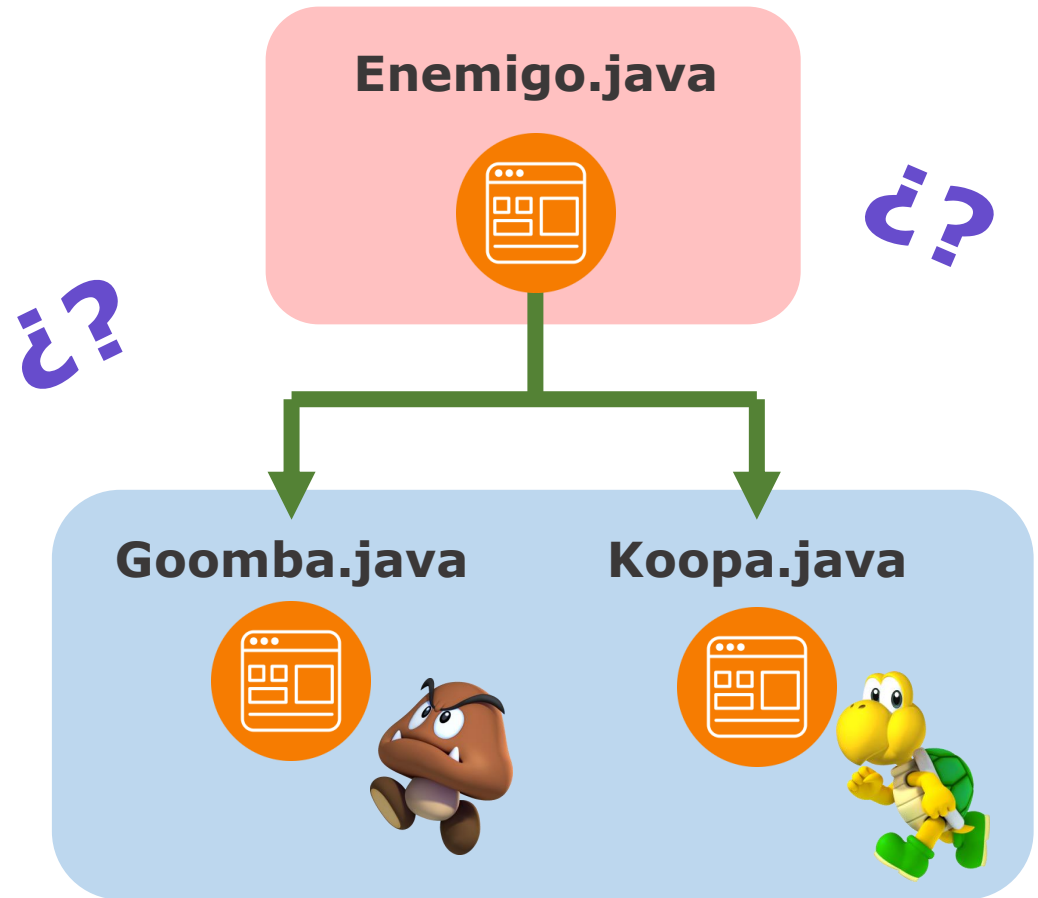
```
String nombre;  
float vida;  
  
void mover() {}
```



# Ahora...

Una vez identificados los elementos que comparten en común...

¿cómo crear esa relación entre la **clase general** con las **clases específicas**?







# Respuesta...

Con una palabra clave  
llamada **extends**

Goomba.java



```
public class Goomba extends Enemy {  
    [Código de Goomba.java]  
}
```

Koopa.java

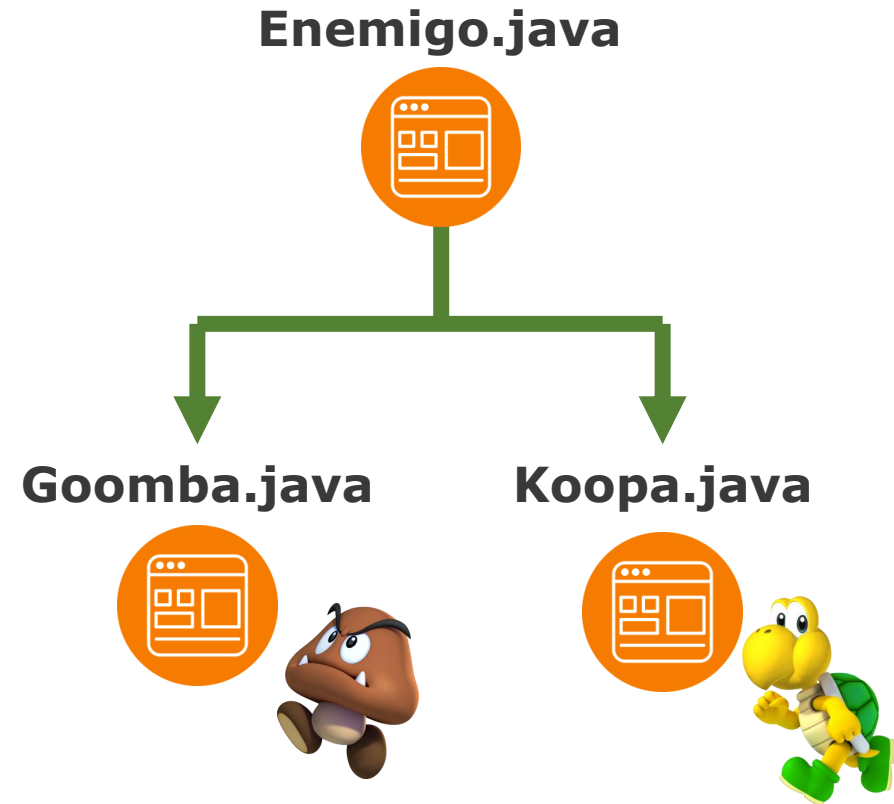


```
public class Koopa extends Enemy {  
    [Código de Koopa.java]  
}
```

# Extends

## Definición

**Palabra clave** de Java utilizada para **heredar** atributos y métodos de una clase padre a una o varias clases hijas.



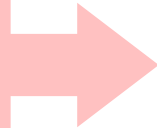
**¿Heredar?...**

Lo veremos en un momento...

# Volviendo al Ejemplo...

Tenemos a la **clase padre**...

**Enemigo.java**



```
public class Enemigo {  
  
    String nombre;  
    float vida;  
  
    void mover() {}  
}
```

# Volviendo al Ejemplo...

Tenemos a la **clase hija**...

Oye... ¡pero no tiene definidos sus atributos y método!

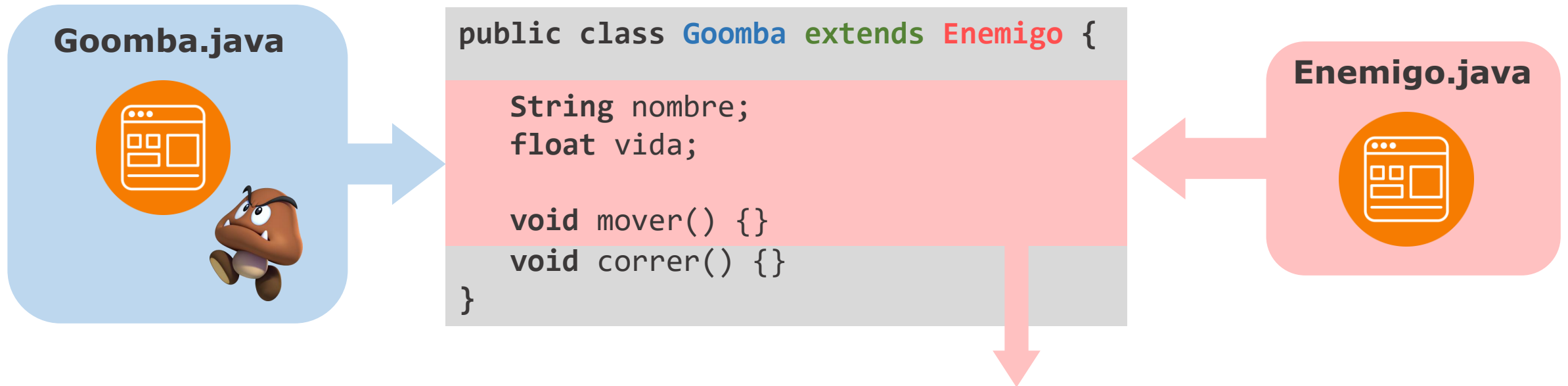
Goomba.java



```
public class Goomba extends Enemy {  
  
    void correr() {}  
}
```

# Volviendo al Ejemplo...

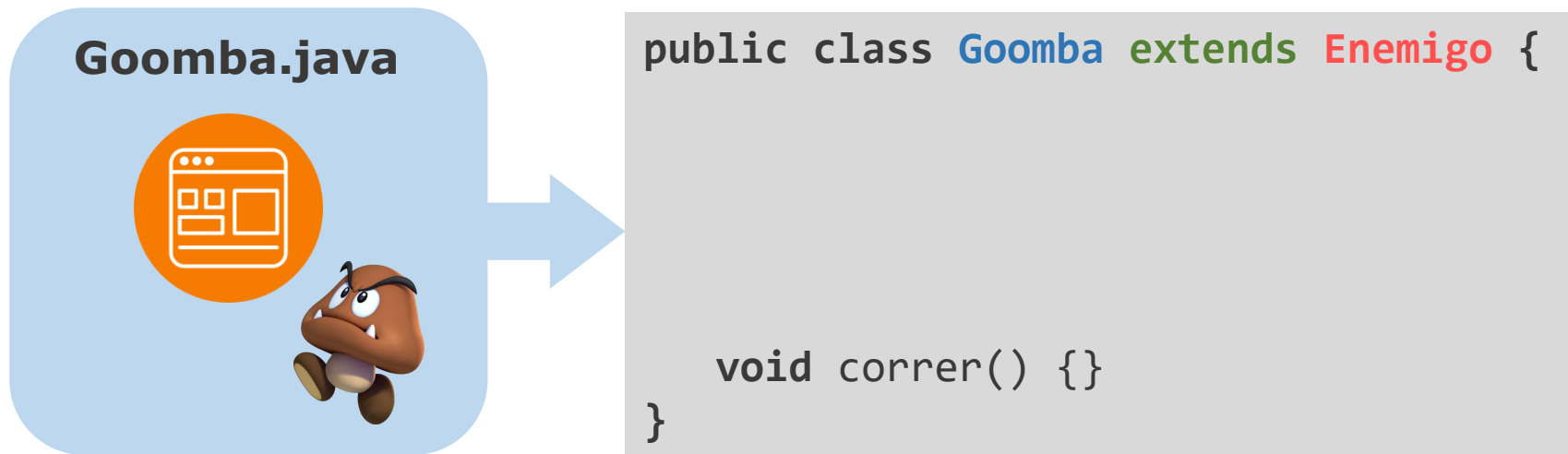
¡No hace falta! Los **hereda** de su clase **padre**



¡Importante! Esto ya **no** se escribe en **Goomba.java**

# Volviendo al Ejemplo...

Por lo tanto, la clase **hija** se queda así:



# Pregunta importante #1

Declarar las variables y funciones como **private**...

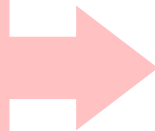


```
public class Enemigo {  
  
    private String nombre;  
    private float vida;  
  
    private void mover() {}  
}
```

...¿las heredar  hacia las clases hijas?

# Pregunta importante #1

Declarar las variables y funciones como **private**...



```
public class Enemigo {  
  
    private String nombre;  
    private float vida;  
  
    private void mover() {}  
}
```

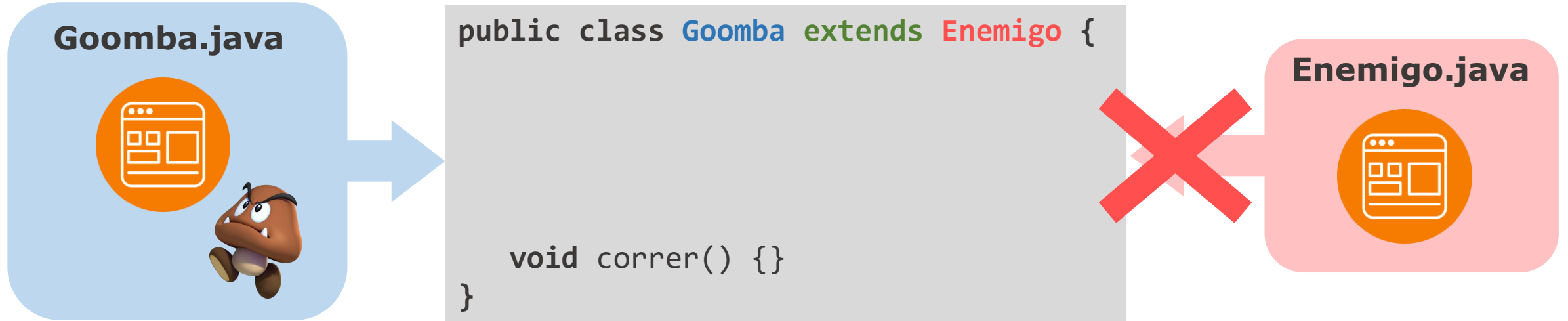
...¿las heredaré hacia las clases hijas?

**¡NO!**



# ¿Por qué no?

Al ser **private**, sólo las puede usar la clase **padre**.  
Por lo tanto... ¡NO HAY HERENCIA!



# Pregunta importante #2

```
System.out.println(vida);
```

>> ¿?

¿Qué valor imprime?

Enemigo.java



```
public class Enemigo {  
    float vida = 10f;  
}
```

Goomba.java



```
public class Goomba extends Enemigo {  
    float vida = 25f;  
}
```

# Respuesta

```
System.out.println(vida);
```

```
>> 25.0f
```

Imprime **25.0f**

Enemigo.java



```
public class Enemigo {  
    float vida = 10f;  
}
```

Goomba.java



```
public class Goomba extends Enemigo {  
    float vida = 25f;  
}
```

# ¿Cómo imprimir el atributo del padre?

```
System.out.println(vida);
```

>> ¿?

¿Se puede?

Enemigo.java



```
public class Enemigo {  
    float vida = 10f;  
}
```

Goomba.java



```
public class Goomba extends Enemigo {  
    float vida = 25f;  
}
```

# Respuesta

```
System.out.println(super.vida);
```

```
>> 10.0f
```

Imprime **10.0f**

Enemigo.java



```
public class Enemigo {  
    float vida = 10f;  
}
```

Goomba.java

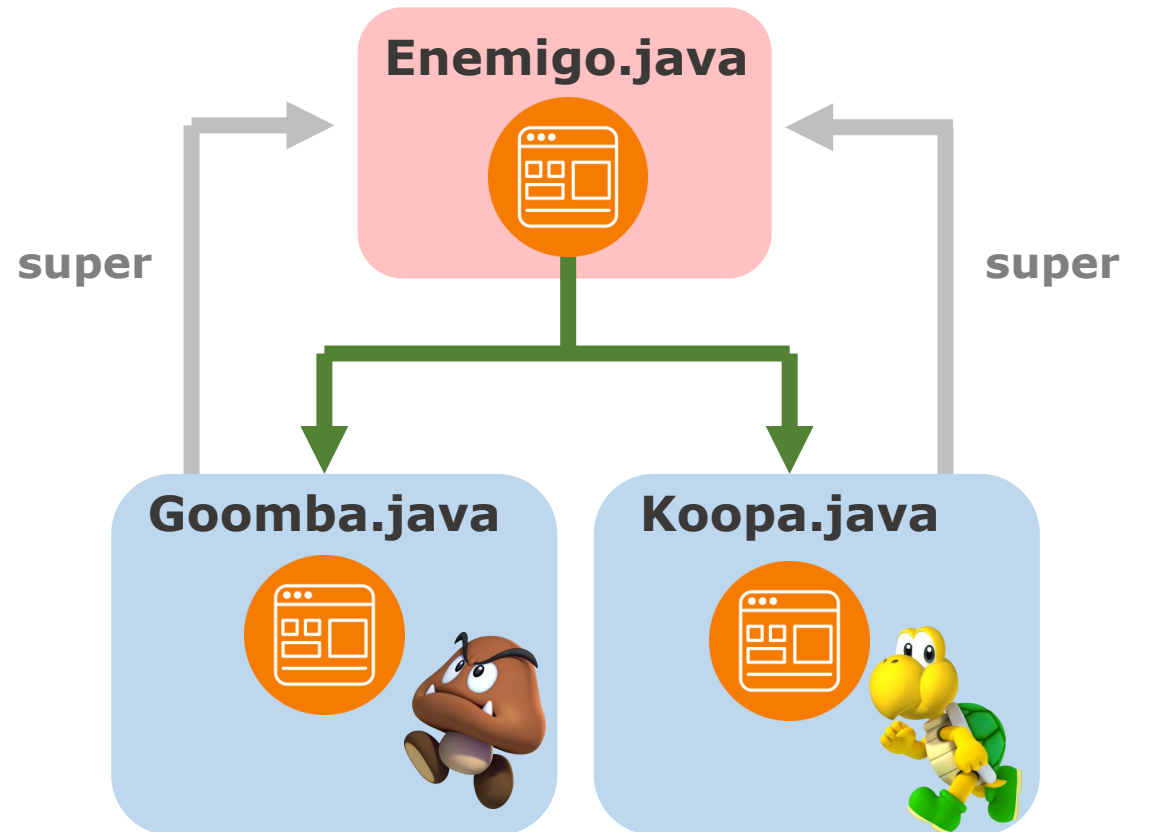


```
public class Goomba extends Enemigo {  
    float vida = 25f;  
}
```

# Super

## Definición

“**Palabra clave** de Java que hace **referencia** a la **clase padre**, sus atributos y métodos”.



Fuente:

[Java super Keyword \(w3schools.com\)](https://www.w3schools.com/java/java_super_keyword.asp)

# ¡¡ Mucho Cuidado !!

La palabra clave **super** funciona **exclusivamente** dentro del código de una **clase hija**.

Goomba.java



```
public class Goomba extends Enemy {  
  
    float vida = 25f;  
  
    Goomba(){  
        super();  
    }  
  
    void mover() {  
        super.mover();  
        print(super.vida);  
    }  
}
```

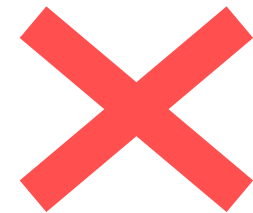
# ¡¡ Mucho Cuidado !!

Si se intenta utilizar en **otro código externo**, dará **ERROR**.

```
public class Principal {  
    public static void main(String[] args){  
        Goomba goomba = new Goomba();  
  
        // Estas líneas darán error  
        print(goomba.super.vida);  
        goomba.super.mover();  
    }  
}
```

>> javac Principal.java

Error de  
compilación





# En general...

Sintaxis para clases en jerarquía...



```
public class ClaseHija extends ClasePadre {...}
```



**Alcance de  
la clase**



**Nombre de la  
clase hija**

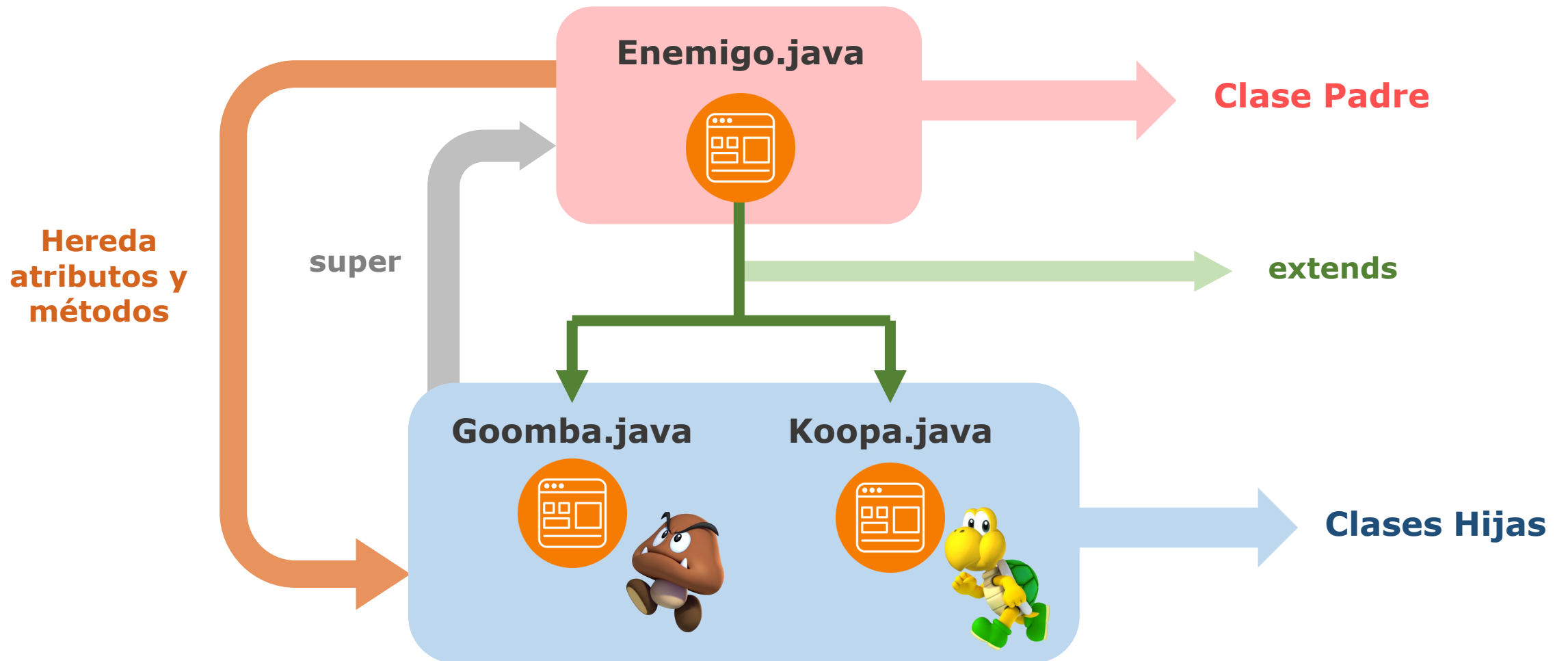


**Nombre de la  
clase padre**



**Bloque de  
código de la  
clase hija**

# Visualmente...



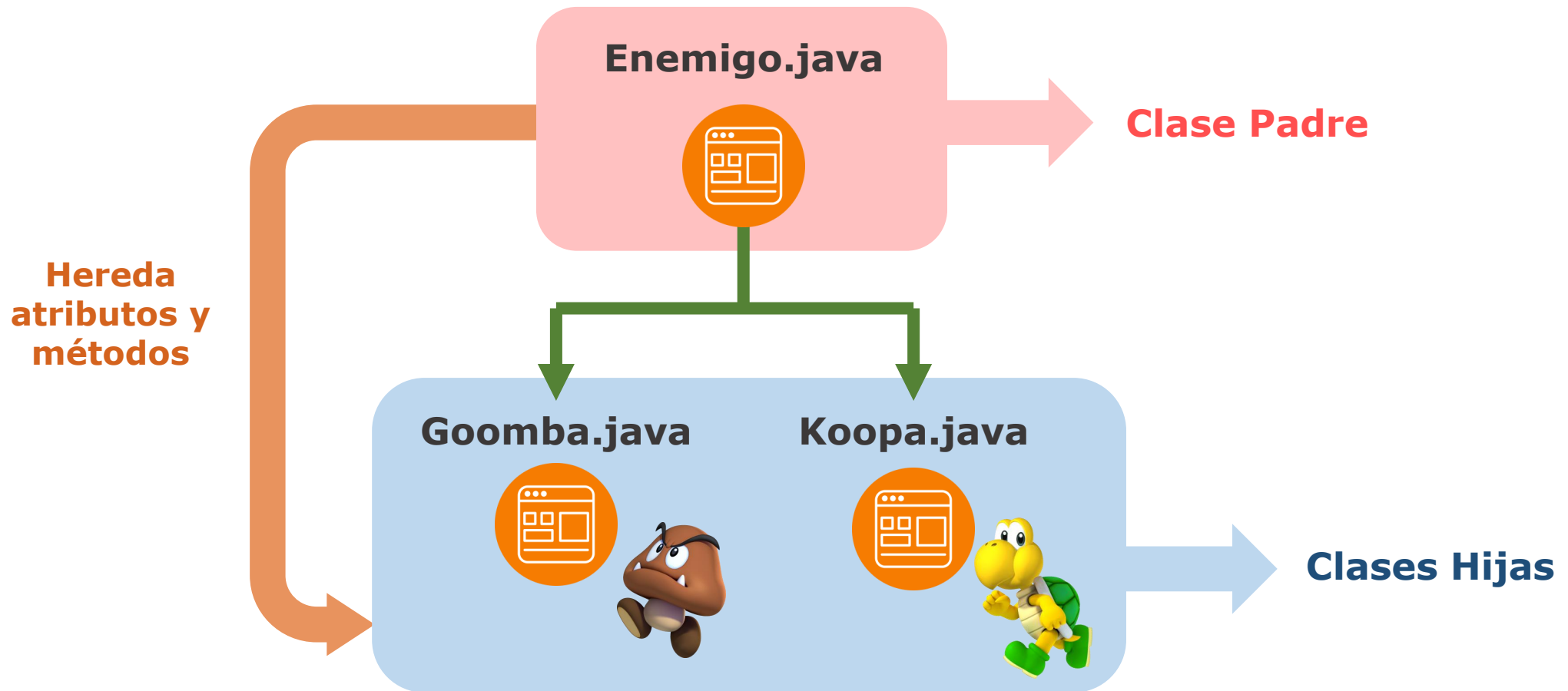
## Tema 5

# Polimorfismo

... oye ¿en verdad me  
parezco a mi padre?...

- **Tipado entre clases (padre-hijas)**
- *Casting* de clases
- **Tipos de Polimorfismo**

# Anteriormente...



# Supongamos que...

...quiero escribir este código:

```
Enemigo mi_enemigo = new Goomba();
```



## ¿Se puede?

Visualmente:



Enemigo mi\_enemigo;



Goomba



# Respuesta

...quiero escribir este código:

```
Enemigo mi_enemigo = new Goomba();
```

**Sí** se puede, pero  
¿por qué?

Visualmente:

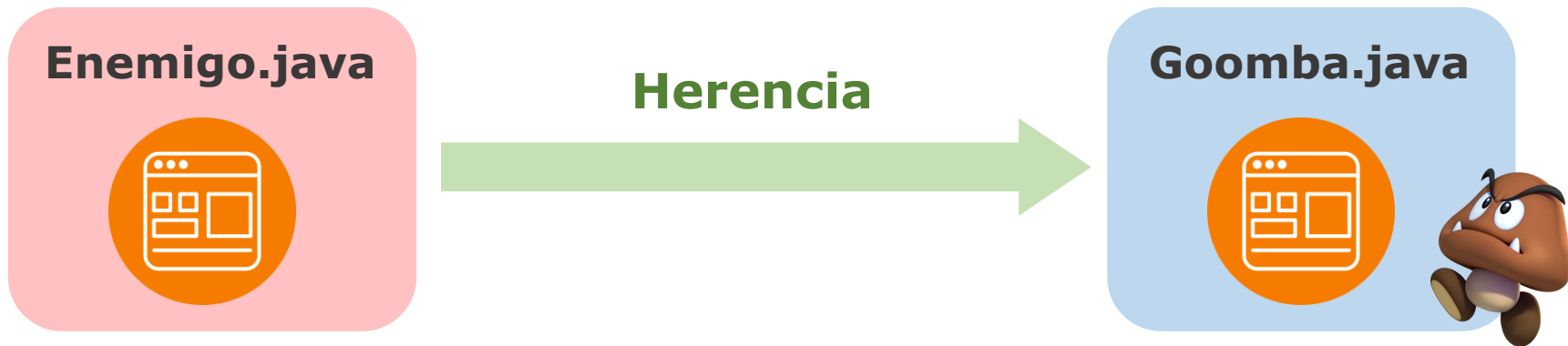
```
Enemigo mi_enemigo;
```



Goomba



# Recordemos que...

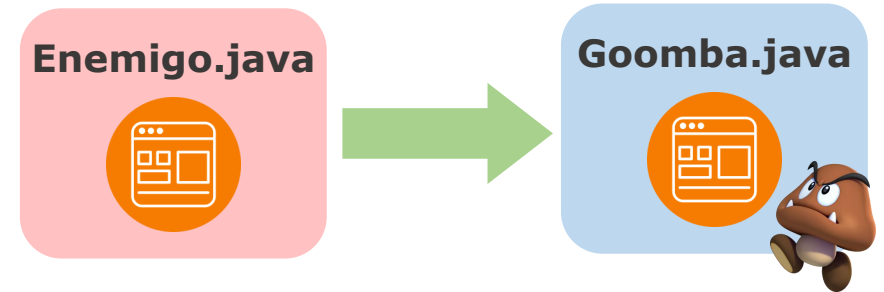


```
public class Enemyo {  
    [Bloque de código de Enemyo.java]  
}
```

```
public class Goomba extends Enemyo {  
    [Bloque de código de Goomba.java]  
}
```

# Observamos lo siguiente...

- **Enemigo** hereda sus atributos y métodos a la clase **Goomba**
- **Goomba** los hereda porque en la declaración de la clase lo especifica
- Vemos que **Goomba** se está “comportando” como un enemigo



**Goomba** extends **Enemigo**





# Por lo tanto...

Se puede concluir que...

Goomba extends Enemygo



**ii Goomba es un tipo de Enemygo !!**

...o dicho de otra manera...

“Goomba es, como tal, un Goomba,  
pero también es un Enemygo”

# Por lo tanto...

También se concluye que...

```
Enemigo mi_enemigo;  
  
mi_enemigo = new Goomba();
```

```
Enemigo mi_enemigo;
```



Goomba

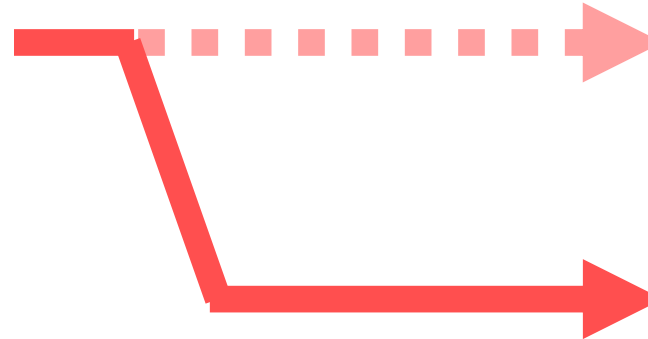


# Por lo tanto...

También se concluye que...

```
Enemigo mi_enemigo;  
  
mi_enemigo = new Goomba();  
  
mi_enemigo = new Koopa();
```

Enemigo mi\_enemigo;



Goomba



Koopa

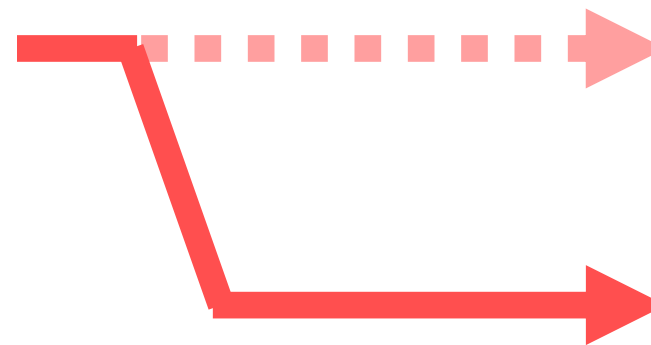


# Por lo tanto...

También se concluye que...

```
Enemigo mi_enemigo;  
  
mi_enemigo = new Goomba();  
  
mi_enemigo = new Koopa();
```

Enemigo mi\_enemigo;



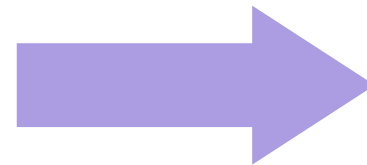
Goomba



Koopa



¡¡ Un **Enemigo** puede ser un **Goomba** o un **Koopa** !!



¡Polimorfismo!

# ¡Mucho OJO!

Lo que **NO** se puede hacer...

```
Goomba mi_goomba = new Koopa();
```

```
Koopa mi_koopa = new Goomba();
```

Goomba mi\_goomba;



Koopa



Koopa mi\_koopa;



Goomba



Si bien ambos son un tipo específico de **Enemigo**...

¡¡ **Goomba** **NO** es un tipo de **Koopa** !!

¡¡ **Koopa** **NO** es un tipo de **Goomba** !!

# Otro escenario...

Tenemos el siguiente código...

```
Goomba goomba = new Goomba();  
Enemigo enemigo = goomba;  
Goomba goomba_aux = enemigo;
```

Ya vimos que esto funciona:

Enemigo enemigo;



Goomba



¿Compilará?

# Otro escenario...

Tenemos el siguiente código...

```
Goomba goomba = new Goomba();  
Enemigo enemigo = goomba;  
Goomba goomba_aux = enemigo;
```

Ya vimos que esto funciona:

Enemigo enemigo;




Goomba



**NO**, porque la clase **Enemigo** no puede ser convertida “**implícitamente**” a la clase **Goomba**.

## Dicho de otra manera...



```
Enemigo enemigo = new Goomba();
```



Un **Goomba** es un **tipo** específico de **Enemigo**...



```
Goomba goomba = new Enemigo();
```



... pero un **Enemigo NO** es un **tipo** específico de **Goomba**.



# Detalles...

Tenemos el siguiente código...

```
Goomba goomba = new Goomba();  
Enemigo enemigo = goomba;  
Goomba goomba_aux = enemigo;
```

Ya vimos que esto funciona:

Enemigo enemigo;



Goomba



Pero, la referencia **enemigo** apunta a un objeto de tipo **Goomba**...

**¿hay una forma de acceder al objeto?**

# Detalles...

Tenemos el siguiente código...

```
Goomba goomba = new Goomba();  
Enemigo enemigo = goomba;  
Goomba goomba_aux = (Goomba) enemigo;
```

Ya vimos que esto funciona:

Enemigo enemigo;



Goomba

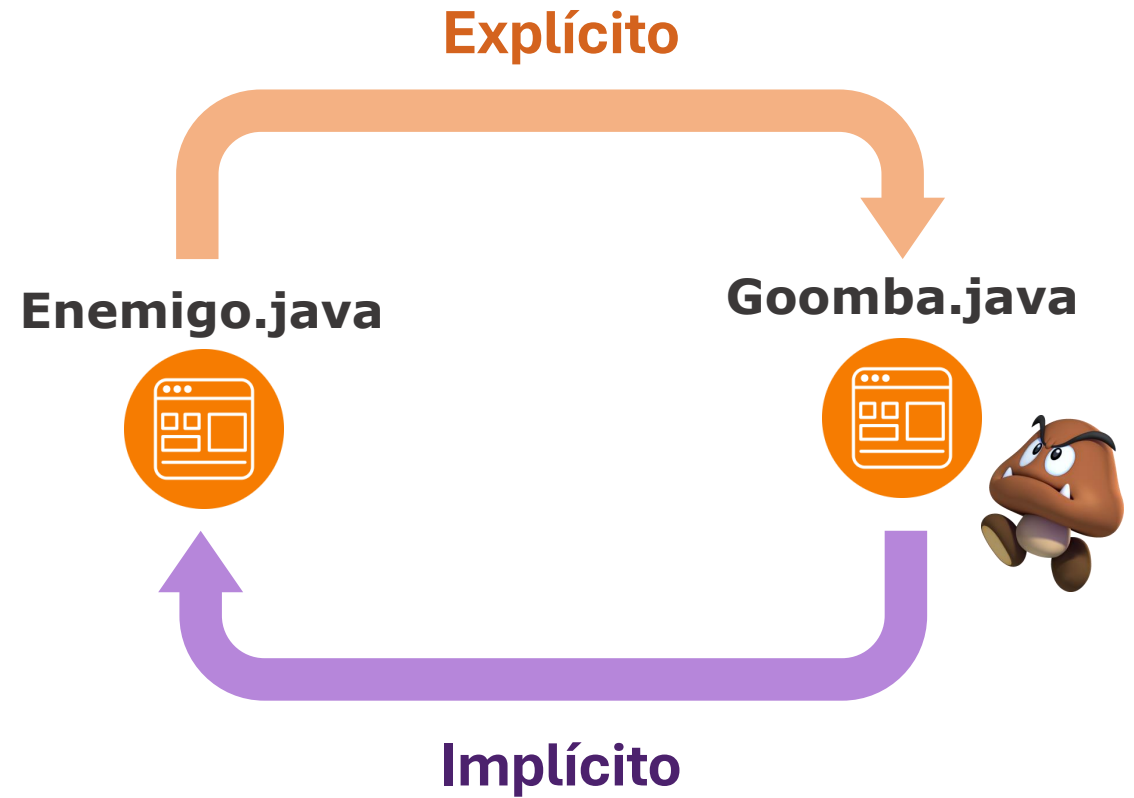


**Sí**, con algo llamado *Casting*.

# Casting

## Definición

“Proceso de **convertir** el valor de **un tipo de dato** (primitivo u objeto) a **otro tipo de dato**.”



Fuente:

[Java Type Casting \(With Examples\) \(programiz.com\)](https://programiz.com/java/type-casting/)

# Tipos de *Casting*

## Implícito



```
Enemigo enemigo = new Goomba();
```

## Explícito



```
Goomba goomba = (Goomba) enemigo;
```

**¡OJO!** La referencia **enemigo** debe estar apuntando a un **objeto** de tipo **Goomba** durante la **ejecución del programa**.

# Ejemplo #1

Si hacen esto...

```
Goomba goomba = (Goomba) new Enemigo();
```

Compilación: ¡sin problema! ✓

Ejecución: ¡dará error! ✗ ... pero, ¿por qué?

Porque **goomba** está **referenciando** a un **objeto** de tipo **Enemigo**, lo cual **NO es válido**.

## Ejemplo #2

Tenemos este otro código...

```
Enemigo enemigo = new Goomba();  
Koopa koopa = (Koopa) enemigo;
```

**Compilación:** ¡sin problema! ✓

**Ejecución:** ¡dará error! ✗

Porque **koopa** está **referenciando** a un **objeto** de tipo **Goomba**, lo cual **NO es válido**.

Nuevamente, **sí funciona**:

```
Enemigo enemigo;
```

**Goomba**



## Ejemplo #2

Visualmente, ocurre esto...

Koopa koopa;



Enemigo

(Koopa)



Enemigo enemigo;



Goomba



Equivale a...

Koopa koopa;

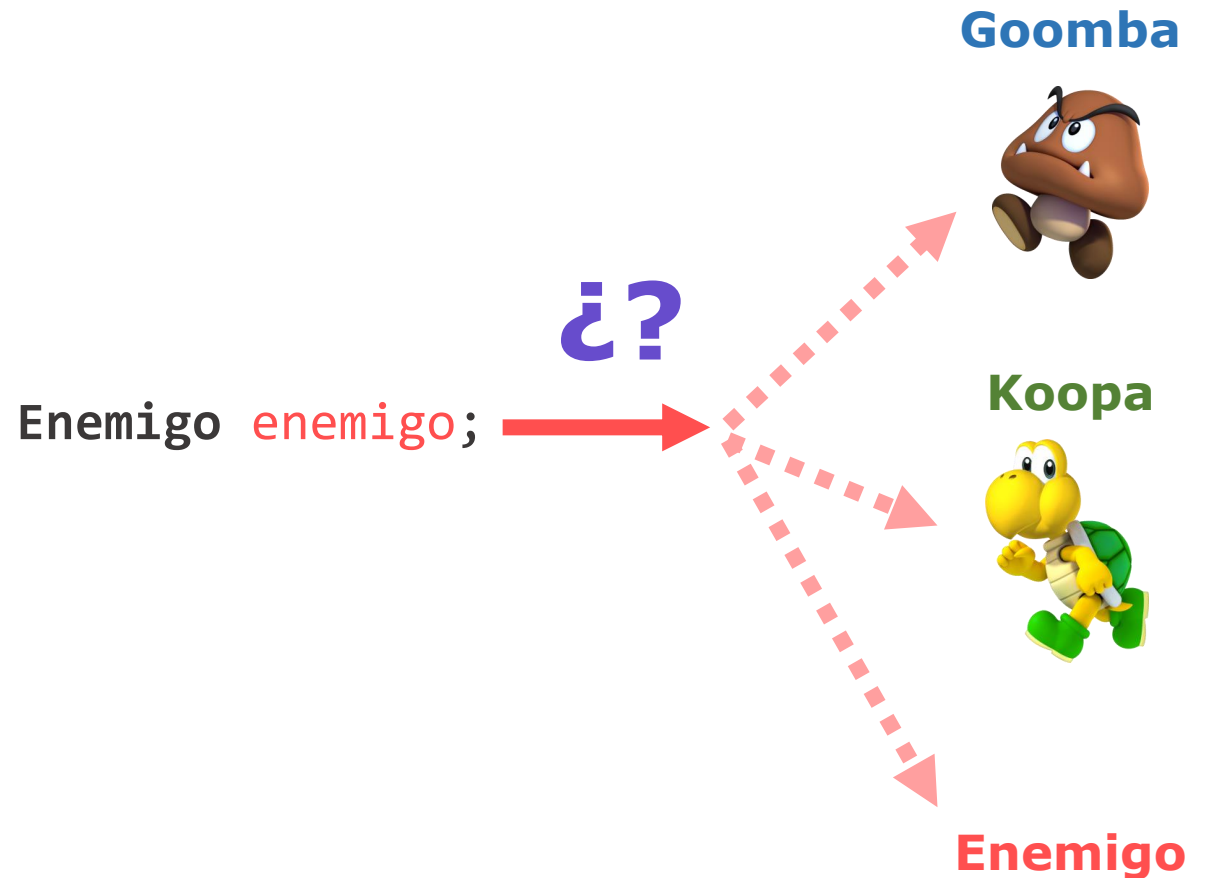


Goomba



# Prevención de errores en *Casting*

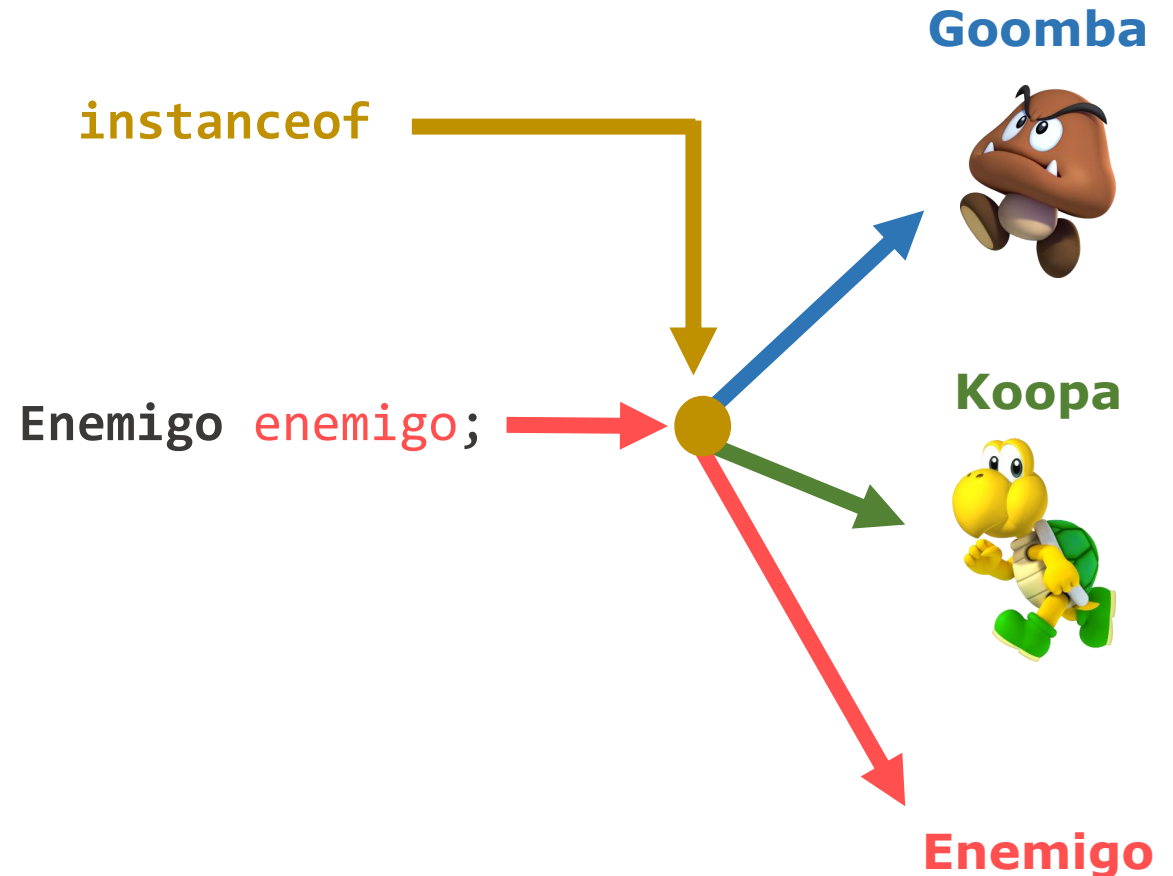
Durante la ejecución, el programa puede no **saber** a qué tipo de objeto está apuntando la referencia `enemigo`.





# Prevención de errores en *Casting*

Para ello, podemos ayudarle al programa declarando la palabra clave **instanceof**.



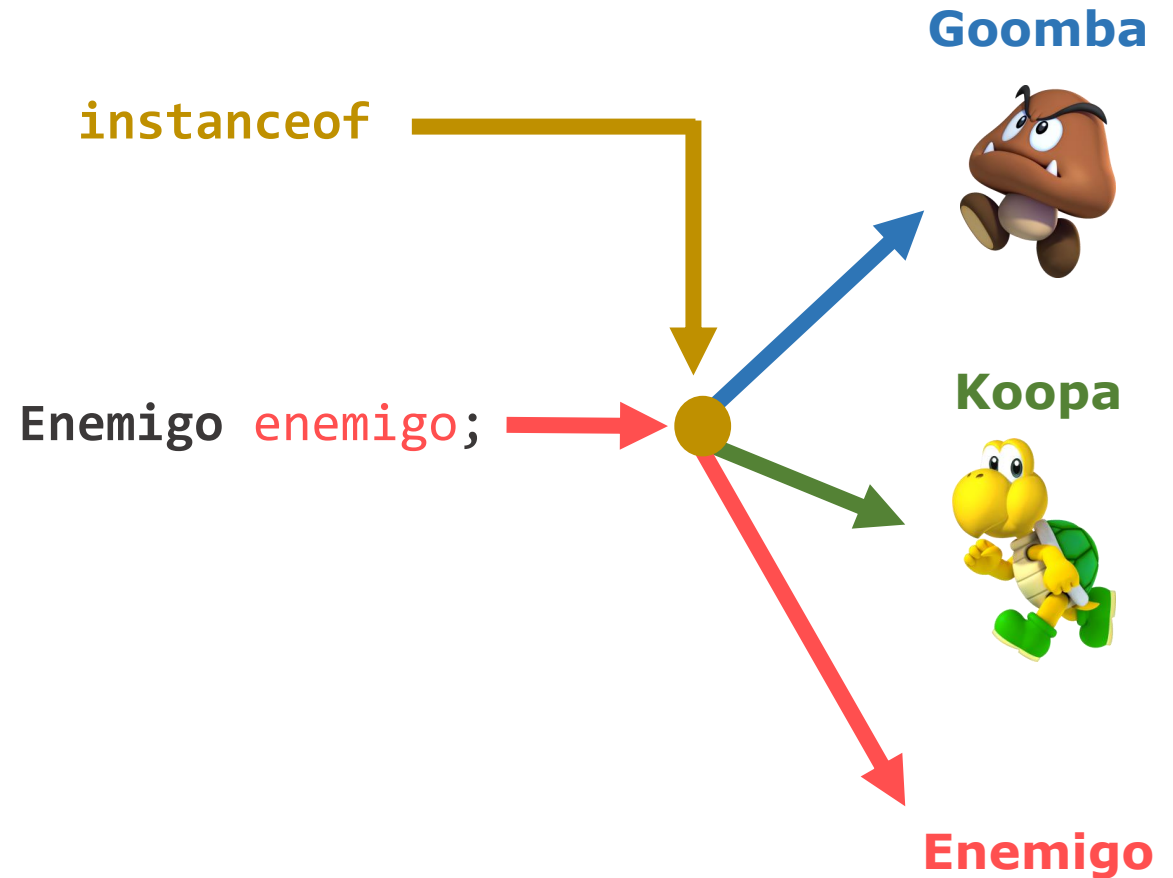
# InstanceOf

## Definición

“Palabra clave de Java que permite identificar si un **objeto** es una **instancia** de una **clase** **específica**”.

Fuente:

[Java instanceof Keyword \(w3schools.com\)](https://www.w3schools.com/java/java_instanceof_keyword.asp)



# Prevención de errores en *Casting*

```
void funcion(Enemy enemigo) {  
  
    // El enemigo es un Goomba  
    if (enemigo instanceof Goomba) {  
        Goomba goomba = (Goomba) enemigo;  
    }  
    // El enemigo es un Koopa  
    else if (enemigo instanceof Koopa) {  
        Koopa koopa = (Koopa) enemigo;  
    }  
    // El enemigo es un enemigo "genérico"  
    else {  
    }  
}
```

Enemy enemigo;



Goomba



Enemy enemigo;



Koopa



Enemy enemigo;



Enemy

# Interpretaciones de Instanceof

**enemigo instanceof Goomba**

La interpretación “obvia”, se puede leer como...

**¿El enemigo es una instancia de Goomba?**

Otra forma más fácil de comprender sería...

**¿El enemigo tiene la forma de un Goomba?**

O aún más fácil...

**¿El enemigo es un Goomba?**

# Tipos de Polimorfismo

En síntesis, existen **3 tipos de polimorfismo en POO**:

- Entre **Objetos** (lo acabamos de ver...)
- **Sobrecarga** de Métodos
- **Sobreescritura** de Métodos

# Sobrecarga de Métodos

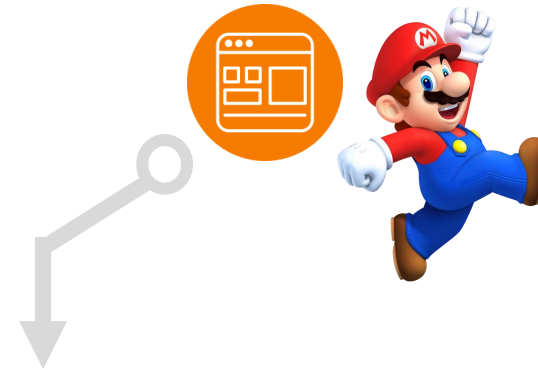
## Definición

“Capacidad de una clase de tener métodos con el **mismo nombre**, pero con **diferentes argumentos** (parámetros)”.

Fuente:

[Polimorfismo en Java | Programando Java](#)

## Jugador.java



```
void dialogar() {}  
  
void dialogar(String nombre) {}  
  
void dialogar(int vidas) {}  
  
void dialogar(String nombre, int vidas) {}
```

# Sobrecarga de Métodos

Tenemos el siguiente código...

**Jugador.java**



```
void dialogar() {  
    print("Hola");  
}  
  
void dialogar(String nombre) {  
    print("Hola, soy " + nombre);  
}  
  
void dialogar(int vidas) {  
    print("Hola, tengo " + vidas + " vidas");  
}  
  
void dialogar(String nombre, int vidas) {  
    print("Hola, soy " + nombre + " y tengo "  
        + vidas + " vidas");  
}
```

# Sobrecarga de Métodos

```
public static void main (String[] args) {  
    Jugador j = new Jugador();  
    j.dialogar();  
    j.dialogar("Mario");  
    j.dialogar(4);  
    j.dialogar("Luigi", 7);  
}
```

>> Hola

>> Hola, soy Mario

>> Hola, tengo 4 vidas

>> Hola, soy Luigi y  
tengo 7 vidas



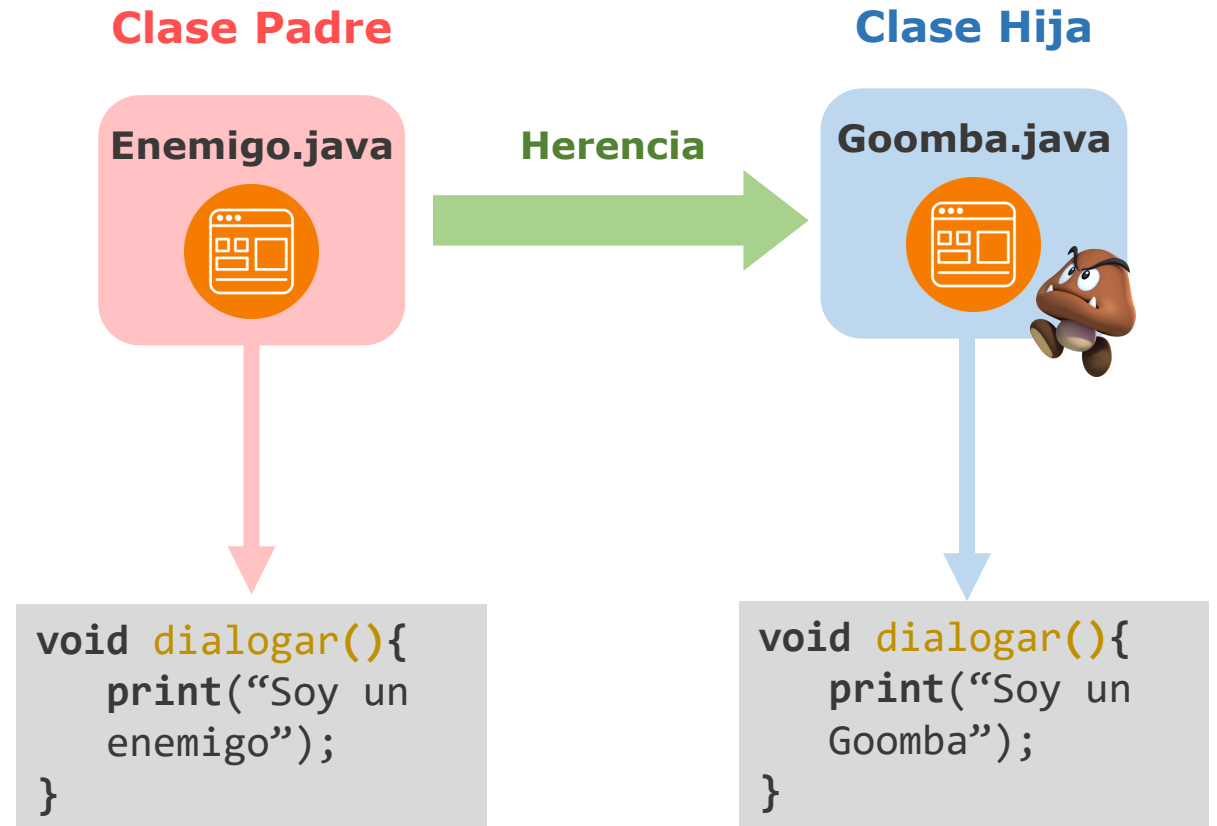
# Sobreescritura de Métodos

## Definición

“Capacidad de una clase **hija** de **redefinir** por completo un **método heredado** desde su clase **padre**”.

Fuente:

[Polimorfismo en Java: Programación orientada a objetos | IfgeekthenNTTdata](#)



# ¿Se acuerdan de esto?

```
Goomba goomba = new Goomba();  
System.out.println(goomba.super.vida);
```

>> 10.0f

Imprime 10.0f

Enemigo.java



```
public class Enemigo {  
    float vida = 10f;  
}
```

Goomba.java



```
public class Goomba extends Enemigo {  
    float vida = 25f;  
}
```

# ¿Y de esto?

```
Goomba goomba = new Goomba();  
System.out.println(goomba.vida);
```

>> 25.0f

Imprime 25.0f

Enemigo.java



```
public class Enemigo {  
    float vida = 10f;  
}
```

Goomba.java



```
public class Goomba extends Enemigo {  
    float vida = 25f;  
}
```

# Pues... resulta que...

Goomba está  
**sobreescribiendo** el  
atributo “vida” **heredado**  
desde **Enemigo**

Enemigo.java



```
public class Enemigo {  
    float vida = 10f;  
}
```

Goomba.java



```
public class Goomba extends Enemigo {  
    float vida = 25f;  
}
```

# Lo mismo ocurre con los métodos...

**Goomba** está  
**sobreescribiendo** el  
método “dialogar”  
heredado desde **Enemigo**

Enemigo.java



```
public class Enemigo {  
  
    void dialogar(){  
        print("Soy un enemigo");  
    }  
}
```

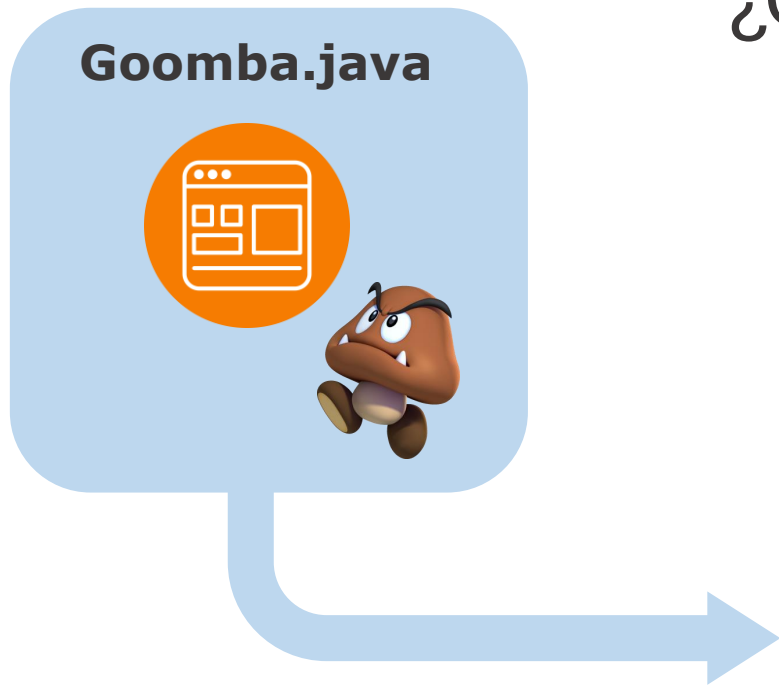
Goomba.java



```
public class Goomba extends Enemigo {  
  
    void dialogar(){  
        print("Soy un Goomba");  
    }  
}
```

# Pero...

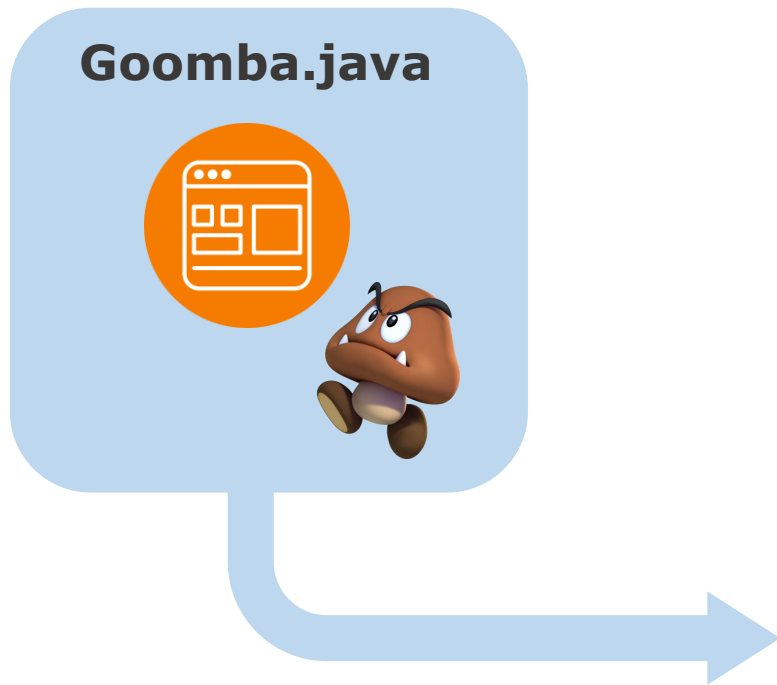
¿Cómo nos aseguramos que en verdad se esté **sobreescribiendo** el método?



```
public class Goomba extends Enemy {  
  
    void dialogar(){  
        print("Soy un Goomba");  
    }  
}
```

# Respuesta

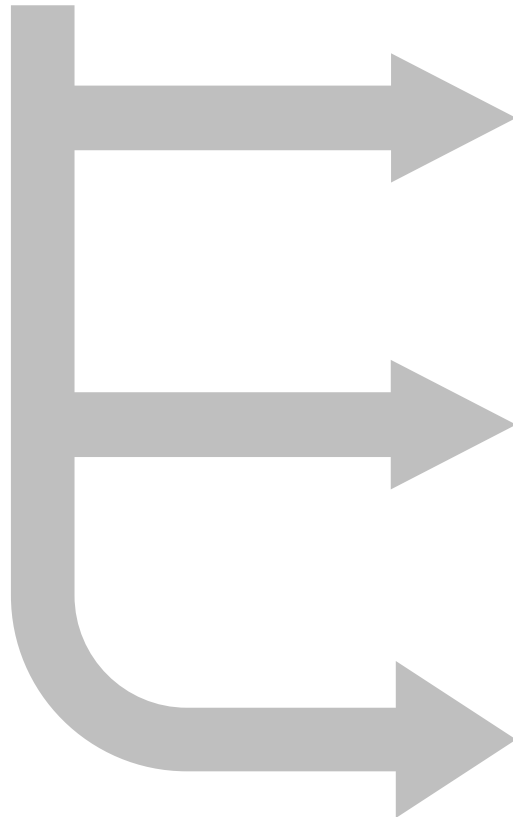
Con una anotación llamada  
**@Override**



```
public class Goomba extends Enemy {  
  
    @Override  
    void dialogar(){  
        print("Soy un Goomba");  
    }  
}
```

# Como concepto...

## @Override



**Anotación** que le indica al **compilador de Java** que una **función heredada** será **sobreescrita**.

El nombre de la función de la **clase hija** debe **coincidir** (tanto en **nombre** como en **argumentos**) con la función de la **clase padre**.

Si **no hay coincidencia**, al momento de compilar generará **error**.

Fuente:

[¿Para que sirve la línea @Override en java? - Stack Overflow en español](#)



# Ejemplo #1

## Compila

La función de la clase  
**hija coincide** con la de  
la clase **padre**

Enemigo.java



```
public class Enemigo {  
  
    void dialogar(){  
        print("Soy un enemigo");  
    }  
}
```

Goomba.java



```
public class Goomba extends Enemigo {  
  
    @Override  
    void dialogar(){  
        print("Soy un Goomba");  
    }  
}
```

# Ejemplo #2

## Da error

La función de la clase  
**hija NO coincide** con la  
de la clase **padre**

Enemigo.java



```
public class Enemigo {  
  
    void dialogar(){  
        print("Soy un enemigo");  
    }  
}
```

Goomba.java



```
public class Goomba extends Enemigo {  
  
    @Override  
    void Dialogar(String nombre){  
        print("Soy un Goomba");  
    }  
}
```

# Sobreescritura de Métodos

```
public static void main (String[] args) {  
  
    Enemy enemy = new Enemy();  
    enemy.dialogar();  
  
    Enemy goomba = new Goomba();  
    goomba.dialogar();  
  
}
```

>> Soy un **enemigo**

>> Soy un **Goomba**

## Tema 6

# Paquetes

... carpetas, carpetas, y más **carpetas**... ¡Ah! Y más **clases**...

- **Creación de un Paquete en Java**
- **Importar un Paquete**
- **Encapsulamiento en Paquetes**

# Paquete

## Definición para Java

**Conjunto de clases**  
(generalmente **relacionadas** entre sí) cuyo propósito es facilitar la **organización del código** de un proyecto de software.



**Personajes**

**Jugador.class**



**Enemigo.class**



**Armas**

**Rifle.class**



**Items**

**Consumible.class**



# ¿Cómo creamos un paquete?

Con la siguiente línea de código...

```
package mipaquete.misubpaquete;
```



**Palabra reservada  
para declarar  
paquetes**



**Nombre de  
nuestro paquete**



**Nombre de  
nuestros  
subpaquetes**

# ¿Cómo creamos un paquete?

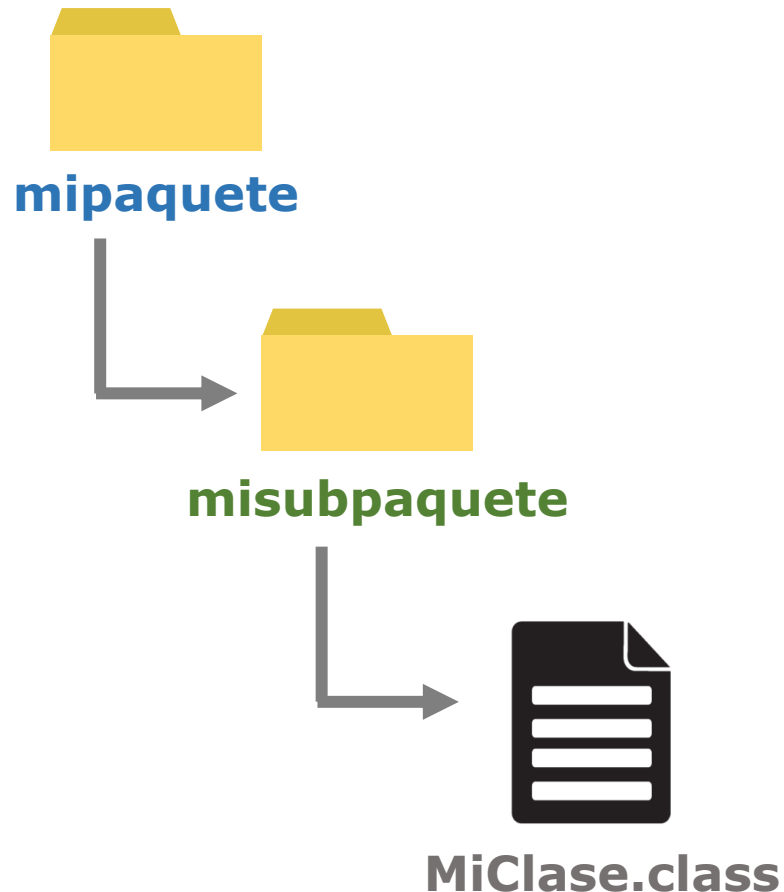
Dentro de una clase...

```
package mipaquete.misubpaquete;  
  
public class MiClase {  
  
    // Atributos  
    // Constructores  
    // Métodos  
}
```

## Observaciones

- La línea de código va **antes de definir una clase**
- Cada paquete se separa con **punto**
- Los nombres de paquetes deben estar escritos en **minúsculas**
- Cada nombre de paquete representa una **carpeta** para la jerarquía de directorios

# Visualmente... en nuestros archivos...



## ¡Importante!

El paquete contiene únicamente los **archivos ejecutables** (.class)

Nosotros **NO** debemos crear las carpetas que conforman al paquete

Para ello se encarga el comando **javac**, pero con otras características



# ¿Cómo se genera el paquete?

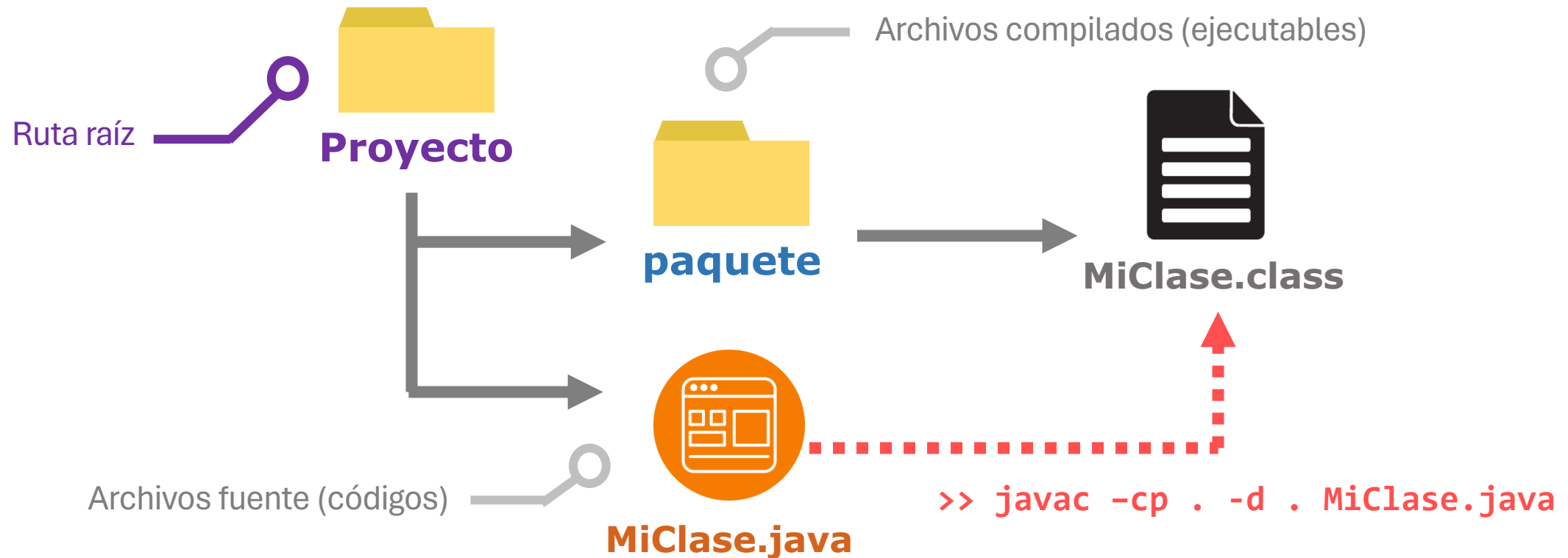
Con el siguiente comando...

```
>> javac -cp ruta_raíz -d ruta_raíz MiClase.java
```

ruta_raíz	Ruta raíz desde donde se creará o está todo nuestro paquete
-cp	Permite a la Terminal <b>obtener los archivos .class</b> que ya se encuentren en el paquete en caso de que el código actual los requiera para poder compilarse.
-d	<b>Crea nuestro paquete</b> (en caso de que no lo haya hecho). <b>Genera el archivo .class</b> del código actual en la carpeta especificada por la palabra <b>package</b>
MiClase.java	<b>Código a compilar</b> y del cual se obtiene la estructura del paquete a generar

# Generar paquete en el directorio actual

```
>> javac -cp . -d . MiClase.java
```



# ¿Cómo ejecutar un programa en un paquete?

Con este otro comando...

```
>> java -cp ruta_raíz paquete.subpaquete.MiClase
```

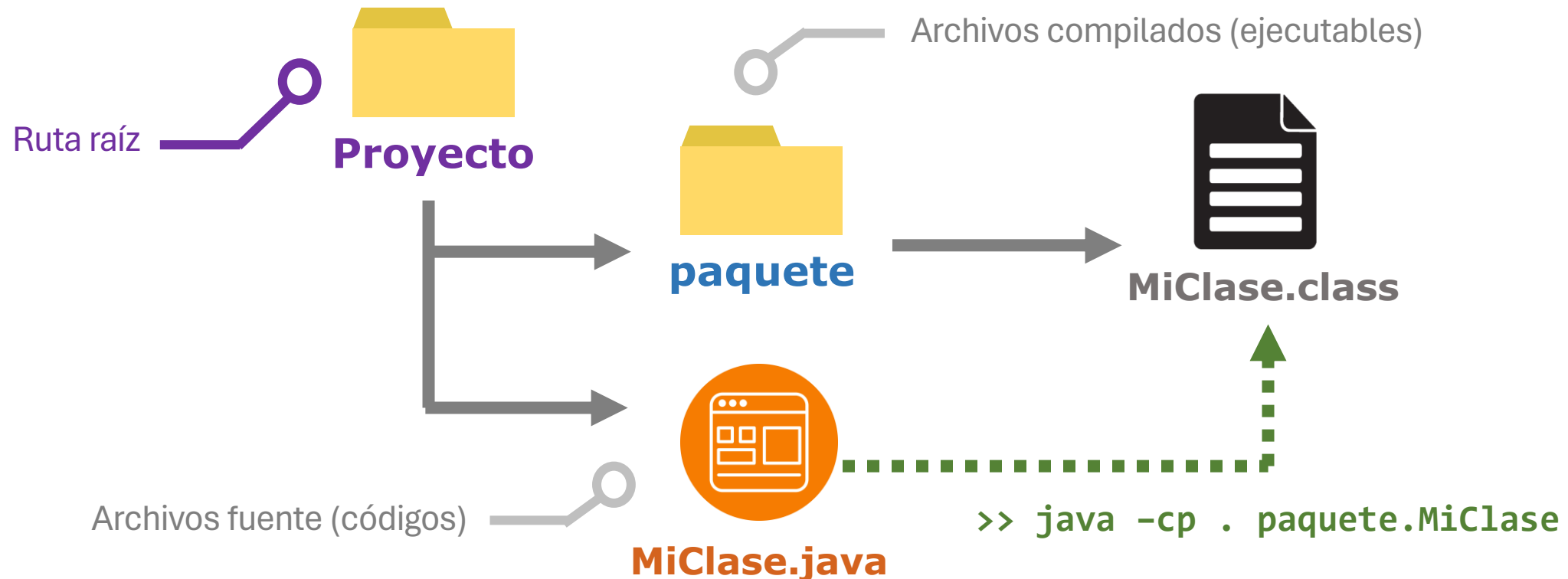
Indica a Java que el **paquete** de nuestro ejecutable se encuentra en la **ruta especificada**.

Programa a ejecutar.

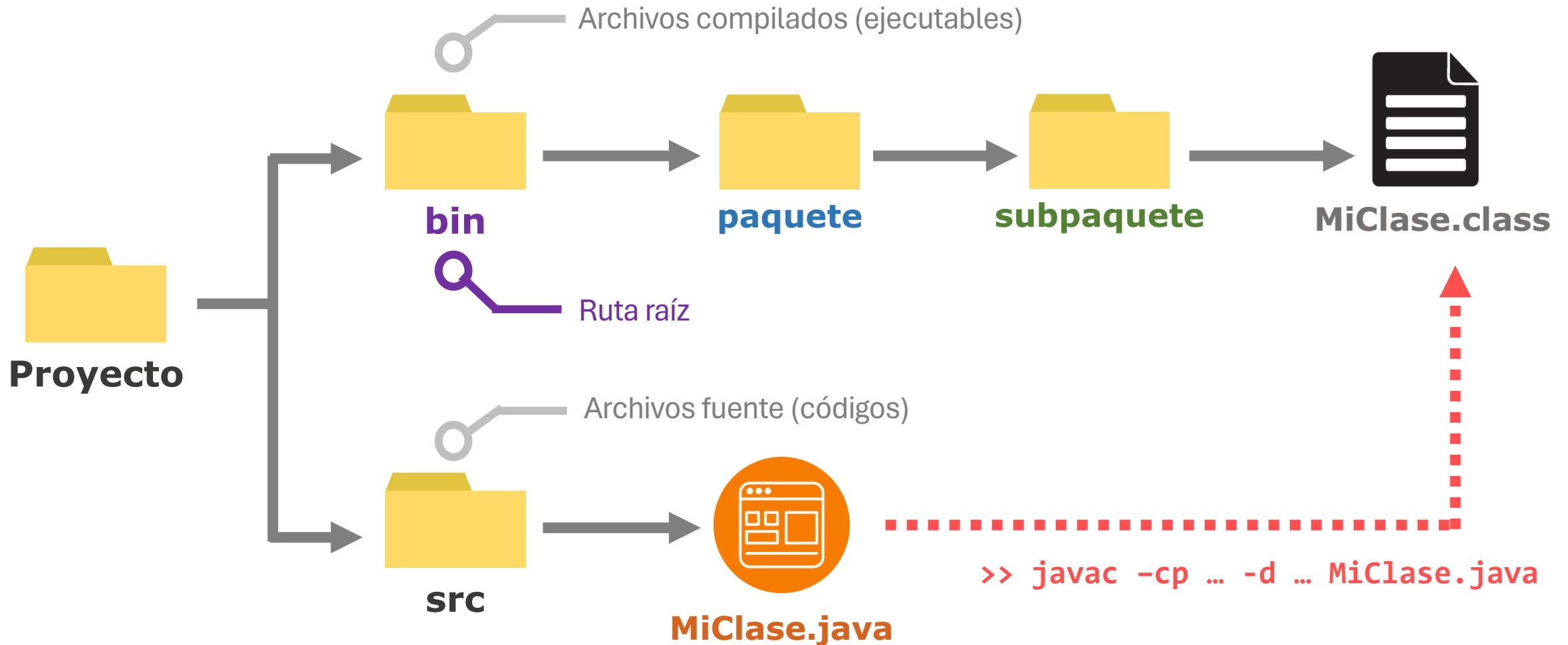
Debe especificarse en qué posición se encuentra el ejecutable dentro de la jerarquía del paquete, o de lo contrario dará **error**.

# Ejecutar un programa en el directorio actual

```
>> java -cp . paquete.MiClase
```



# Recomendación para crear paquetes



# Compilando el ejemplo previo...

La Terminal se encuentra actualmente en la carpeta **src** del Proyecto. Es ahí donde está el código **MiClase.java**

Desktop/Proyecto/src >>

```
javac -cp ../bin -d ../bin MiClase.java
```

Se accede a **otros ejecutables** del paquete por si **MiClase.java** los requiere

Se indica a Java que se va a crear el paquete en la carpeta **bin** del Proyecto

Clase a compilar. Esta misma indica a qué paquete pertenece

# Ejecutando el ejemplo previo...

La Terminal ahora se encuentra en la carpeta **Proyecto**.

Desktop/Proyecto >>

```
java -cp bin paquete.subpaquete.MiClase
```

Indica a Java que el **paquete** de nuestro ejecutable se encuentra en la **ruta especificada**.

El programa a ejecutar se encuentra dentro de las carpetas paquete/subpaquete

# Importar Paquetes

¿Qué pasa si necesitamos utilizar una **clase** que se encuentra en **otro paquete**?

Usamos la siguiente línea de código...

```
import mipaquete.misubpaquete.MiClase;
```



**Palabra reservada  
para importar  
paquetes**



**Nombre completo  
de nuestro paquete**



**Nombre de la clase  
que necesitamos en  
nuestro código**



# Importar Paquetes

Dentro de una clase...

```
package mipaquete.misubpaquete;

import mipaquete2.ClaseExterna;

public class MiClase {

    // Atributos
    ClaseExterna clase_ext;
}
```

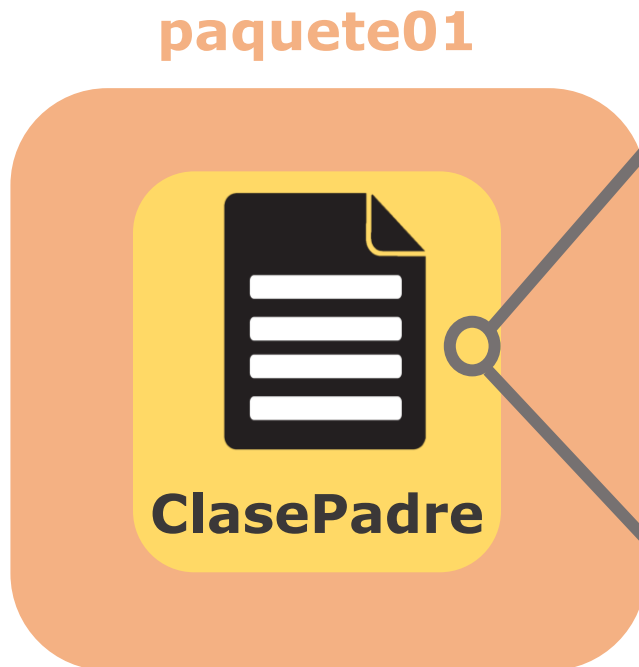
## Observaciones

- La línea de código va **antes de definir una clase y después de definir un paquete**
- Cada paquete se separa con **punto**
- Hasta el último subpaquete, se debe especificar el **nombre de la clase** a importar
- **Consejo:** si se tienen varias clases en un mismo paquete, para **importar todas** se puede declarar la línea como:

```
import mipaquete2.*;
```

# Encapsulamiento en Paquetes

Supongamos el siguiente escenario inicial...



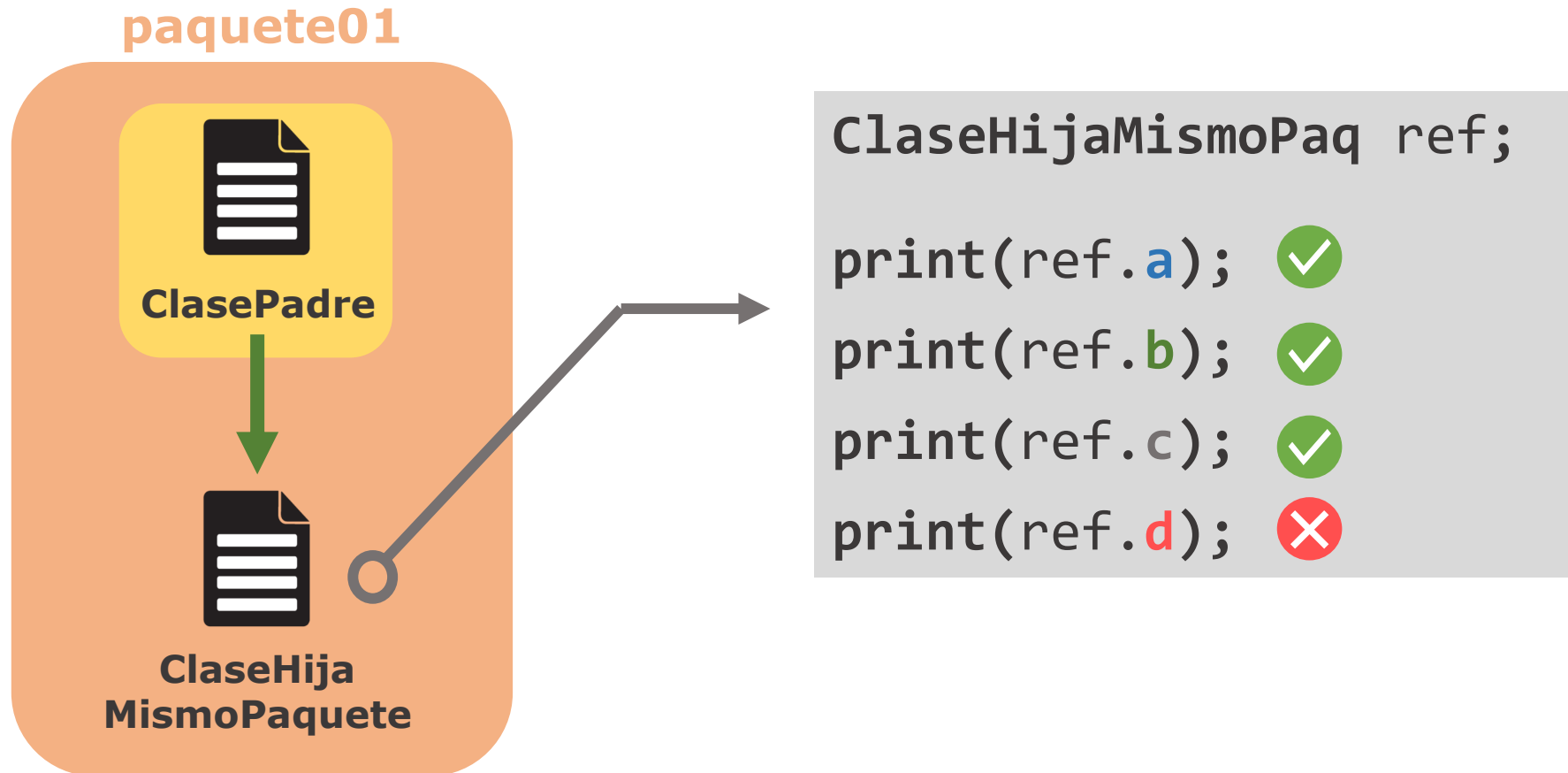
```
public int a;  
protected int b;  
int c;  
private int d;
```

```
ClasePadre ref;  
  
print(ref.a); ✓  
print(ref.b); ✓  
print(ref.c); ✓  
print(ref.d); ✓
```

Volviendo a  
aspectos de  
seguridad...

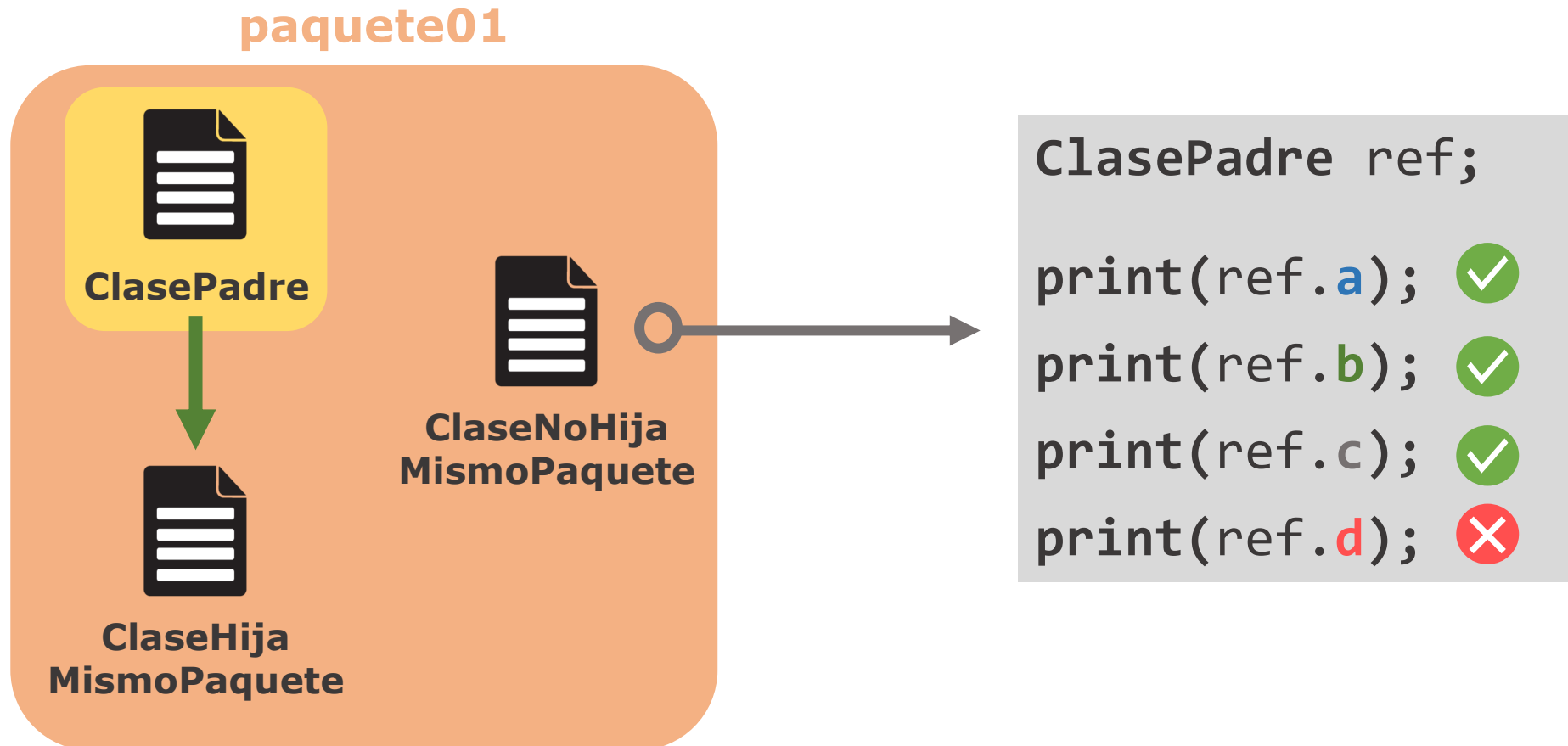
# Encapsulamiento en Paquetes

Ahora agregamos una clase hija en el mismo paquete...



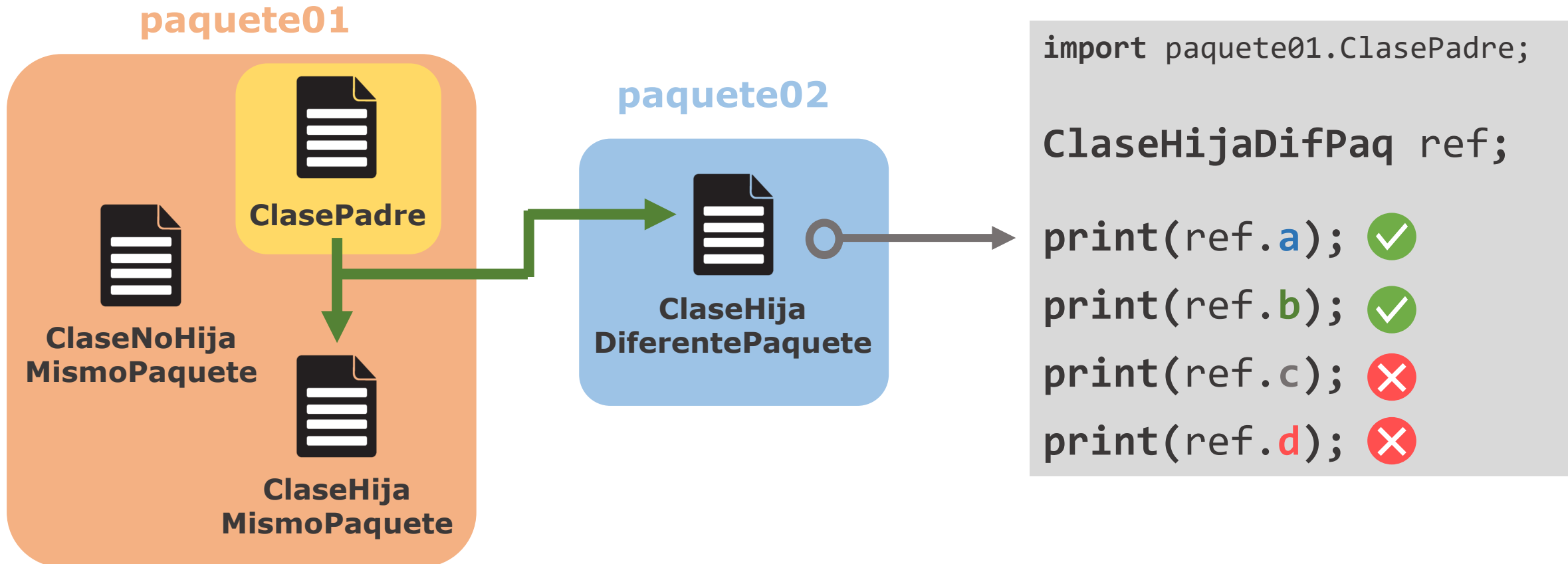
# Encapsulamiento en Paquetes

Luego, agregamos una clase NO hija en el mismo paquete...



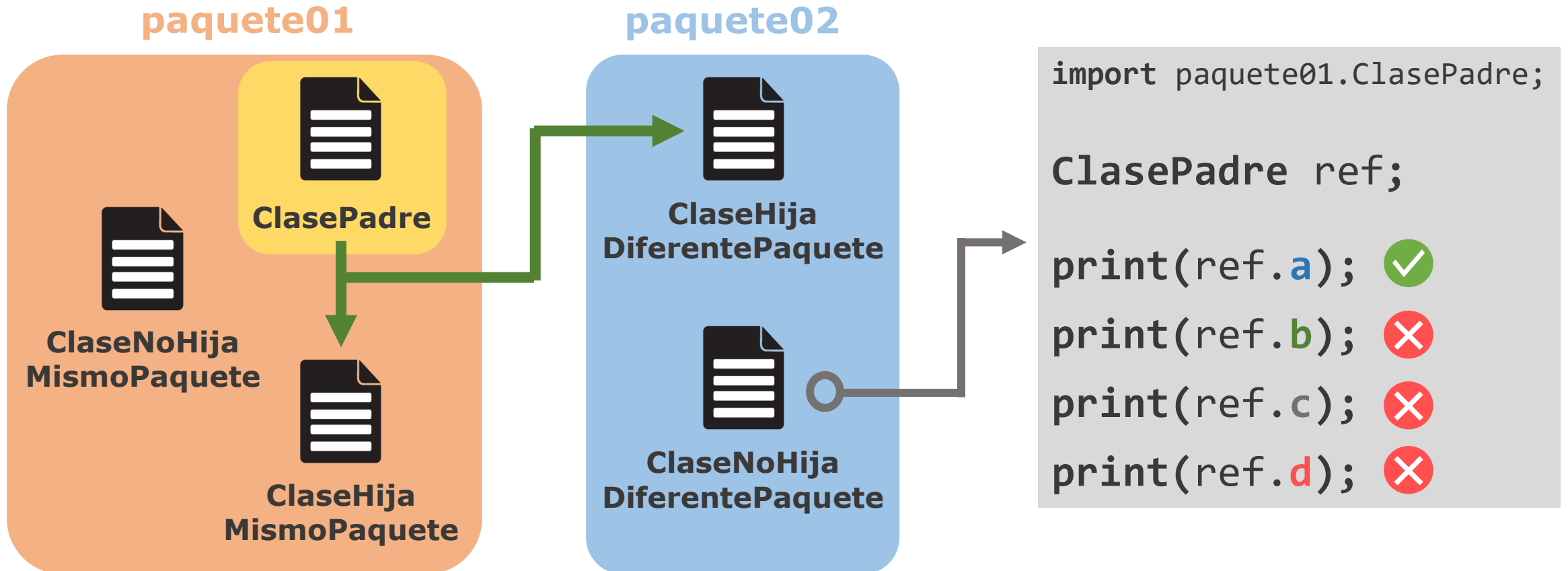
# Encapsulamiento en Paquetes

Después, creamos un nuevo paquete y una nueva clase hija...



# Encapsulamiento en Paquetes

Finalmente, agregamos una clase NO hija en el otro paquete...



# En conclusión...

	public	protected	(sin modificador)	private
Una clase a ella misma	✓	✓	✓	✓
Una clase hija en el mismo paquete	✓	✓	✓	✗
Una clase NO hija en el mismo paquete	✓	✓	✓	✗
Una clase hija en otro paquete	✓	✓	✗	✗
Una clase NO hija en otro paquete	✓	✗	✗	✗

**¡Eso es todo amigos!  
Gracias**

