
Introducción a la POO

Programación III

Agenda

- ¿Qué es paradigma?
- Paradigma Orientado a Objetos.
- Lenguaje de Programación Orientado a Objetos.
- Programación Orientada a Objetos:
 - Abstracción
 - Encapsulamiento
 - Herencia
 - Polimorfismo



Paradigma

Paradigma: Forma de entender y representar la realidad.

Principales paradigmas de programación:

- Paradigma Funcional.
- Paradigma Lógico.
- Paradigma Imperativo o Procedural.
- **Paradigma Orientado a Objetos.**



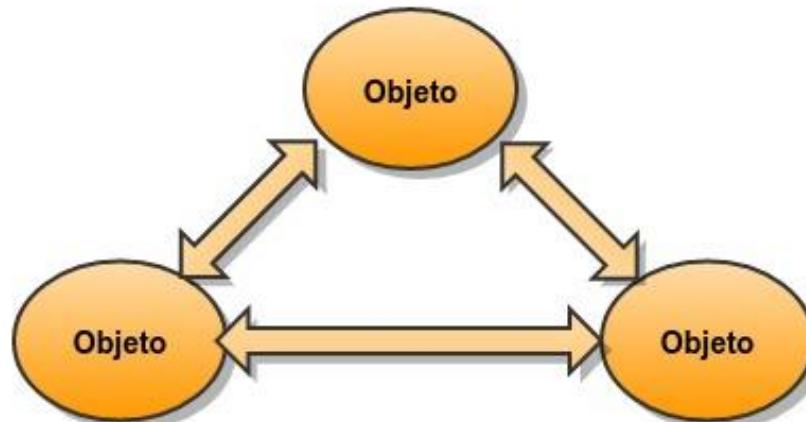
Agenda

- ~~¿Qué es paradigma?~~
- **Paradigma Orientado a Objetos**
- Lenguaje de Programación Orientado a Objetos.
- Programación Orientada a Objetos:
 - Abstracción
 - Encapsulamiento
 - Herencia
 - Polimorfismo



Paradigma Orientado a Objetos (1)

- Metodología de desarrollo de aplicaciones en la cual éstas se organizan como colecciones de **cooperativas** de **objetos**, cada uno de los cuales representan una instancia de alguna clase.



Paradigma Orientado a Objetos (2)

Problema: Pedro va a la florería de Juan, compra un ramo para su novia y detalla la dirección de recepción.

Mecanismo para resolver un problema:

- **Agente** → Juan (dueño de la florería)
- Enviar **mensaje** → Enviar flores a la novia de Pedro
- Es la **responsabilidad** de Juan que la novia de Pedro reciba el ramo de flores → **Método** para realizar la tarea.

Agenda

- ~~¿Qué es paradigma?~~
- ~~Paradigma Orientado a Objetos~~
- **Lenguaje de Programación Orientado a Objetos**
- Programación Orientada a Objetos:
 - Abstracción
 - Encapsulamiento
 - Herencia
 - Polimorfismo



Lenguaje de Programación OO

No basta un lenguaje OO para programar orientado a objetos, para eso hay que seguir un paradigma orientado a objetos.

- Se llama así a cualquier lenguaje de programación que implemente los **conceptos** definidos en la **programación orientada a objetos**.

Ejemplos: C++, C#, PHP, **Java**.



Programación Orientada a Objetos (POO)



Agenda

- ~~¿Qué es paradigma?~~
- ~~Paradigma Orientado a Objetos~~
- ~~Lenguaje de Programación Orientado a Objetos~~
- Programación Orientada a Objetos:
 - Abstracción
 - Encapsulamiento
 - Herencia
 - Polimorfismo



Abstracción

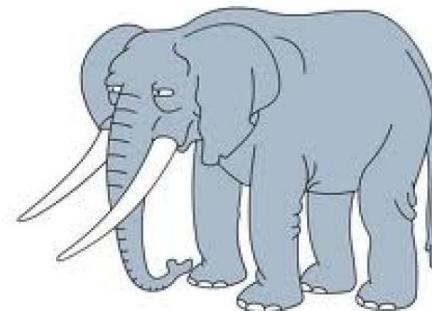
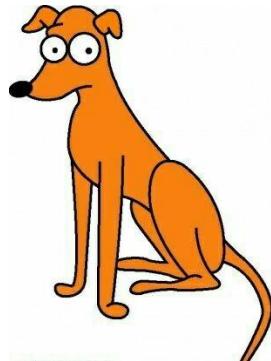
- Consiste en **aislar** un elemento de su contexto → ¿Qué hace?
- Se enfoca en la visión externa de un objeto → Separar el comportamiento específico.
- Quitar las propiedades y acciones de un objeto para dejar solo aquellas que sean necesarias.

La abstracción es clave para diseñar un buen software.



Abstracción - Ejemplo

¿Qué características podemos abstraer de los animales?



- Características: ...
- Comportamiento: ...

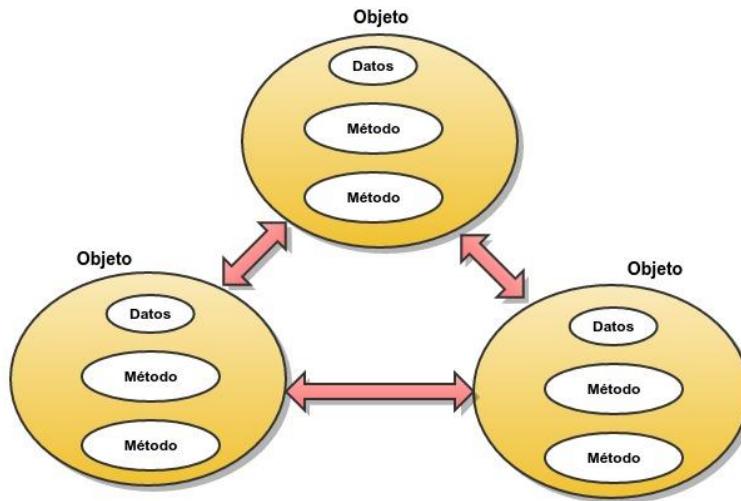
Agenda

- ~~¿Qué es paradigma?~~
- ~~Paradigma Orientado a Objetos~~
- ~~Lenguaje de Programación Orientado a Objetos~~
- Programación Orientada a Objetos:
 - Abstracción
 - **Encapsulamiento**
 - Herencia
 - Polimorfismo

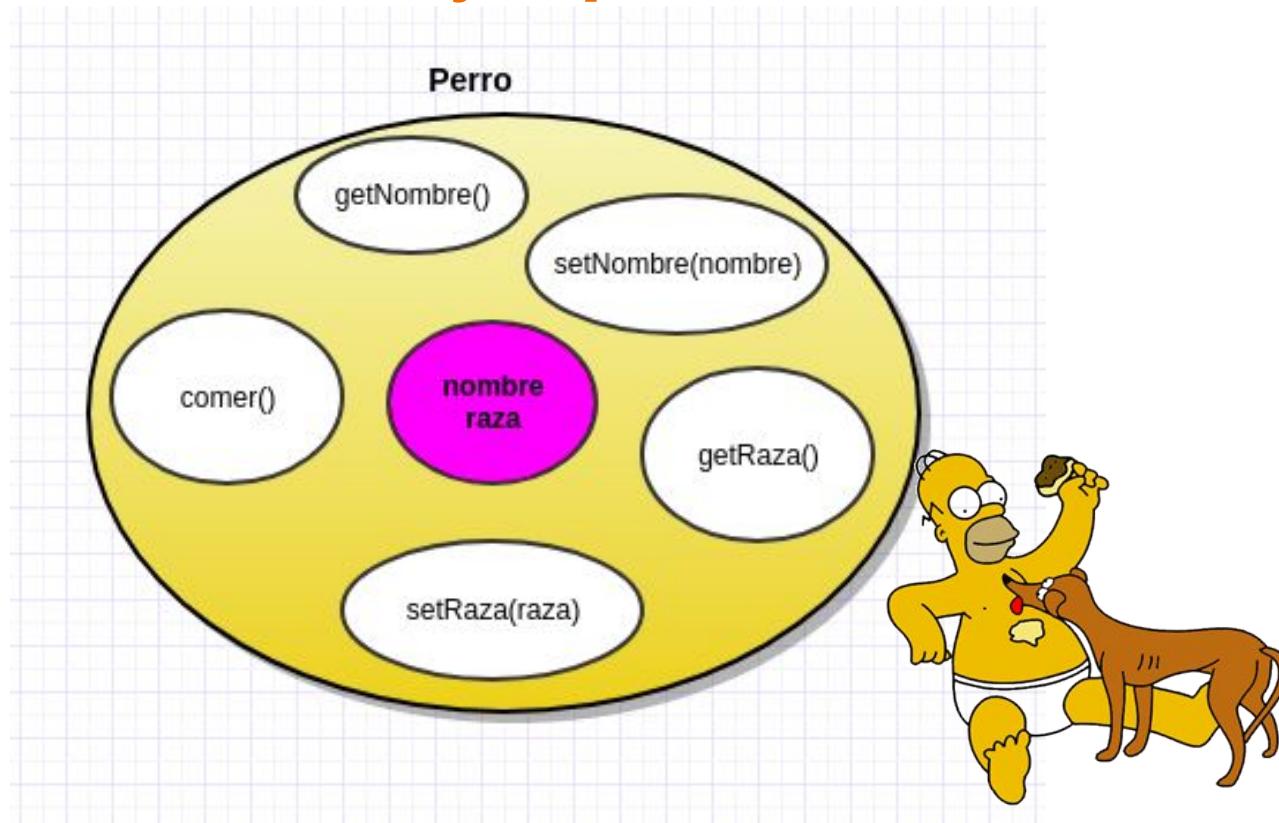


Encapsulamiento

- Ocultamiento de los datos de un objeto de manera que sólo se pueda cambiar mediante las operaciones definidas por ese objeto.
- **Empaqueamiento** → Objetos aislados desde el exterior.



Encapsulamiento - Ejemplo

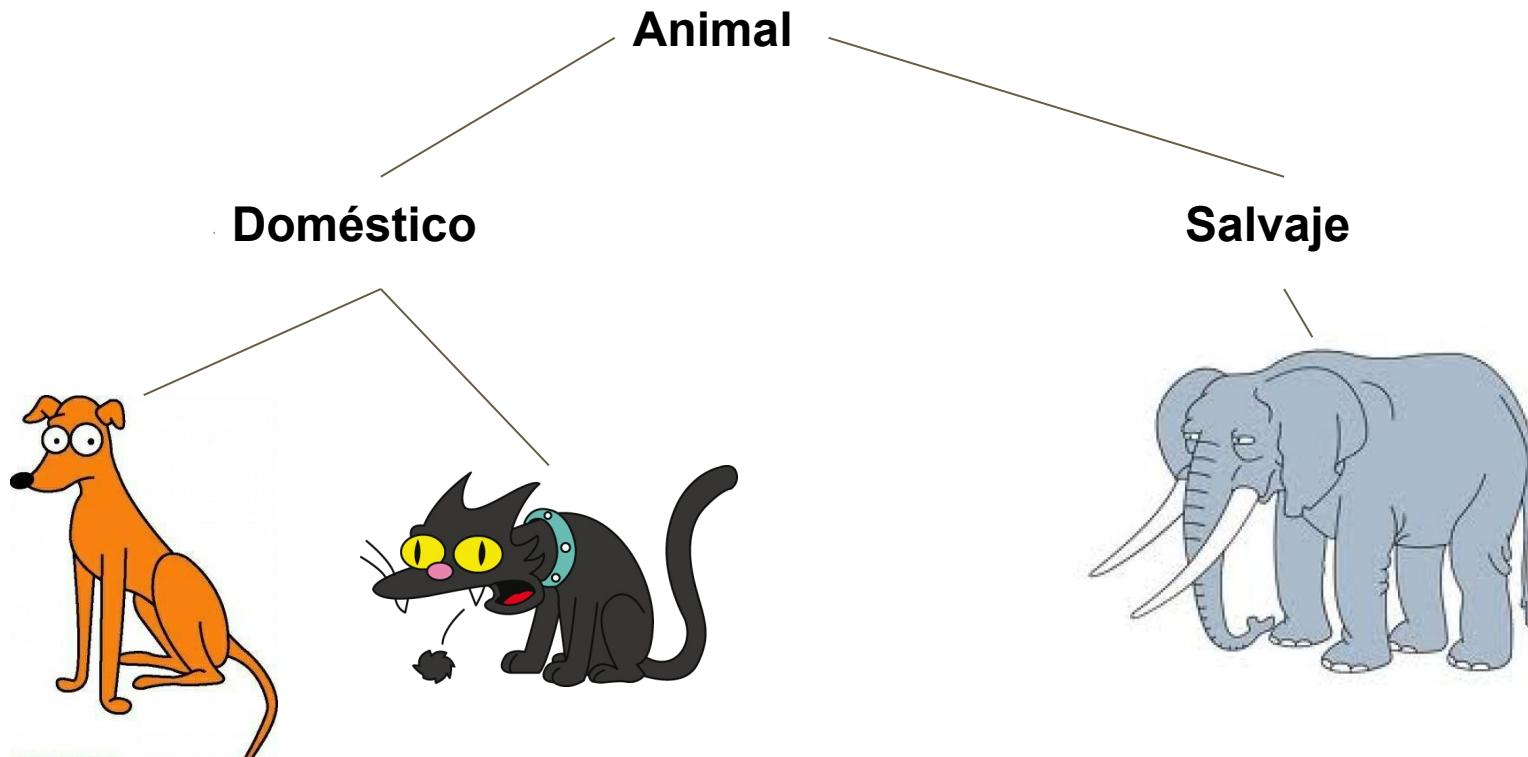


Agenda

- ~~¿Qué es paradigma?~~
- ~~Paradigma Orientado a Objetos~~
- ~~Lenguaje de Programación Orientado a Objetos~~
- Programación Orientada a Objetos:
 - Abstracción
 - Encapsulamiento
 - **Herencia**
 - Polimorfismo



Herencia - Ejemplo



Agenda

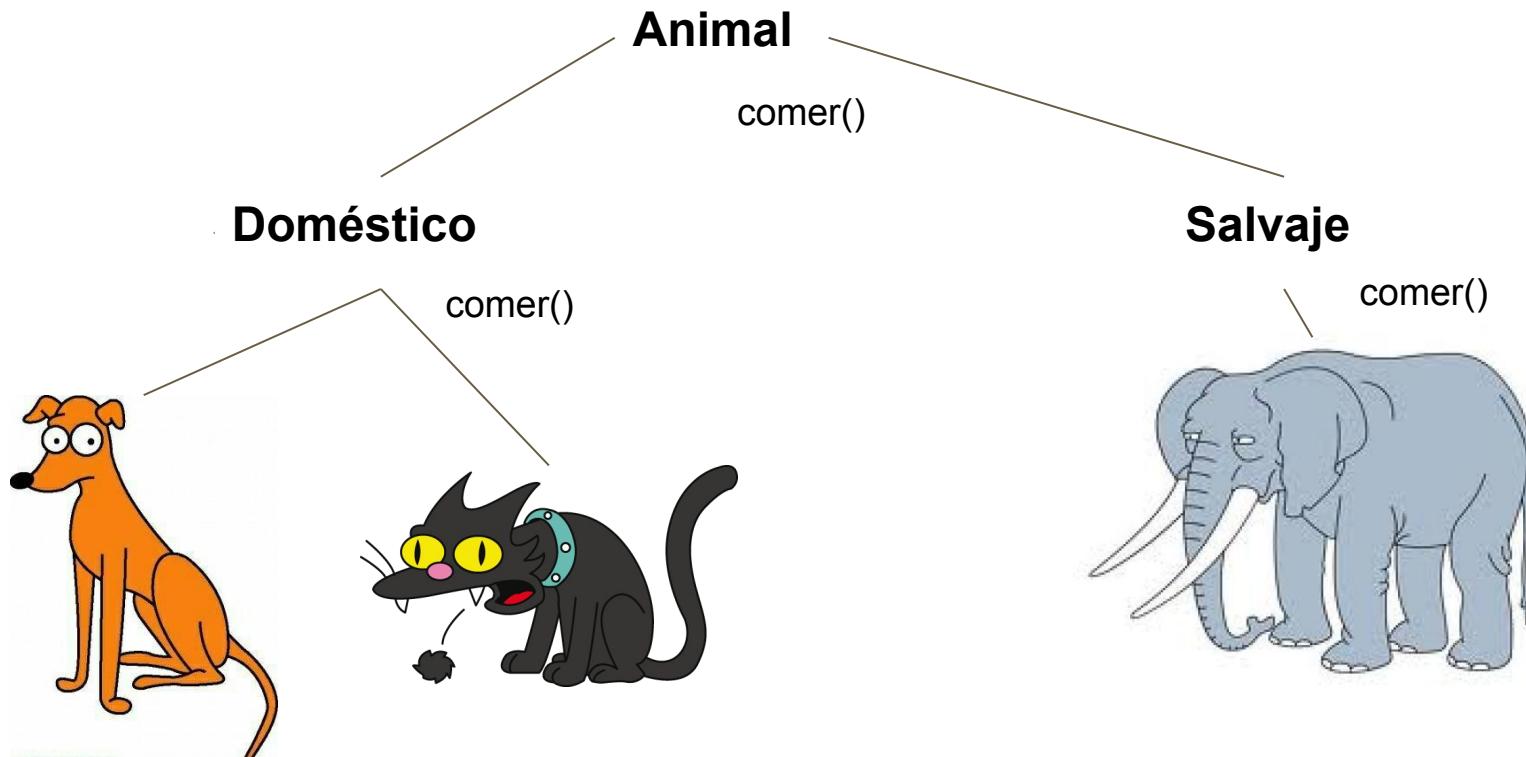
- ~~¿Qué es paradigma?~~
- ~~Paradigma Orientado a Objetos~~
- ~~Lenguaje de Programación Orientado a Objetos~~
- Programación Orientada a Objetos:
 - Abstracción
 - Encapsulamiento
 - Herencia
 - Polimorfismo



Polimorfismo

- Varias formas de responder el mismo mensaje → Muchos mensajes con el **mismo nombre en diferentes clases**.
- Formas de polimorfismo:
 - Sobre-carga de métodos: los mensajes se diferencian en los parámetros.
 - Sobre-escritura de métodos: un hijo sobreescribe un método de la clase padre.
 - Vinculación dinámica: Herencia

Polimorfismo - Ejemplo



POO - Conceptos

— Programación y Laboratorio III —

Agenda

- **Objeto**
- Clase
- Instancia
- Comunicación entre objetos
- Modificadores de acceso



Objeto

- Definición:

Es una entidad autónoma que contiene atributos y comportamiento.

- Se combinan datos y la lógica de programación.
- Tienen **estado** y **comportamiento**.
 - Estado: sustantivos
 - Comportamiento: verbos



Objeto: ejemplo

Objeto Persona

Atributos:

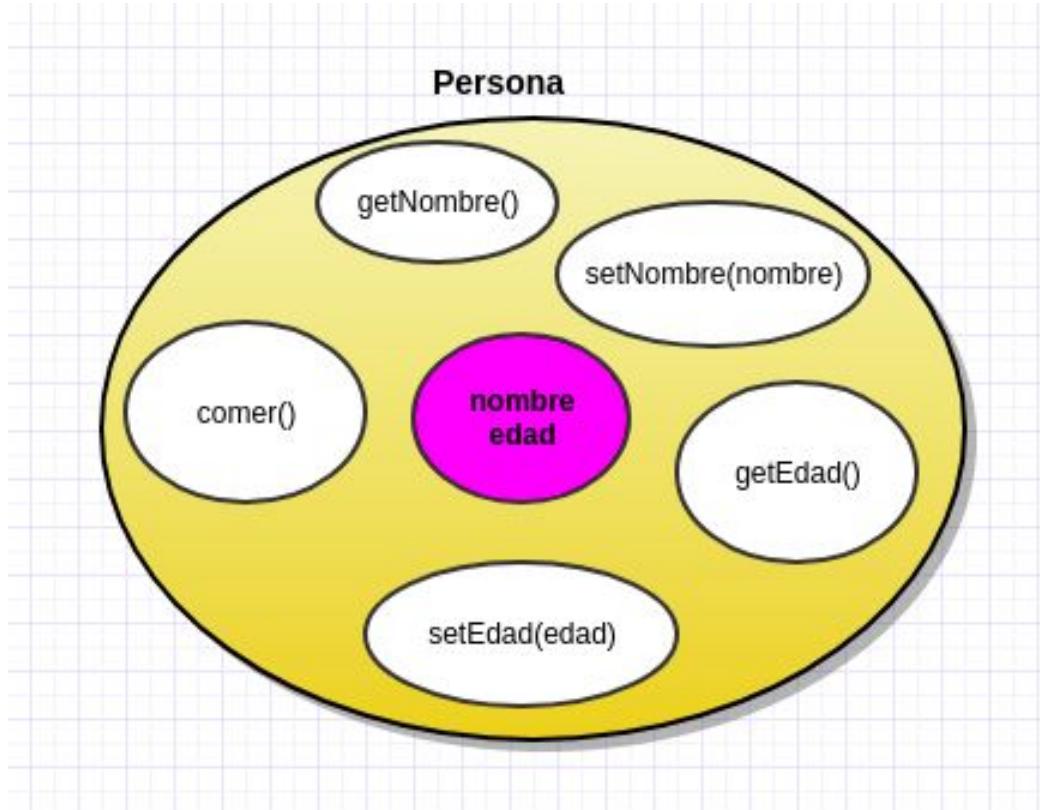
- Nombre
- Edad
- Peso
- Altura
- Fecha de nacimiento
- etc.

Comportamiento:

- Hablar
- Caminar
- Comer
- Imprimir datos
- etc.



Objeto: ejemplo



Agenda

— Objeto

- Clase
- Instancia
- Comunicación entre objetos
- Modificadores de acceso



Clase

- **Definición:**

Es una plantilla para la creación de objetos.

- Cada clase es un modelo que define un conjunto de variables (atributos) y métodos (comportamiento).

Clase: ejemplo

```
class Persona {  
  
    //Atributos  
    String nombre;  
    int edad;  
  
    //Métodos  
    void caminar() {  
        //TODO: cuerpo del método  
    }  
  
    String getNombre() {  
        return nombre;  
    }  
}
```

Agenda

— **Objeto**

— **Clase**

- **Instancia**

- Comunicación entre objetos
- Modificadores de acceso

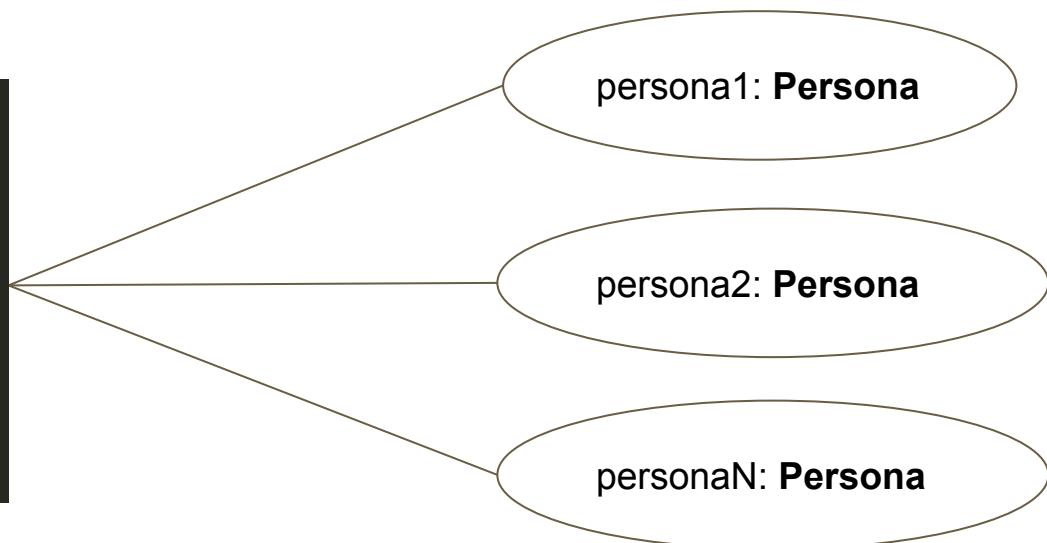


Instancia

- Definición:

Cada objeto creado a partir de una clase.

```
class Persona {  
    //Atributos  
    String nombre;  
    int edad;  
  
    //Métodos  
    void caminar() {  
        //TODO: cuerpo del método  
    }  
  
    String getNombre() {  
        return nombre;  
    }  
}
```

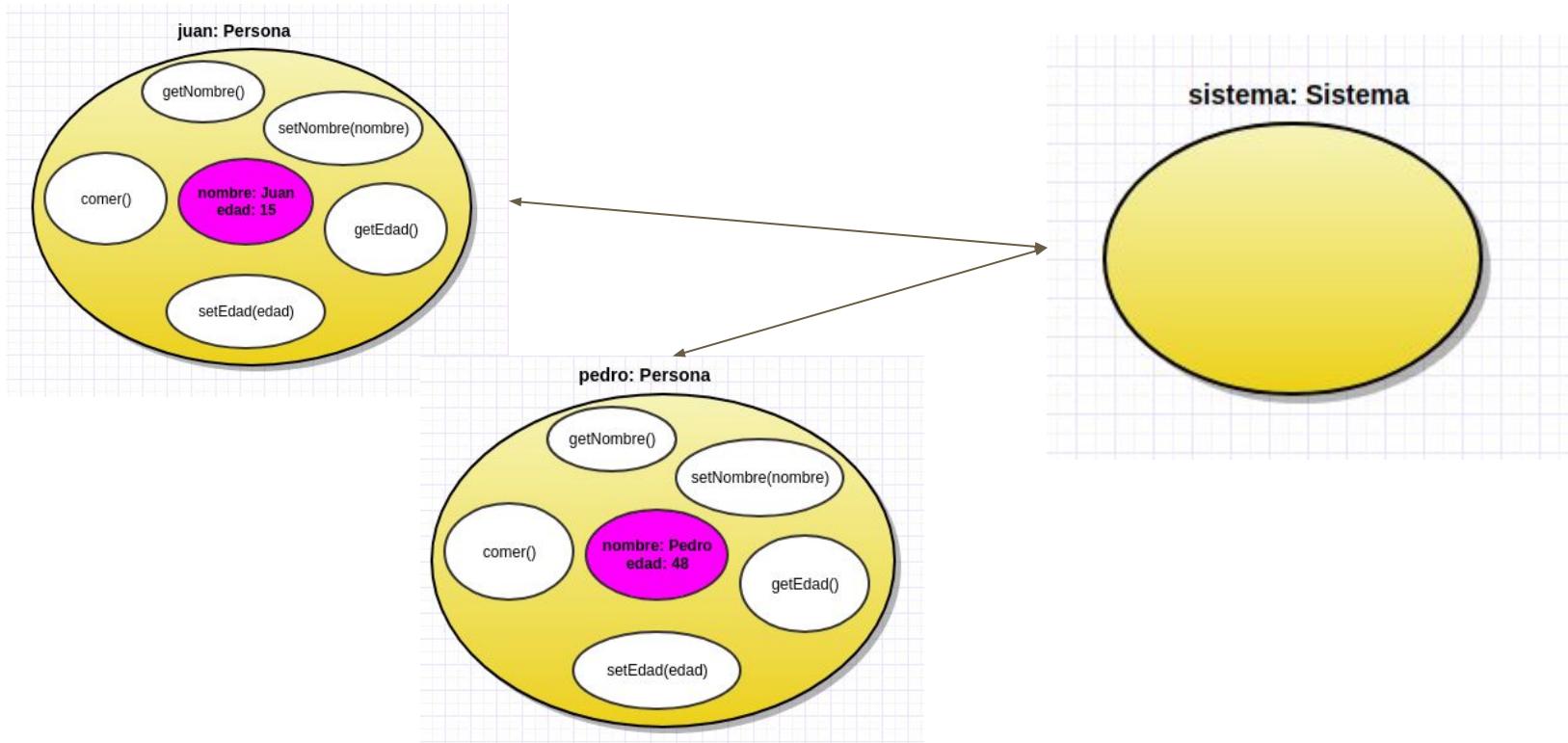


Agenda

- Objeto
- Clase
- Instancia
- **Comunicación entre objetos**
- Modificadores de acceso



Comunicación entre objetos



Agenda

- Objeto
- Clase
- Instancia
- Comunicación entre objetos
- Modificadores de acceso



Modificadores de acceso

- **public:** ofrece la máxima visibilidad. Una variable, método o clase será visible desde cualquier clase.
- **private:** cuando un método o un atributo es declarado como private, su uso queda restringido al interior de la misma clase.
- **protected:** un método o atributo declarado como protected es visible para las clases del mismo paquete y subclases.
- **(default):** visibilidad para clases del mismo paquete.

Modificadores de acceso

Visibilidad	Public	Protected	Default	Private
Desde la misma Clase	SI	SI	SI	SI
Desde cualquier Clase del mismo Paquete	SI	SI	SI	NO
Desde una SubClase del mismo Paquete	SI	SI	SI	NO
Desde una SubClase fuera del mismo Paquete	SI	SI, a través de la herencia	NO	NO
Desde cualquier Clase fuera del Paquete	SI	NO	NO	NO

Modificadores de acceso: ejemplo

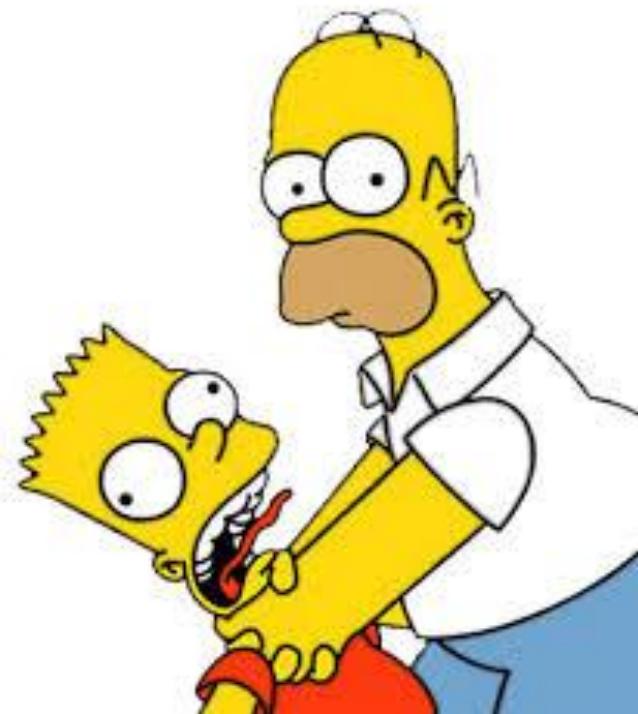
```
public class Persona {  
  
    //Atributos  
    private String nombre;  
    private int edad;  
  
    //Métodos  
    public void caminar() {  
        //TODO: cuerpo del método  
    }  
  
    public String getNombre() {  
        return nombre;  
    }  
}
```

Clase 3: Variables y tipos

— Programación y Laboratorio III —

Agenda

- **Variables**
 - Clasificación
 - Nomenclatura de identificador
- Tipos primitivos
 - Categorización
 - Ejemplos
 - Valores por defecto
- Java Virtual Machine
 - Secciones
 - Tareas principales
 - Garbage Collector
- Tipo objeto
 - Creación
- Diferencias entre variables de tipo objeto y tipo primitivo
- Clases "Wrapper"
 - Ejemplos
 - Unboxing y Autoboxing



Variables

- Una variable es un identificador que representa una palabra de memoria que contiene información.
- Sintaxis:

`<tipo> <identificador>;`

`<tipo> <identificador> = <valor>;`

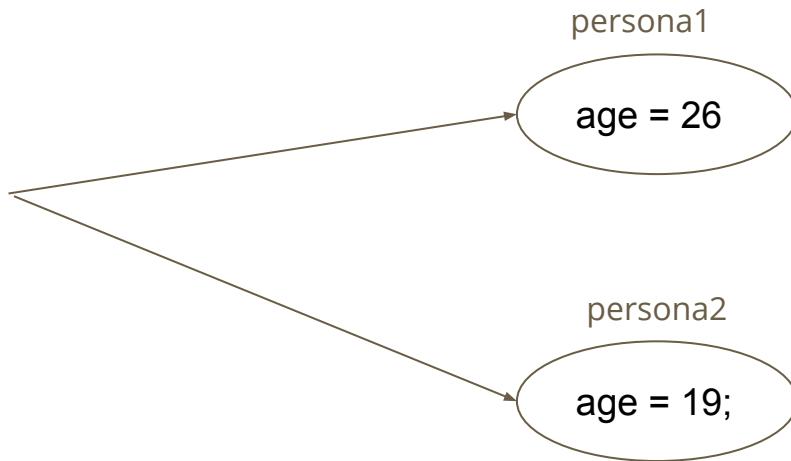


Variables - Clasificación

- **Variables de instancia:** los valores que pueden tomar son únicos para cada instancia.

Ejemplo:

```
class Person {  
    int age;  
}
```



Variables - Clasificación

- **Variables de clase:** se declaran con el modificador static para indicarle al compilador que hay exactamente una copia de la variable, y es compartida por todas las instancias.

Ejemplo:

```
class Bicicleta {  
    static int cantRuedas = 2;;  
}
```



Variables - Clasificación

- **Variables locales:** la determinación viene desde la ubicación en donde la variable fue creada, es decir, local al método. Sólo es visible al método donde fue declarada y no puede ser accedida desde el resto de las clases.

Ejemplo:

```
public void incrementarContador() {  
  
    int contador = 0;  
  
    contador = contador + 1;  
  
}
```

Variables - Clasificación

- **Parámetros:** se pasan entre métodos.

Ejemplos:

```
public void incrementarContador(int contador) {  
    contador = contador + 1;  
}
```



Variables - Nomenclatura de identificador

Reglas:

- 1) Siempre debe comenzar con una letra.
- 2) Los caracteres subsecuentes pueden ser letras, dígitos, "\$", o "_"
- 3) Usar palabras descriptivas y no abreviaciones.
- 4) Los nombres no deben contener palabras reservadas en Java.
- 5) Si el nombre contiene más de una palabra, se capitaliza la primer letra de cada palabra subsecuente.

Variables - Nomenclatura de identificador

Ejemplos:

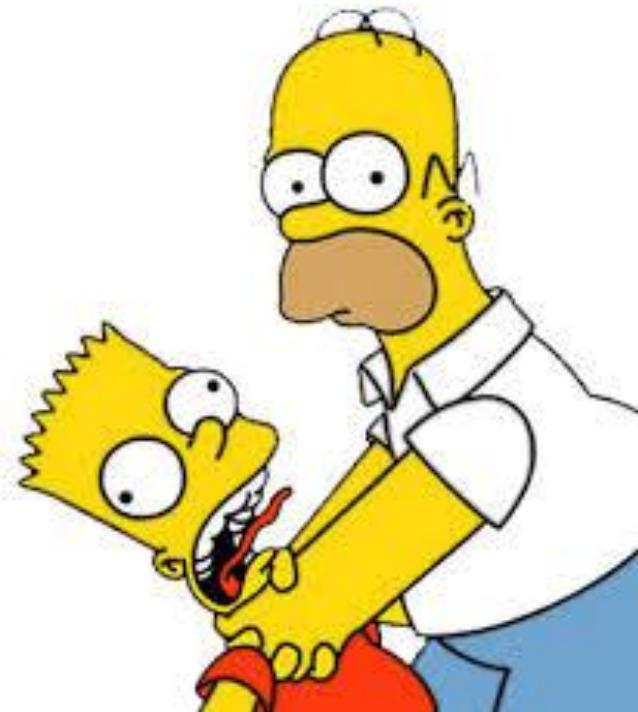
Identificadores válidos	Identificadores no válidos
customerValidObject	7world
\$rate, £Value, _sine	%value
happy2Help, nullValue	Digital!, books@manning
Constant	null, true, false, goto

Variables - Nomenclatura de identificadores

abstract	default	goto	package	this
assert	do	if	private	throw
boolean	double	implements	protected	throws
break	else	import	public	transient
byte	enum	instanceof	return	true
case	extends	int	short	try
catch	false	interface	static	void
char	final	long	strictfp	volatile
class	finally	native	super	while
const	float	new	switch	
continue	for	null	synchronized	

Agenda

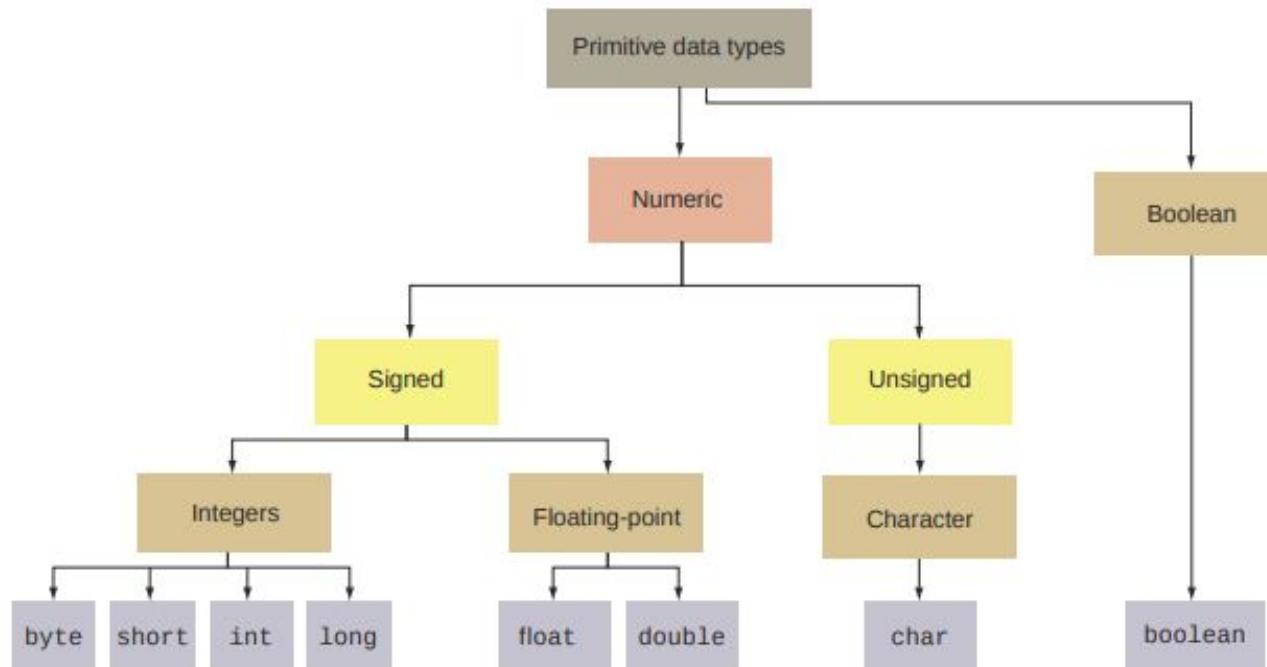
- Variables
 - Clasificación
 - Nomenclatura de identificador
- Tipos primitivos
 - Categorización
 - Ejemplos
 - Valores por defecto
- Java Virtual Machine
 - Secciones
 - Tareas principales
 - Garbage Collector
- Tipo objeto
 - Creación
- Diferencias entre variables de tipo objeto y tipo primitivo
- Clases “Wrapper”
 - Ejemplos
 - Unboxing y Autoboxing



Tipo primitivo

- Son el tipo más simple de la programación orientada a objetos.
- En Java ya están predefinidos.
- Los nombres de los tipos primitivos describen el valor que pueden almacenar.
- Ocho tipos de primitivos:
 - char
 - byte
 - short
 - int
 - long
 - float
 - double
 - boolean

Tipo primitivo - Categorización



Tipo primitivo - Ejemplos

char	byte (8 bits)	short (16 bits)	int (32 bits)	long (64 bits)	float (32 bits)	double (64 bits)	boolean
'a' 'D' '122'	100	1240	48764 0413 0x10B 0b100001011	214748368	20.12F 1765.65f 120.1762	120.176D 120.1762	true false

Tipo primitivo - Valores por defecto

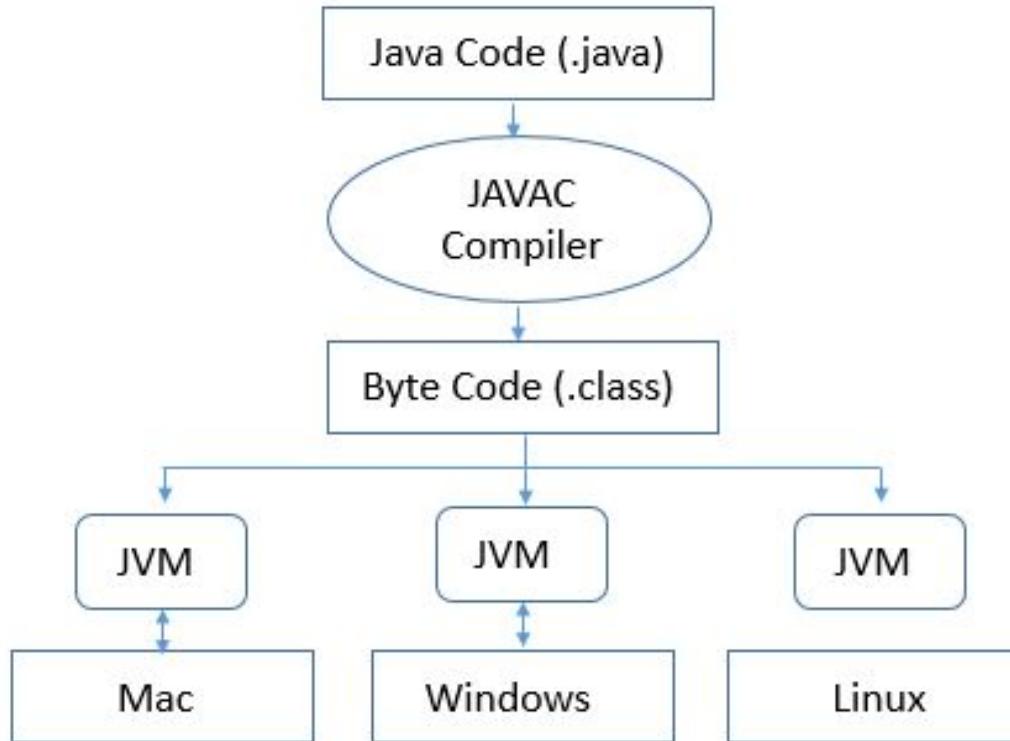
Tipo	Valor por defecto
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
boolean	false

Agenda

- Variables
 - Clasificación
 - Nomenclatura de identificador
- Tipos primitivos
 - Categorización
 - Ejemplos
 - Valores por defecto
- **Java Virtual Machine**
 - Secciones
 - Tareas principales
 - Garbage Collector
- Tipo objeto
 - Creación
- Diferencias entre variables de tipo objeto y tipo primitivo
- Clases “Wrapper”
 - Ejemplos
 - Unboxing y Autoboxing



JVM (Java Virtual Machine)



JVM - Secciones

- **Zona de datos** → donde se almacenan las instrucciones del programa, las clases con sus métodos y constantes. No se puede modificar en tiempo de ejecución.
- **Stack** → El tamaño se define en tiempo de compilación y es estático en tiempo de ejecución. Aquí se almacenan las instancias de los objetos y los datos primitivos (int, float, etc.)
- **Heap** → zona de memoria dinámica. Almacena los objetos que se crean.

JVM - Tareas principales

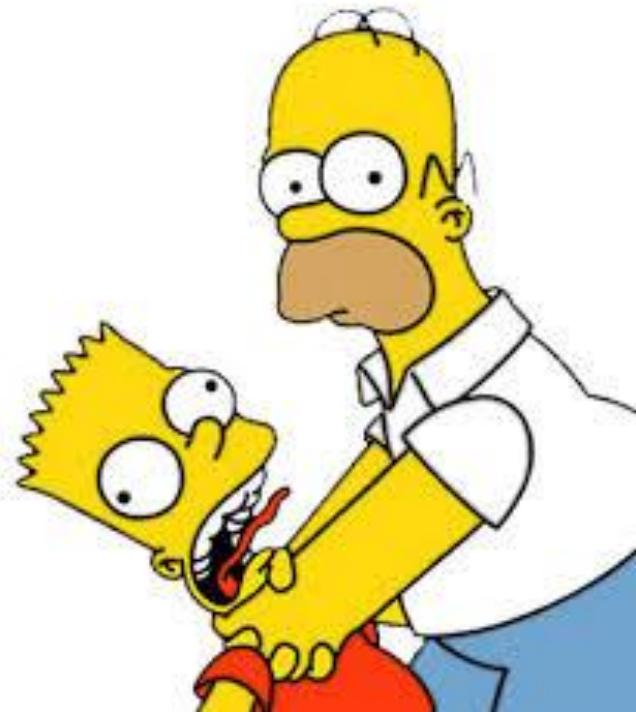
- Reservar espacio en memoria para los objetos creados.
- Liberar la memoria no usada.
- Asignar variables a registros y pilas.
- Llamar al sistema huésped para ciertas funciones, como los accesos a los dispositivos.
- Vigilar el cumplimiento de las normas de seguridad de las aplicaciones Java. Por ejemplo:
 - No se permiten realizar ciertas conversiones (casting) entre distintos tipos de datos.
 - Las referencias a arrays son verificadas en el momento de la ejecución del programa.

JVM - Garbage Collector

- Es un proceso de baja prioridad que se ejecuta dentro de la JVM.
- Técnica por la cual el ambiente de objetos se encarga de destruir y asignar automáticamente la memoria heap.
- El programador no debe preocuparse por la asignación y liberación de memoria, ya que el entorno la asignará al crear un nuevo objeto y la liberará cuando nadie la esté usando.
- Un objeto podrá ser “limpiado” cuando desde el stack ninguna variable haga referencia al mismo.

Agenda

- Variables
 - Clasificación
 - Nomenclatura de identificador
- Tipos primitivos
 - Categorización
 - Ejemplos
 - Valores por defecto
- Java Virtual Machine
 - Secciones
 - Tareas principales
 - Garbage Collector
- **Tipo objeto**
 - Creación
- Diferencias entre variables de tipo objeto y tipo primitivo
- Clases “Wrapper”
 - Ejemplos
 - Unboxing y Autoboxing

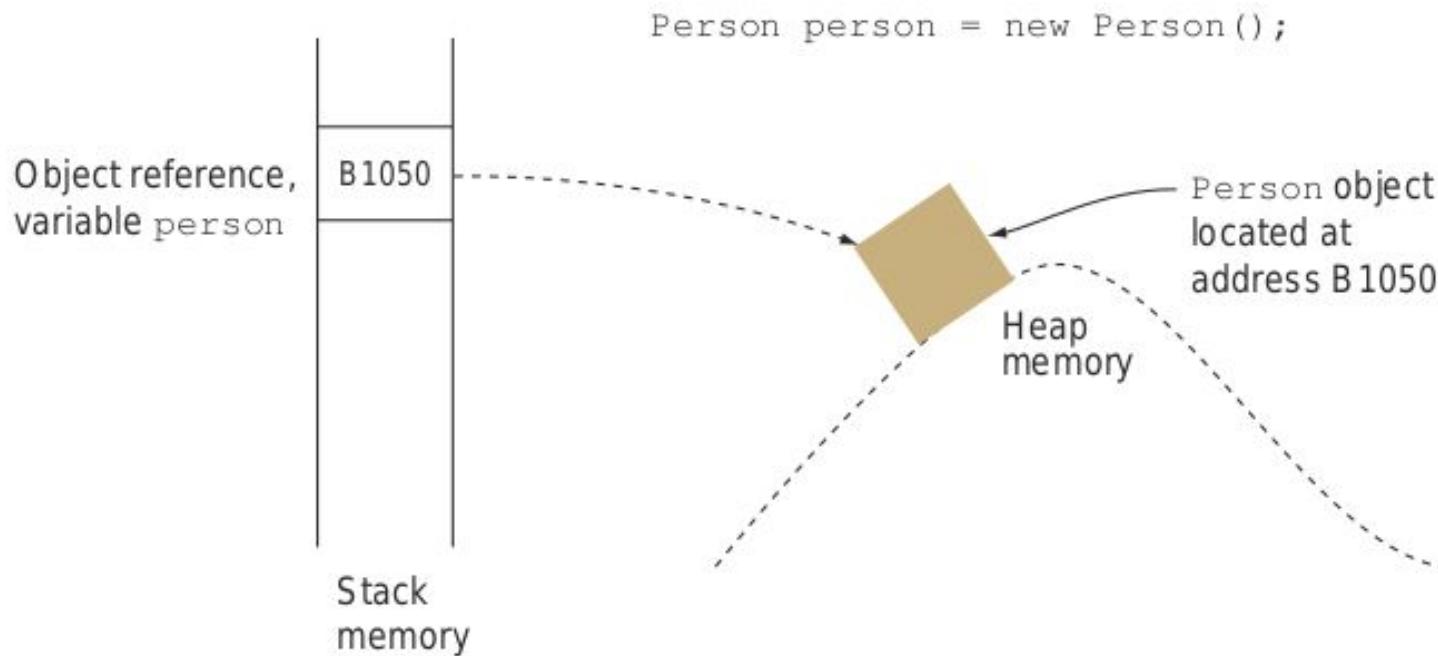


Tipo objeto

- Son instancias de clases, que pueden estar predefinidas o definidas por el programador.
- Cuando un objeto es instanciado utilizando el operador **new**, se retorna una dirección de memoria.
- La dirección en memoria contiene una referencia a una variable.



Tipo objeto - Creación



Agenda

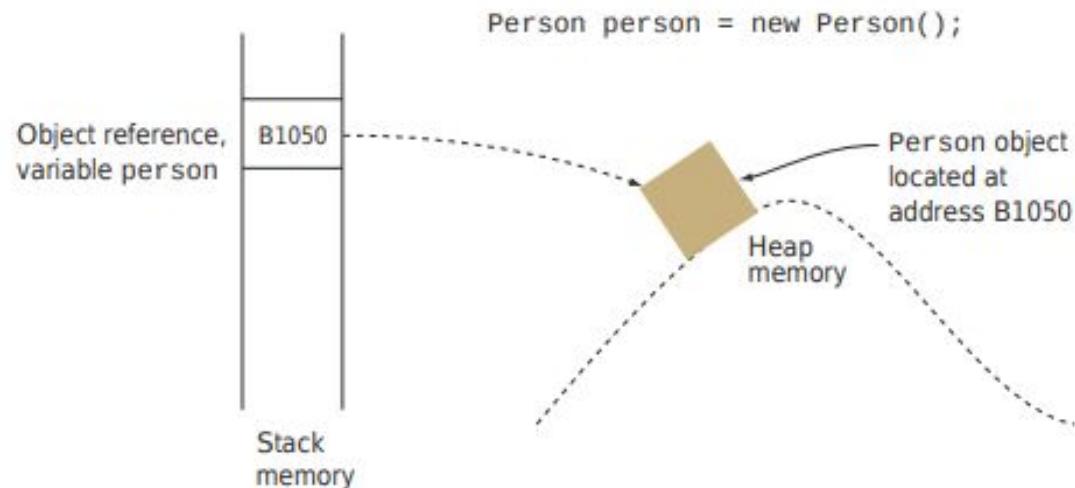
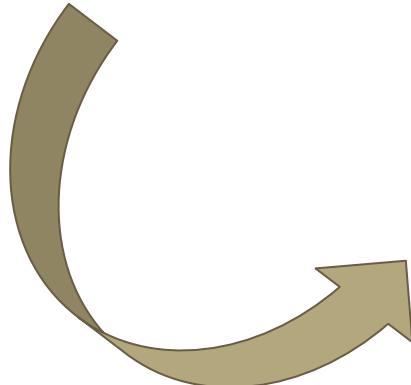
- Variables
 - Clasificación
 - Nomenclatura de identificador
- Tipos primitivos
 - Categorización
 - Ejemplos
 - Valores por defecto
- Java Virtual Machine
 - Secciones
 - Tareas principales
 - Garbage Collector
- Tipo objeto
 - Creación
- **Diferencias entre variables de tipo objeto y tipo primitivo**
- Clases “Wrapper”
 - Ejemplos
 - Unboxing y Autoboxing



Diferencias entre variables de tipo objeto y tipo primitivo

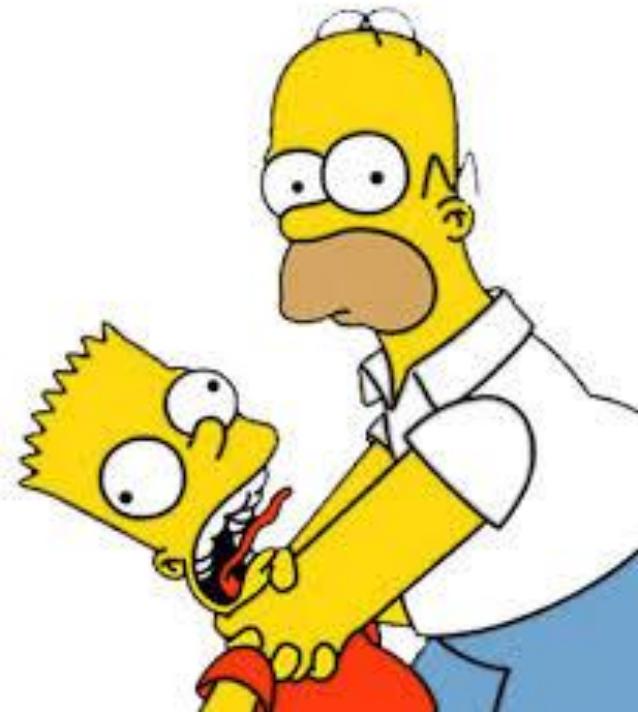
```
int a = 77;
```

```
Persona person = new Person();
```



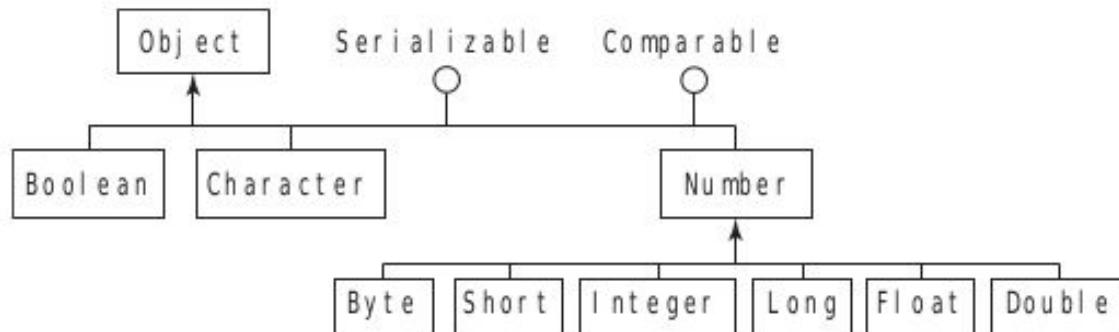
Agenda

- Variables
 - Clasificación
 - Nomenclatura de identificador
- Tipos primitivos
 - Categorización
 - Ejemplos
 - Valores por defecto
- Java Virtual Machine
 - Secciones
 - Tareas principales
 - Garbage Collector
- Tipo objeto
 - Creación
- Diferencias entre variables de tipo objeto y tipo primitivo
- **Clases "Wrapper"**
 - Ejemplos
 - Unboxing y Autoboxing



Clases “Wrapper”

- Java define una clase “wrapper” para cada tipo primitivo.



Clases “Wrapper” - Ejemplos

```
Boolean bool1 = true;
```

```
Character char1 = 'a';
```

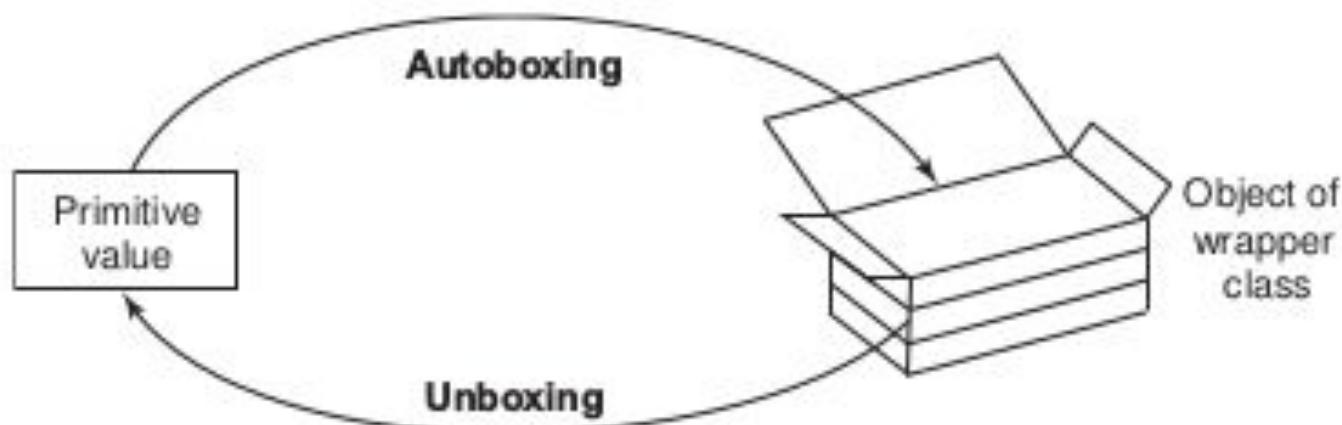
```
Double double1 = 10.98;
```

```
Boolean bool2 = new Boolean(true);
```

```
Character char2 = new Character('a');
```

```
Double double2 = new Double(10.98);
```

Clases “Wrapper” - Unboxing y Autoboxing



Operadores y Flujo de control

— Programación y Laboratorio III —

Agenda

- Operadores
 - Operadores de asignación
 - Operadores aritméticos
 - Operadores relacionales
 - Operadores lógicos
 - Precedencia de operadores
- Control de flujo
 - if, if-else
 - switch
 - for
 - while, do-while



Operadores

Tipo	Operadores	Propósito
Asignación	=, +=, -=, *=, /=	Asignar valor a una variable
Aritmético	+, -, *, /, %, ++, --	Sumar, restar, multiplicar, dividir y módulo de primitivas
Relacional	<, <=, >, >=, ==, !=	Comparan primitivas
Lógico	!, &&,	Aplicar NOT, AND y OR a primitivas

Agenda

- Operadores
 - Operadores de asignación
 - Operadores aritméticos
 - Operadores relacionales
 - Operadores lógicos
 - Precedencia de operadores
- Control de flujo
 - if, if-else
 - switch
 - for
 - while, do-while



Operadores de asignación

= → usado para inicializar variables con valores o reasignar nuevos valores.

a -= b → a = a - b

a += b → a = a + b

a *= b → a = a * b

a /= b → a = a / b

a %= b → a = a % b

Operadores de asignación - Ejemplos

```
double d = 10.2;
int a = 10;
int b = a;
float f = 10.2F;

b += a;    // b = 20
a = b = 10;
b -= a;    // b = 0

d = true;
boolean b = 0;
boolean bTrue = true;
boolean bFalse = false;
bTrue -= bFalse;

long num = 100976543356L;
int val = num; // No se permite

int num1 = 1009;
long val1 = num1; // Se permite
```

Agenda

- **Operadores**
 - Operadores de asignación
 - **Operadores aritméticos**
 - Operadores relacionales
 - Operadores lógicos
 - Precedencia de operadores
- Control de flujo
 - if, if-else
 - switch
 - for
 - while, do-while



Operadores aritméticos

Operador	Propósito
+	Sumar
-	Restar
*	Multiplicar
/	Dividir
%	Resto en la división
++	Incrementa en 1
--	Decrementa en 1

Operadores aritméticos - Ejemplos

```
char char1 = 'a';
System.out.println(char1 + char1); //194

byte age1 = 10;
byte age2 = 20;
short sum = age1 + age2; //Tipos incompatibles

int a = 20;
int b = 10;
++a;
b++;
System.out.println(a); // 21
System.out.println(b); // 11

int a = 20;
int b = 10;
int c = a - ++b;
System.out.println(c); // 9
System.out.println(b); // 11

int a = 20;
int b = 10;
int c = a - b++;
System.out.println(c); // 10
System.out.println(b); // 11
```

Agenda

- **Operadores**
 - Operadores de asignación
 - Operadores aritméticos
 - **Operadores relacionales**
 - Operadores lógicos
 - Precedencia de operadores
- Control de flujo
 - if, if-else
 - switch
 - for
 - while, do-while



Operadores relacionales

- Se usan para determinar si el valor de una primitiva es igual, menor o mayor al valor de otra.

Operador	Uso
>, >=, <, <=	Comparan mayor y menor.
==, !=	Comparan igualdad.

Operadores relacionales - Ejemplos

```
int i1 = 10;
int i2 = 20;
System.out.println(i1 >= i2);    // false

long l1 = 10;
long l2 = 20;
System.out.println(l1 <= l2);    // true

int a = 10;
int b = 20;
System.out.println(a == b);    // false
System.out.println(a != b);    // true

boolean b1 = false;
System.out.println(b1 == true);  // false
System.out.println(b1 != false); // false
System.out.println(b1 == false); // true
```

Agenda

- Operadores
 - Operadores de asignación
 - Operadores aritméticos
 - Operadores relacionales
 - Operadores lógicos
 - Precedencia de operadores
- Control de flujo
 - if, if-else
 - switch
 - for
 - while, do-while



Operadores lógicos

- Se utilizan para evaluar una o más expresiones. La evaluación retorna un valor boolean.

Operador	Uso
&&	AND
	OR
!	NOT

Operadores lógicos - Ejemplos

```
int a = 10;
int b = 20;
System.out.println(a > 20 && b < 10); // false
System.out.println(a > 20 || b > 10); // true
System.out.println(!(b > 10)); // false
System.out.println(!(a > 20)); // true

int marks = 8;
int total = 10;
System.out.println(total < marks && ++marks > 5); // false
```

Agenda

- Operadores
 - Operadores de asignación
 - Operadores aritméticos
 - Operadores relacionales
 - Operadores lógicos
 - Precedencia de operadores
- Control de flujo
 - if, if-else
 - switch
 - for
 - while, do-while



Precedencia de operadores

Operador
expresion++, expresion--
++expresion, --expresion, +expresion, -expresion, !
*, /, %
+, -
<, >, <=, >=
==, !=
&&
=, +=, -=, *=, /=, %=

Agenda

- Operadores

- Operadores de asignación
- Operadores aritméticos
- Operadores relacionales
- Operadores lógicos
- Precedencia de operadores

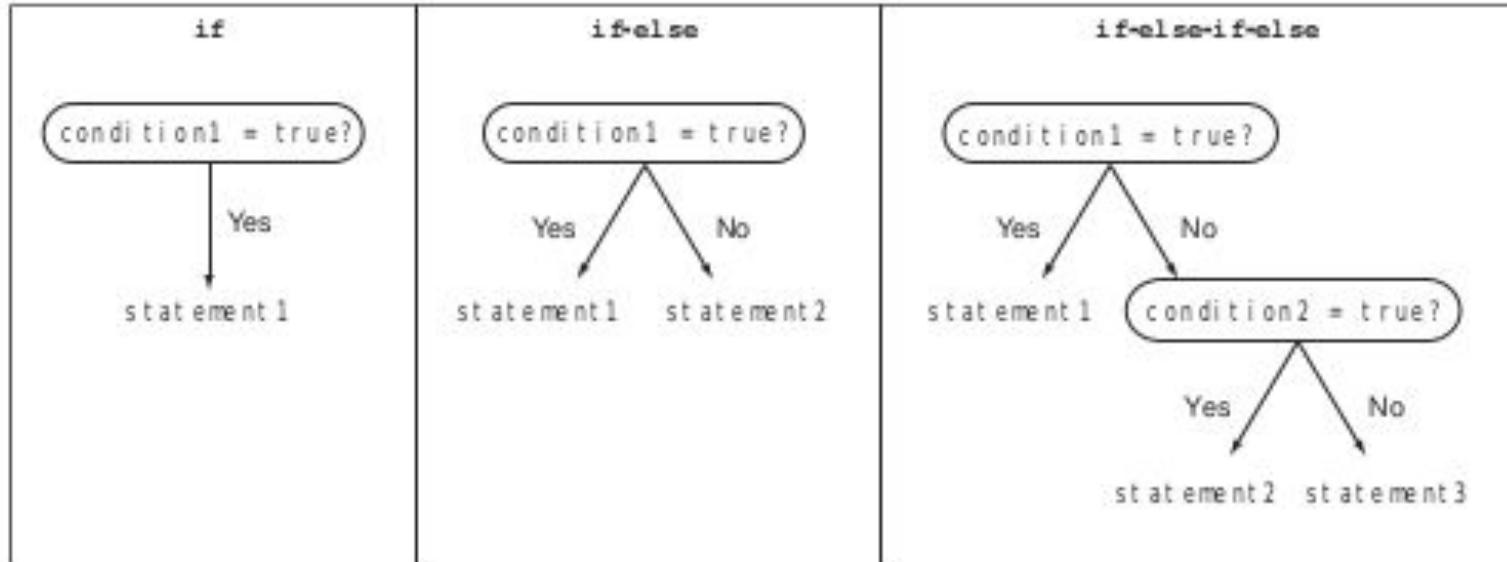
- Control de flujo

- if, if-else
- switch
- for
- while, do-while



Control de flujo: if, if-else

- Nos permite ejecutar una serie de sentencias, según el resultado de una condición.
- El resultado de evaluar la condición debe ser boolean o Boolean.



if, if-else - Ejemplos

```
//if
if (b == true) {
    score = 200;
}

//if-else
if (b == true) {
    score = 200;
} else {
    score = 300;
}

if (score == 200) {
    result = 'A';
} else if (score == 300) {
    result = 'B';
} else {
    result = 'C';
}
```

Agenda

~~Operadores~~

- ~~Operadores de asignación~~
- ~~Operadores aritméticos~~
- ~~Operadores relacionales~~
- ~~Operadores lógicos~~
- ~~Precedencia de operadores~~

Control de flujo

- ~~if, if else~~
- **switch**
- **for**
- **while, do-while**



Control de flujo: switch

- Se utiliza cuando la variable a evaluar tiene múltiples valores.

```
switch (value) {  
    case sth1 :  
        statements;  
        break;  
    case sth2 :  
        statements;  
        break;  
    default :  
        statements;  
        break;  
}
```

switch - Ejemplo

```
int marks = 20;

switch (marks) {
    case 10 : System.out.println(10);
    break;
    case 20 : System.out.println(20);
    break;
    case 30 : System.out.println(30);
    break;
    default : System.out.println("default");
    break;
}
```

Agenda

~~Operadores~~

- ~~Operadores de asignación~~
- ~~Operadores aritméticos~~
- ~~Operadores relacionales~~
- ~~Operadores lógicos~~
- ~~Precedencia de operadores~~

Control de flujo

- ~~if, if else~~
- ~~switch~~
- **for**
- **while, do-while**



Control de flujo: for

- Se utiliza cuando se necesita repetir la(s) misma(s) línea(s) de código múltiples veces.

```
for (initialization; condition; update) {  
    statements;  
}
```

for - Ejemplo

```
int tableOf = 25;
for (int ctr = 1; ctr <=5; ctr++) {
    System.out.println(tableOf * ctr);
}

for (int hrs = 1; hrs <=6; hrs++) {
    for (int min = 1; min <= 60; min++) {
        System.out.println(hrs + ":" + min);
    }
}
```

Agenda

~~Operadores~~

- ~~Operadores de asignación~~
- ~~Operadores aritméticos~~
- ~~Operadores relacionales~~
- ~~Operadores lógicos~~
- ~~Precedencia de operadores~~

Control de flujo

- ~~if, if else~~
- ~~switch~~
- ~~for~~
- **while, do-while**



Control de flujo: while, do-while

- Ejecutan una serie de sentencias hasta que la condición de corte sea igual a true.
- La principal diferencia entre ambos es que `while` chequea la condición antes de ejecutar el cuerpo, mientras que en `do-while` evalúa la condición después de ejecutar las sentencias definidas en el cuerpo.

```
while (condition) {  
    statements;  
}
```

```
do {  
    statements;  
} while (condition);
```

while, do-while - Ejemplos

```
int num = 9;
boolean divisibleBy7 = false;

while (!divisibleBy7) {
    System.out.println(num);
    if (num % 7 == 0) {
        divisibleBy7 = true;
    }
    --num;
}

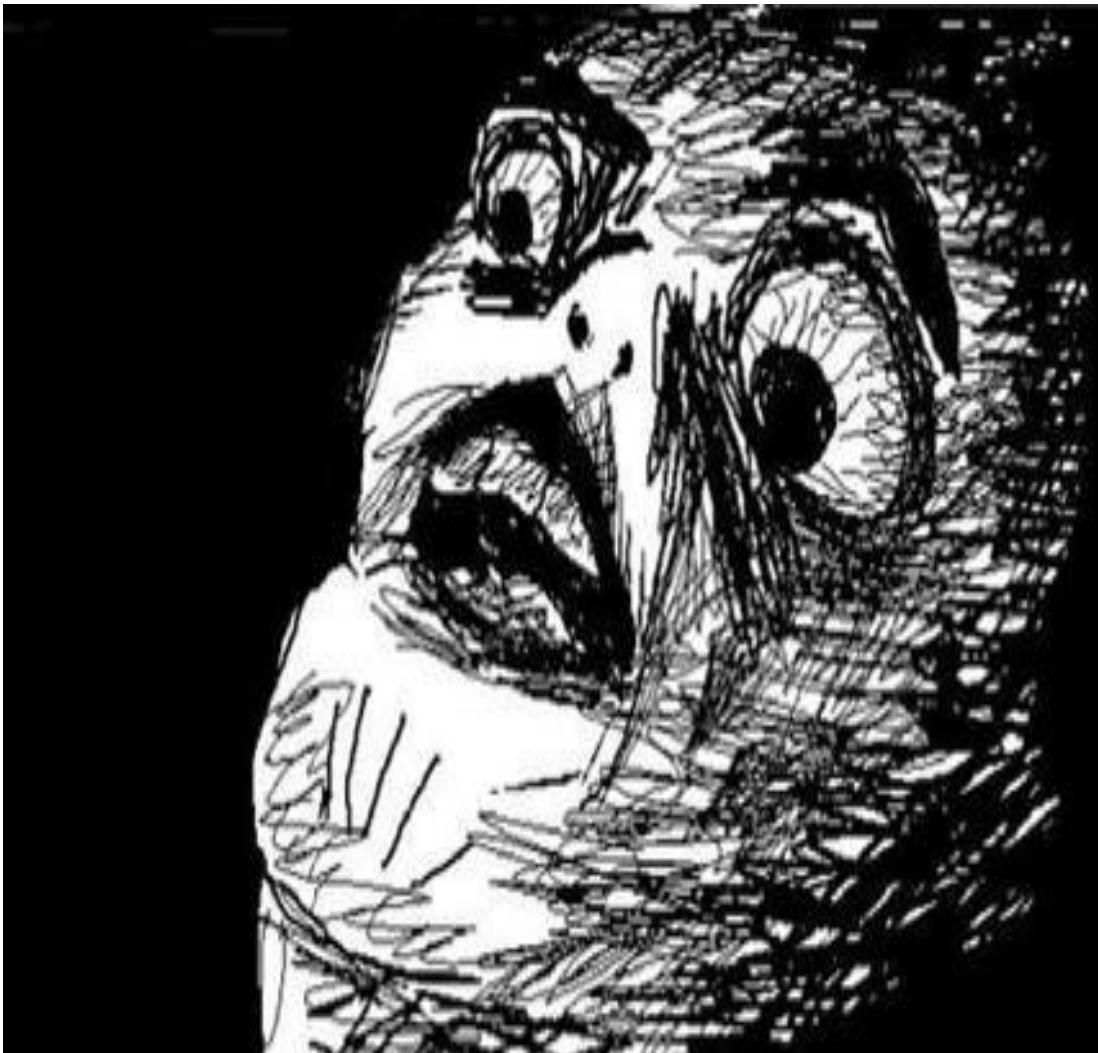
int num = 9;
boolean divisibleBy7 = false;
do {
    System.out.println(num);
    if (num % 7 == 0) {
        divisibleBy7 = true;
    }
    num--;
} while (divisibleBy7 == false);
```

static & non-static

— Programación y Laboratorio III —

Métodos que no usan
valores variables de
instancia.

No es necesario crear
una instancia de la
clase.

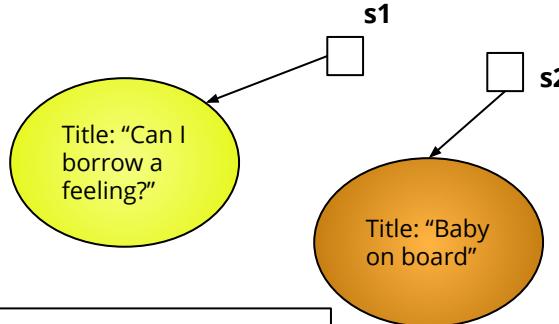
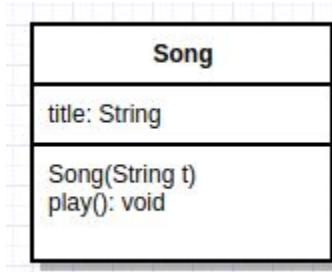


Agenda

- Método regular (**non-static**)
- Método estático
- Métodos regulares vs. estáticos
- Variables estáticas
- Constantes
- **final**



Método regular (non-static)



```
Song s1 = new Song();\ns1.play();\nSong s2 = new Song();\ns2.play();
```

Variable de instancia

```
public class Song {\n    private String title;\n\n    public Song(String t) {\n        title = t;\n    }\n\n    public void play() {\n        SoundPlayer player = new SoundPlayer();\n        player.playSound(title);\n    }\n}
```

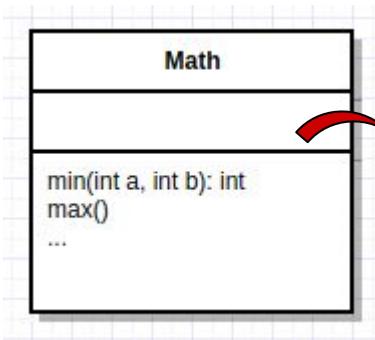


Agenda

- ~~Método regular (non static)~~
- **Método estático**
- Métodos regulares vs. estáticos
- Variables estáticas
- Constantes
- final



Método estático



```
public static int min(int a, int b) {  
    //return a or b  
}
```



No hay instancias
de objetos

```
int min = Math.min(42, 39);
```

Agenda

- ~~Método regular (non static)~~
- ~~Método estático~~
- **Métodos regulares vs. estáticos**
- Variables estáticas
- Constantes
- final



Métodos regulares (non-static) vs. estáticos

- Un método declarado con la palabra reservada `static` nos indica que se puede invocarlo sin necesidad de crear una instancia de la clase.
- Se pueden combinar métodos regulares y estáticos en la misma clase.
- Los métodos estáticos no pueden usar variables de instancia.
- Los métodos estáticos no pueden usar métodos regulares, porque usan variables de instancias.



Ejemplos

```
public class Duck {  
    private int size;  
  
    public static void main (String[] args) {  
        System.out.println("Size = " + size);  
    }  
  
    public void setSize(int s) {  
        size = s;  
    }  
  
    public int getSize() {  
        return size;  
    }  
}
```



```
public class Duck {  
    private int size;  
  
    public static void main (String[] args) {  
        System.out.println("Size = " + getSize());  
    }  
  
    public void setSize(int s) {  
        size = s;  
    }  
  
    public int getSize() {  
        return size;  
    }  
}
```



Agenda

- ~~Método regular (non static)~~
- ~~Método estático~~
- ~~Métodos regulares vs. estáticos~~
- **Variables estáticas**
- Constantes
- final



Variables estáticas

“Un valor compartido por todas las instancias una clase”

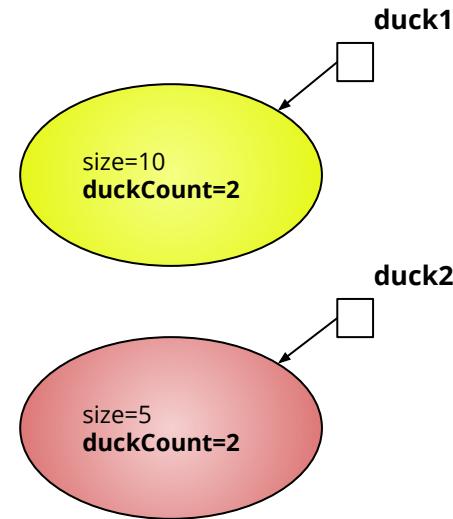


Variables estáticas

```
public class Duck {  
    private int size;  
    private static int duckCount = 0;  
  
    Public Duck() {  
        duckCount++;  
    }  
  
    public void setSize(int s) {  
        size = s;  
    }  
  
    public int getSize() {  
        return size;  
    }  
}
```



**Se inicializa una única vez,
cuando la clase de carga por
primera vez.**



Agenda

- ~~Método regular (non static)~~
- ~~Método estático~~
- ~~Métodos regulares vs. estáticos~~
- ~~Variables estáticas~~
- **Constantes**
- **final**



Constantes

```
public static final double PI = 3.141592653589793
```

- La palabra reservada `final` indica que una vez inicializada, el valor de la variable no puede cambiar.
- Generalmente se establecen como `public` para que puedan ser accedidas desde cualquier lugar de nuestro código.
- Son estáticas para que no sea necesario crear una instancia de la clase para poder usarlas.
- EL NOMBRE DE UNA CONSTANTE DEBE ESTAR EN MAYÚSCULA.

Agenda

- Método regular (~~non static~~)
- Método estático
- Métodos regulares vs. estáticos
- Variables estáticas
- Constantes
- **final**



final

- La palabra reservada `final` no es sólo para variables estáticas.
- Se puede usar `final` para variables de instancias, variables locales, parámetros de métodos y clases.
- Indica que el valor no puede cambiar una vez que fue inicializado.



final



- Una variable **final** significa que no puede cambiar su valor.
- Un método **final** significa que no puede sobreescribirse.
- Una clase **final** significa que no puede tener subclases.

Bibliografía oficial

- <https://docs.oracle.com/javase/tutorial/java/javaOO/classvars.html>
- <https://docs.oracle.com/javase/tutorial/java/javaOO/initial.html>
- <https://docs.oracle.com/javase/8/docs/technotes/guides/language/static-import.html>
- <https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html#jvms-2.5>

Clase 6: Herencia

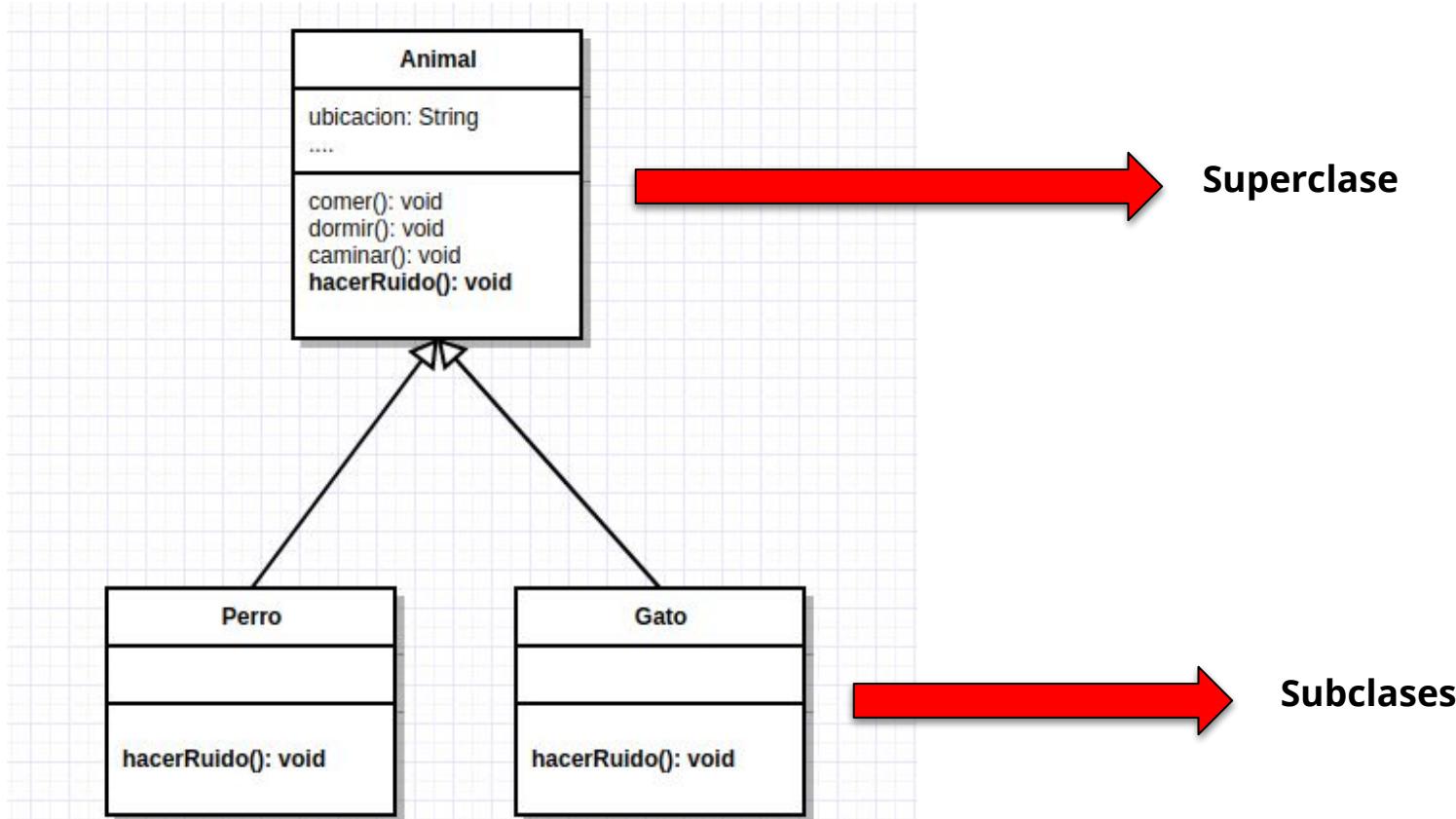
— Programación y Laboratorio III —

Agenda

- Herencia
 - UML
 - Definición
 - Superclase y subclases
 - Ventajas
- Herencia y Abstracción
- Herencia en Java
- Clase abstracta
 - Definición
 - Método abstracto
 - Constructores



Herencia UML - Ejemplo práctico



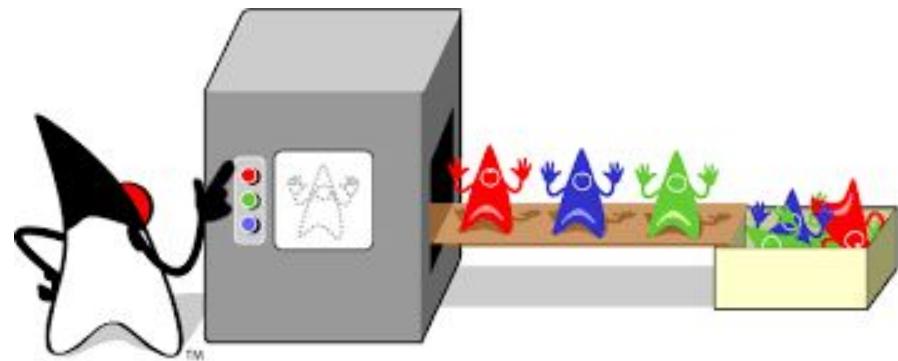
Agenda

- **Herencia**
 - UML
 - Definición
 - Superclase y subclases
 - Ventajas
- Herencia y Abstracción
- Herencia en Java
- Clase abstracta
 - Definición
 - Método abstracto
 - Constructores



Herencia - Definición

- La herencia es un mecanismo que permite la definición de una clase a partir de la definición de otra ya existente.
- Permite compartir automáticamente métodos y características entre clases.
- Está fuertemente ligada a la reutilización de código.



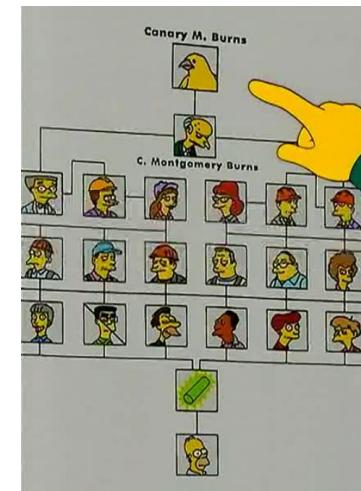
Agenda

- Herencia
 - UML
 - Definición
 - Superclase y subclases
 - Ventajas
- Herencia y Abstracción
- Herencia en Java
- Clase abstracta
 - Definición
 - Método abstracto
 - Constructores



Herencia - Superclase y subclases

- El concepto de herencia conduce a una estructura jerárquica de clases o estructura de árbol.
- En la estructura jerárquica, cada clase tiene sólo una clase padre → **SUPERCLASE**.
 - Una superclase puede tener cualquier número de subclases.
- La clase hija de una superclase se llama **SUBCLASE**.
 - Una subclase puede tener sólo una superclase.
- En el ejemplo:
 - Animal es la superclase de Gato y Perro.
 - Gato y Perro son subclases de Animal.



Agenda

- **Herencia**
 - UML
 - Definición
 - Superclase y subclases
 - **Ventajas**
- Herencia y Abstracción
- Herencia en Java
- Clase abstracta
 - Definición
 - Método abstracto
 - Constructores



Herencia - Ventajas

- Evitar la duplicidad y favorecer la **reutilización** de código (las subclases utilizan el código de la superclase).
- Facilitar el **mantenimiento** de las aplicaciones que diseñamos.
- Facilitar la **extensión** de las aplicaciones que diseñamos.

Agenda

- Herencia

- UML
- Definición
- Superclase y subclases
- Ventajas

- Herencia y Abstracción

- Herencia en Java
- Clase abstracta
 - Definición
 - Método abstracto
 - Constructores



Herencia y Abstracción

- Se puede pensar en una jerarquía de clases como la definición de **conceptos abstractos** en lo alto de la jerarquía.
- Las subclases no están limitadas al estado y comportamiento provisto por la superclase → pueden agregar variables y métodos además de los que ya heredan.
- Las clases hijas también pueden **sobreescribir** los métodos que heredan.

Agenda

~~Herencia~~

- ~~UML~~
- ~~Definición~~
- ~~Superclase y subclases~~
- ~~Ventajas~~

~~Herencia y Abstracción~~

Herencia en Java

Clase abstracta

- Definición
- Método abstracto
- Constructores



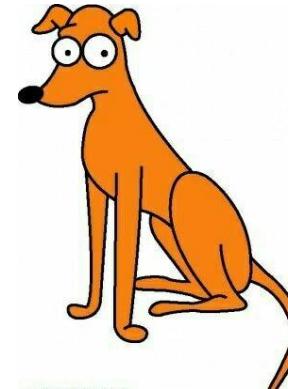
Herencia en Java

```
public abstract class Animal {  
  
    private String ubicacion;  
    ...  
  
    public abstract void hacerRuido();  
  
    public void comer() {  
        System.out.println("Soy un animal y estoy comiendo");  
    }  
    ...  
}
```

Herencia en Java (2)

```
public class Perro extends Animal {  
  
    @Override  
    public void hacerRuido() {  
        System.out.println("Guau!!");  
    }  
}
```

```
public class Gato extends Animal {  
  
    @Override  
    public void hacerRuido() {  
        System.out.println("Miau!!");  
    }  
}
```



Herencia en Java (3)

- También se puede heredar **sin tener métodos abstractos**.

```
public class Animal {  
  
    private String ubicacion;  
  
    ...  
  
    public void comer() {  
        System.out.println("Soy un animal y estoy comiendo");  
    }  
  
    ...  
}
```

Herencia en Java (4)

```
public class Perro extends Animal {  
    ...  
}  
  
public class Gato extends Animal {  
    @Override  
    public void comer() {  
        System.out.println("Soy un gato y estoy comiendo");  
    }  
}
```

Podemos sobreescibir los métodos necesarios en las subclases necesarias

Agenda

~~Herencia~~

- ~~UML~~
- ~~Definición~~
- ~~Superclase y subclases~~
- ~~Ventajas~~

~~Herencia y Abstracción~~

~~Herencia en Java~~

Clase abstracta

- Definición
- Método abstracto
- Constructores



Clase abstracta - Definición

- Es similar a una clase concreta: posee atributos y métodos pero tiene una condición:

- **Al menos uno de sus métodos debe ser abstracto.**

Agenda

- ~~Herencia~~
 - UML
 - Definición
 - Superclase y subclases
 - Ventajas
- ~~Herencia y Abstracción~~
- ~~Herencia en Java~~
- **Clase abstracta**
 - Definición
 - **Método abstracto**
 - Constructores



Clase abstracta - Método abstracto

- Un método abstracto se caracteriza por dos detalles:
 - Está precedido por la palabra clave **abstract**.
 - No tiene cuerpo y su encabezado termina con punto y coma.
- Si un método se declara como abstracto, se debe marcar la clase como abstracta → No puede haber métodos abstractos en una clase concreta.
- Los métodos abstractos **deben implementarse** en las clases concretas (subclases). → **@Override**



Agenda

- ~~Herencia~~
 - UML
 - Definición
 - Superclase y subclases
 - Ventajas
- ~~Herencia y Abstracción~~
- ~~Herencia en Java~~
- **Clase abstracta**
 - Definición
 - Método abstracto
 - **Constructores**



Clase abstracta - Constructores?

- Las clases abstractas no se pueden instanciar!!!

```
public abstract class Animal {
```

```
...
```

```
    public Animal() {  
    }
```

?

```
    public abstract void hacerRuido();
```

```
...
```

```
}
```

Clase abstracta - Constructores? (2)

~~Animal animal = new Animal();~~



- Es posible definir constructores en las superclases, pero no es posible crear instancias.

Clase abstracta - Constructores? (3)

```
public class Perro extends Animal {  
  
    private String nombre;  
  
    public Perro() {  
        super();  
    }  
  
    public Perro(String nombre) {  
        super();  
        this.nombre = nombre;  
    }  
    ...  
}
```

Agenda

~~Herencia~~

- ~~UML~~
- ~~Definición~~
- ~~Superclase y subclases~~
- ~~Ventajas~~

~~Herencia y Abstracción~~

~~Herencia en Java~~

~~Clase abstracta~~

- ~~Definición~~
- ~~Método abstracto~~
- ~~Constructores~~



Bibliografía oficial

- <https://docs.oracle.com/javase/tutorial/java/landl/subclasses.html>
- <https://docs.oracle.com/javase/tutorial/java/landl/abstract.html>

Clase 7: Herencia & Polimorfismo

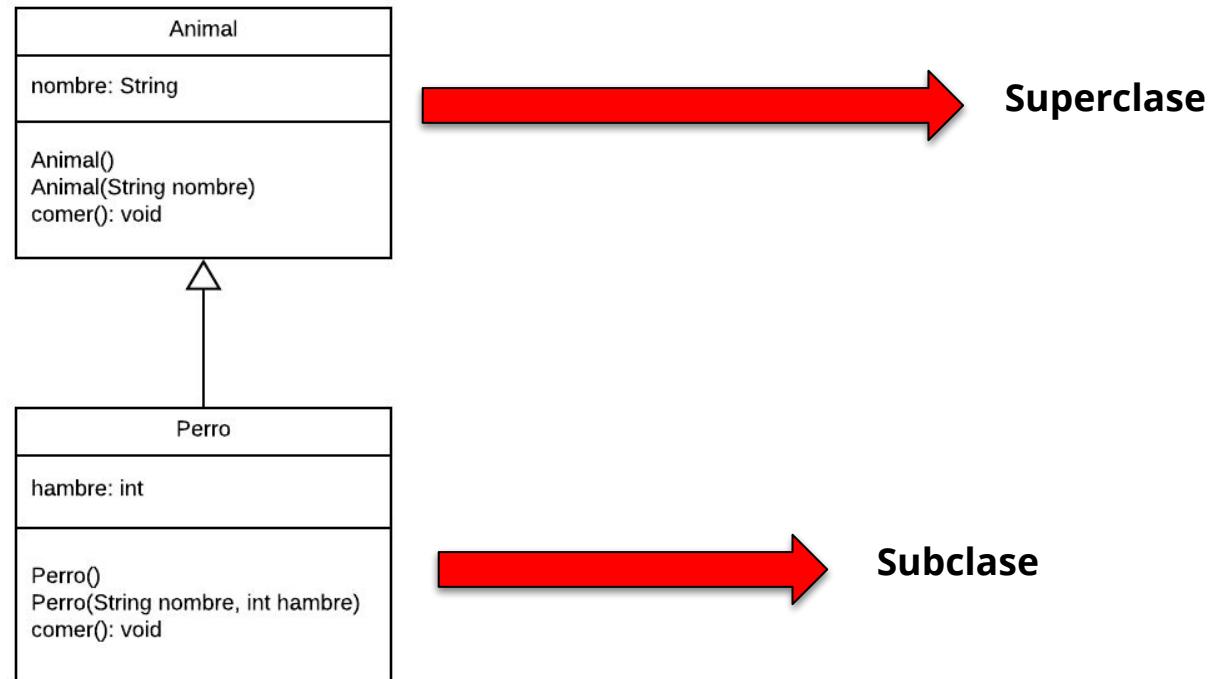
— Programación y Laboratorio III —

Agenda

- Ejemplo UML
- Palabra reservada super
 - Ejemplo
- Modificador de acceso: protected
- Ejecución dinámica de métodos
- Polimorfismo
 - Definición
 - Ejemplo
 - Warning
- UML - Ejemplo completo para practicar



Ejemplo UML



Agenda

~~Ejemplo UML~~

- Palabra reservada super
 - Ejemplo
 - Modificador de acceso: protected
 - Ejecución dinámica de métodos
 - Polimorfismo
 - Definición
 - Ejemplo
 - Warning
 - UML - Ejemplo completo para practicar



Palabra reservada: super

- Se utiliza para invocar **métodos de la superclase.**

```
super.metodo();
```

- En el caso de los constructores, debe ser la primer línea en el constructor de la subclase.

```
super();
```

ó

```
super(lista de parámetros);
```



Palabra reservada: super(2)

- Si el constructor en la subclase no invoca explícitamente al constructor de la superclase, el compilador de Java lo inserta automáticamente.

```
public Perro(String nombre) {  
    [espacio reservado para que el compilador agregue super(); ]  
    this.nombre = nombre;  
}
```



Agenda

~~Ejemplo UML~~

- Palabra reservada super
 - Ejemplo
- Modificador de acceso: protected
- Ejecución dinámica de métodos
- Polimorfismo
 - Definición
 - Ejemplo
 - Warning
- UML - Ejemplo completo para practicar



Ejemplo - Palabra reservada: super

```
public class Animal {
```

Nombre de la clase: primer letra mayúscula y en singular

```
//Variable de instancia
```

```
private String nombre;
```

```
//Constructor por defecto
```

```
public Animal() {  
}
```

Los constructores tienen el mismo nombre de la clase!!!

```
//Constructor con parámetros
```

```
public Animal(String nombre) {  
    this.nombre = nombre;  
}
```

```
//Método
```

```
public void comer() {  
    System.out.println("Estoy comiendo");  
}
```

Método concreto (no es abstracto)

```
}
```

Ejemplo - Palabra reservada: super (2)

```
public class Perro extends Animal {  
    //Variable de instancia  
    private int hambre;  
  
    //Constructor por defecto  
    public Perro() {  
    }  
  
    //Constructor con parámetros  
    public Perro(String nombre, int hambre) {  
        super(nombre);  
        this.hambre = hambre;  
    }  
  
    //Método sobreescrito  
    @Override  
    public void comer() {  
        super.comer();  
        hambre --;  
    }  
}
```

Hay una herencia, por lo tanto Perro es una subclase de Animal

Hereda la variable de instancia “nombre” y agrega una nueva

Llama al constructor de la superclase Animal

Se sobreescribe el método de la superclase

Se llama al método comer() de la superclase Animal

Agenda

~~Ejemplo UML~~

~~Palabra reservada super~~

— Ejemplo

- **Modificador de acceso: protected**
- Ejecución dinámica de métodos
- Polimorfismo
 - Definición
 - Ejemplo
 - Warning
- UML - Ejemplo completo para practicar



Modificador de acceso: protected

```
public class Animal {  
    //Nombre de la clase: primer letra mayúscula y en singular  
  
    //Variable de instancia  
    private String nombre;  
    //El modificador de acceso “private” indica que la variable de instancia  
    //“nombre” puede ser leída desde la clase Animal. El resto de las clases  
    //que quieran leerla deben acceder a través del getter.  
  
    //Constructor por defecto  
    public Animal() {  
    }  
    //Los constructores tienen el mismo nombre de la clase!!!  
  
    //Constructor con parámetros  
    public Animal(String nombre) {  
        this.nombre = nombre;  
    }  
  
    //Método  
    public String getNombre() {  
        //Método “getter” para leer desde otro objeto el nombre de la  
        //variable “nombre”  
        return nombre;  
    }  
}
```

Modificador de acceso: protected (2)

```
public class Perro extends Animal {  
  
    public void comer() {  
        System.out.println("Me llamo " + this.getNombre() + " y estoy comiendo");  
    }  
}
```

Hay una herencia, por lo tanto Perro es una subclase de Animal

Se lee la variable de instancia "nombre" a través del método getter

Modificador de acceso: protected (3)

```
public class Animal {
```

Nombre de la clase: primer letra mayúscula y en singular

```
//Variable de instancia
```

```
protected String nombre;
```

El modificador de acceso “protected” indica que la variable de instancia “nombre” puede ser leída desde las subclases.

```
//Constructor por defecto
```

```
public Animal() {  
}
```

Los constructores tienen el mismo nombre de la clase!!!

```
//Constructor con parámetros
```

```
public Animal(String nombre) {  
    this.nombre = nombre;  
}
```

```
//Método
```

```
public String getNombre() {  
    return nombre;  
}
```

```
}
```

Modificador de acceso: protected (4)

```
public class Perro extends Animal {  
  
    public void comer() {  
        System.out.println("Me llamo " + nombre + " y estoy comiendo");  
    }  
  
}
```

Hay una herencia, por lo tanto Perro es una subclase de Animal

Se lee la variable de instancia "nombre" sin necesidad de acceder por el método getter

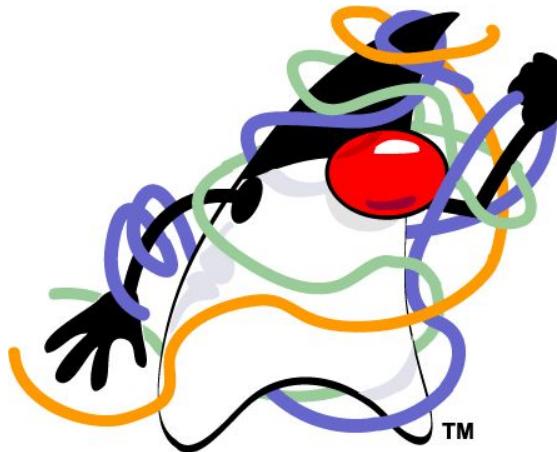
Agenda

- ~~Ejemplo UML~~
- ~~Palabra reservada super~~
 - Ejemplo
- ~~Modificador de acceso: protected~~
 - **Ejecución dinámica de métodos**
 - Polimorfismo
 - Definición
 - Ejemplo
 - Warning
 - UML - Ejemplo completo para practicar

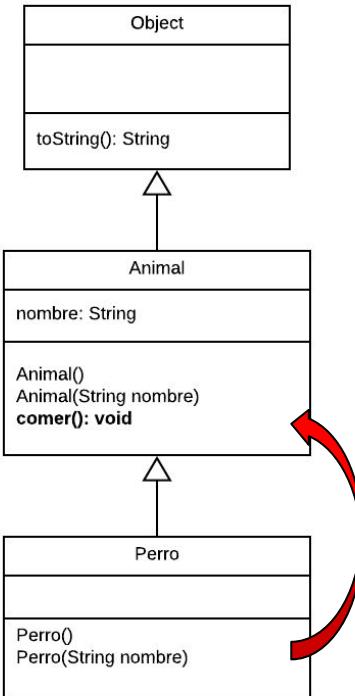


Ejecución dinámica de métodos

- Los métodos sobrescritos de las subclases tienen precedencia sobre los métodos de las subclases.
- La búsqueda del método comienza al final de la jerarquía, entonces la última redefinición de un método es la que se ejecuta primero.



Ejecución dinámica de métodos (2)



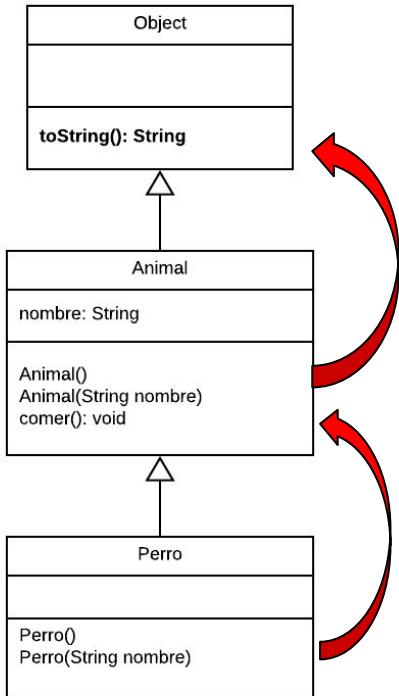
Ejemplo 1: Se quiere ejecutar el método `comer()` de la clase `Perro`.

```
public static void main(String [] args) {
    Perro perro = new Perro("Ayudante de Santa");
    perro.comer();
}
```

Para acceder a los métodos de la clase `Perro`, es necesario crear una instancia.

- 1) Se busca el método **comer()** en la subclase `Perro`.
- 2) Como no se encuentra la sobrescritura, se sube en la jerarquía hasta que se encuentra y se ejecuta.

Ejecución dinámica de métodos (3)



Ejemplo 2: Se quiere ejecutar el método `toString()` de la clase `Perro`.

```
public static void main(String [] args) {
    Perro perro = new Perro("Ayudante de Santa");
    System.out.println(perro.toString());
}
```

Para acceder a los métodos de la clase `Perro`, es necesario crear una instancia.

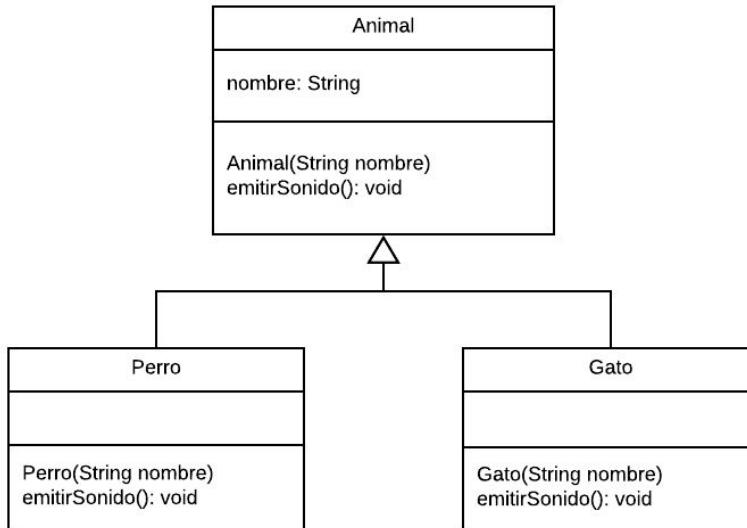
- 1) Se busca el método **`toString()`** en la subclase `Perro`.
- 2) Como no se encuentra la sobreescritura, se sube en la jerarquía hasta que se encuentra y se ejecuta.

Agenda

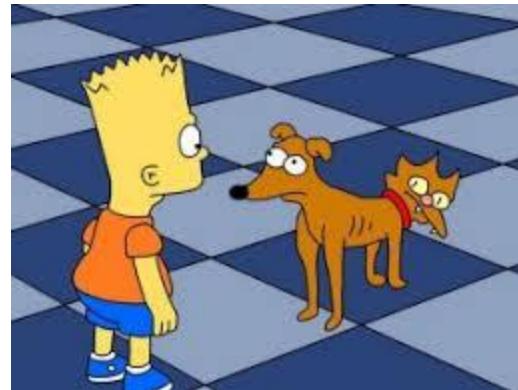
- ~~Ejemplo UML~~
- ~~Palabra reservada super~~
 - Ejemplo
- ~~Modificador de acceso: protected~~
- ~~Ejecución dinámica de métodos~~
- **Polimorfismo**
 - Definición
 - Ejemplo
 - Warning
- UML - Ejemplo completo para practicar



Polimorfismo - Definición



- Se refiere a la propiedad por la que es posible enviar mensajes sintácticamente iguales a objetos de tipos distintos.
- Está ligado a la herencia.



Agenda

- ~~Ejemplo UML~~
- ~~Palabra reservada super~~
 - Ejemplo
- ~~Modificador de acceso: protected~~
- ~~Ejecución dinámica de métodos~~
- **Polimorfismo**
 - Definición
 - **Ejemplo**
 - Warning
- UML - Ejemplo completo para practicar



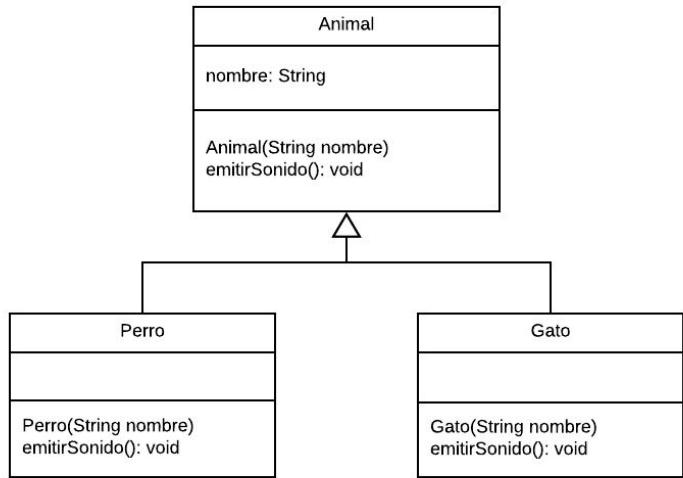
Polimorfismo - Ejemplo

```
public abstract class Animal {  
  
    //Variable de instancia  
    protected String nombre;  
  
    //Constructor con parámetros  
    public Animal(String nombre) {  
        this.nombre = nombre;  
    }  
  
    public abstract void emitirSonido();  
  
}
```

```
public class Perro extends Animal {  
    @Override  
    public void emitirSonido() {  
        System.out.println("Guau!");  
    }  
}
```

```
public class Gato extends Animal {  
    @Override  
    public void emitirSonido() {  
        System.out.println("Miau!");  
    }  
}
```

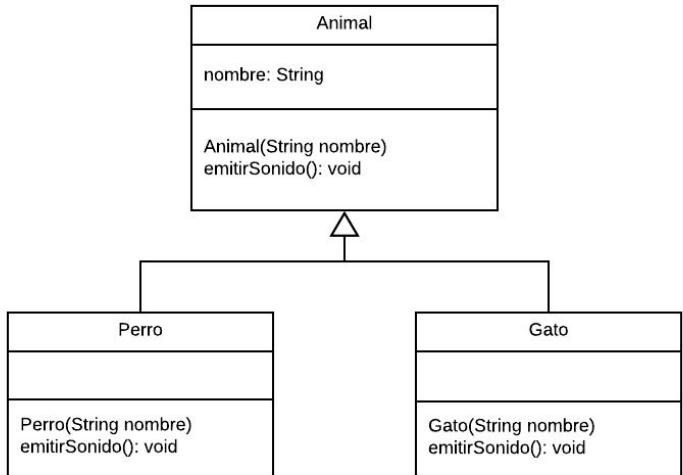
Polimorfismo - Ejemplo (2)



```
public static void main(String [] args) {
    Perro [] perros = new Perro[2];
    Perro perro1 = new Perro("Ayudante de Santa");
    Perro perro2 = new Perro("Procer");
    perros[0] = perro1;
    perros[1] = perro2;
    for (int i = 0; i < perros.length; i++) {
        System.out.println(perros[i].emitirSonido());
    }
}
```

Se invoca al método de la subclase Perro

Polimorfismo - Ejemplo (3)



```
public static void main(String [] args) {
    Animal [] animales = new Animal[2];
    Perro perro = new Perro("Ayudante de Santa");
    Gato gato = new Gato("Bola de nieve I");
    animales[0] = perro;
    animales[1] = gato;
    for (int i = 0; i < animales.length; i++) {
        System.out.println(animales[i].emitirSonido());
    }
}
```

Se invoca al método de la subclase Gato cuando animales[1].emitirSonido();

Se invoca al método de la subclase Perro cuando animales[0].emitirSonido();

Agenda

- ~~Ejemplo UML~~
- ~~Palabra reservada super~~
 - Ejemplo
- ~~Modificador de acceso: protected~~
- ~~Ejecución dinámica de métodos~~
- **Polimorfismo**
 - Definición
 - Ejemplo
 - Warning
- UML - Ejemplo completo para practicar



Polimorfismo - Warning!

- Los métodos que se pueden invocar son los que contiene la clase del tipo declarado.

Se pueden crear instancias de cualquiera de las subclases.

```
Animal perro = new Perro("Ayudante de Santa");
```

Se pueden invocar métodos de la clase Animal.

- Si hay métodos declarados en la subclase (Perro) que no pertenecen a la superclase (Animal), no se pueden invocar.

Ejemplo - Polimorfismo Warning!

```
public abstract class Animal {  
  
    //Variable de instancia  
    protected String nombre;  
  
    //Constructor con parámetros  
    public Animal(String nombre) {  
        this.nombre = nombre;  
    }  
  
    public abstract void emitirSonido();  
}
```

```
public class Perro extends Animal {  
  
    @Override  
    public void emitirSonido() {  
        System.out.println("Guau!");  
    }  
  
    public void agarrarPelota() {  
        System.out.println("Agarré la pelota");  
    }  
}
```

Comportamiento definido para la clase Perro. No existe en la clase Animal.

Ejemplo - Polimorfismo Warning! (2)

```
Animal perro = new Perro("Ayudante de Santa");
```

```
perro.agarrarPelota();
```



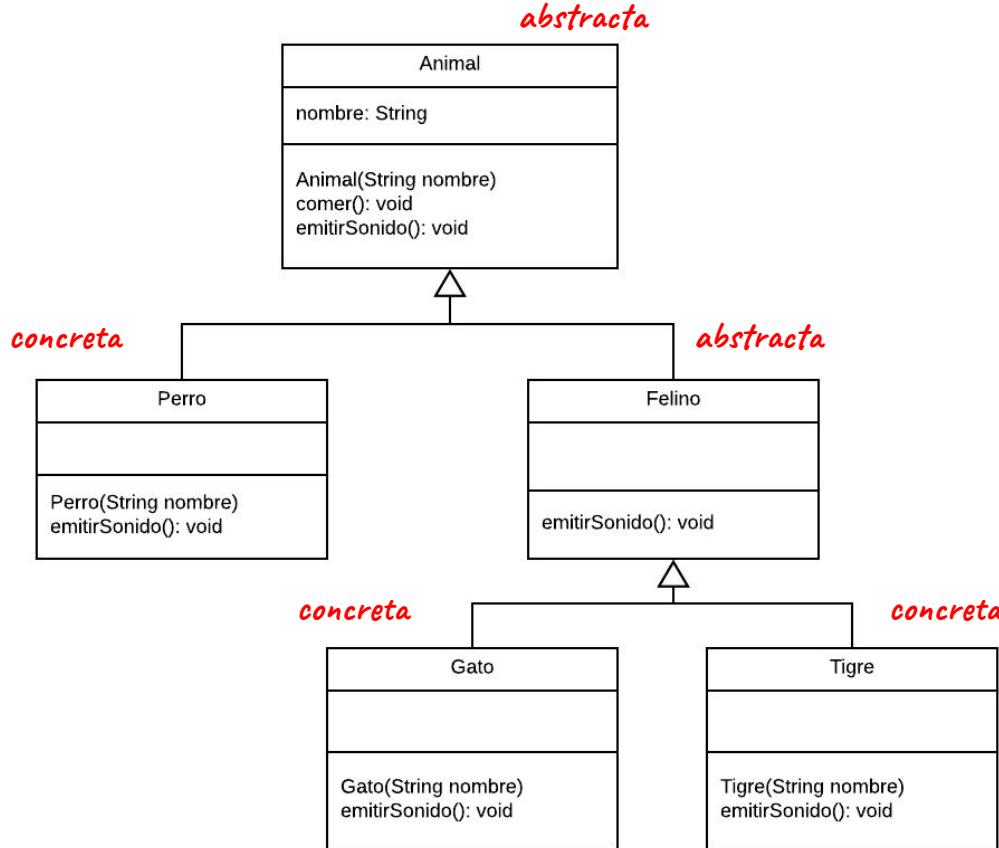
El método no está definido en la clase Animal!!

Agenda

- ~~Ejemplo UML~~
- ~~Palabra reservada super~~
 - Ejemplo
- ~~Modificador de acceso: protected~~
- ~~Ejecución dinámica de métodos~~
- ~~Polimorfismo~~
 - Definición
 - Ejemplo
 - Warning
- **UML - Ejemplo completo para practicar**



UML - Ejemplo completo



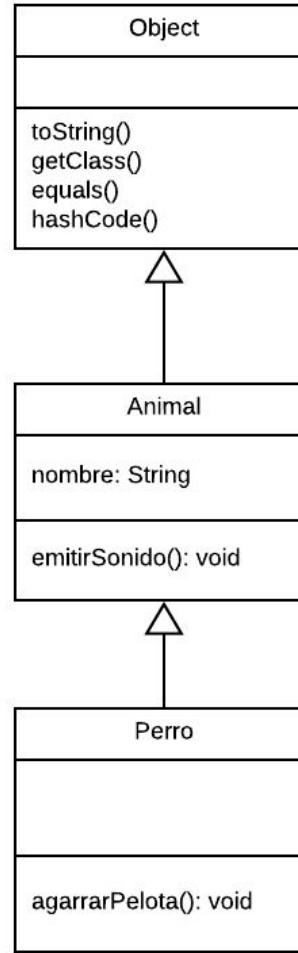
Bibliografía oficial

- <https://docs.oracle.com/javase/tutorial/java/IandI/super.html>
- <https://docs.oracle.com/javase/tutorial/java/IandI/objectclass.html>
- <https://docs.oracle.com/javase/tutorial/java/IandI/polymorphism.html>

Clase 8 - Polimorfismo & Casting

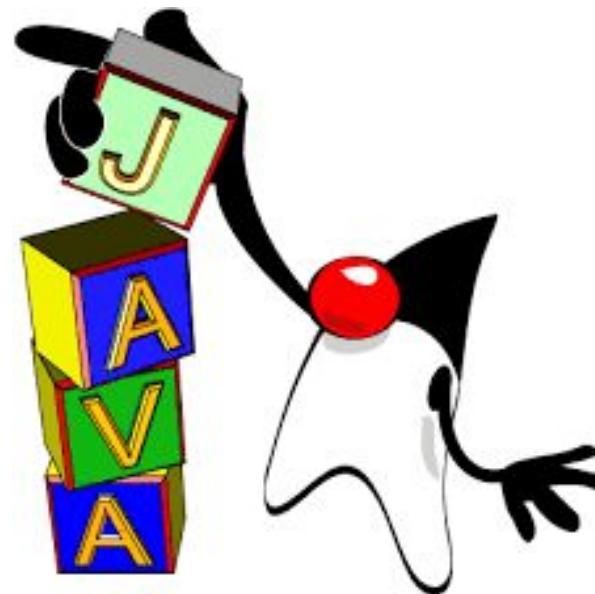
— Programación & Laboratorio III —

UML - Caso de ejemplo



Agenda

- Polimorfismo con Object
- Referencias polimórficas
- Casting



Polimorfismo con Object - Ejemplo

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

```
    Perro [] perros = new Perro[2];
```

El arreglo puede contener sólo instancias de Perro porque el tipo declarado es Perro

```
    Perro perro1 = new Perro("Ayudante de Santa");
```

```
    Perro perro2 = new Perro("Procer");
```

Se crean dos INSTANCIAS de la clase Perro

```
    perros[0] = perro1;  
    perros[1] = perro2;
```

Se asignan las instancias creadas anteriormente en las posiciones 0 y 1 respectivamente

```
    for (int i = 0; i < perros.length; i++) {  
        System.out.println(perros[i].emitirSonido());
```

```
}
```

```
}
```

Accedemos a métodos de la clase Perro porque sabemos que tenemos INSTANCIAS de la clase Perro almacenadas en el arreglo

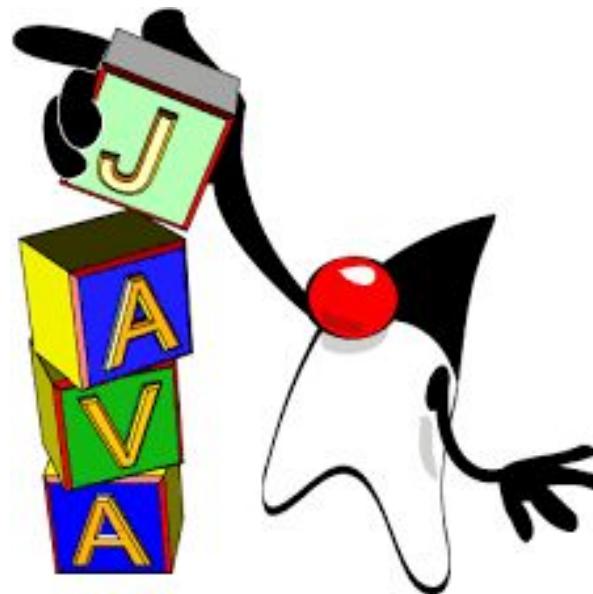
Polimorfismo con Object - Ejemplo (2)

```
public static void main(String [] args) {  
  
    Object [] objects = new Object[2];      El arreglo puede contener instancias de Object y cualquiera de las  
    subclases en la jerarquía  
  
    Object object = new Object();          Se crean dos INSTANCIAS: una de la clase Object y otra  
    de la clase Perro  
    Perro perro = new Perro("Ayudante de Santa");  
  
    objects[0] = object;  
    objects[1] = perro;  
  
    for (int i = 0; i < objects.length; i++) {  
        System.out.println(objects[i].emitirSonido());  
    }  
}
```

**No compila! Pero cómo sabemos que estamos leyendo una
instancia de Perro?**

Agenda

- ~~Polimorfismo con Object~~
- Referencias polimórficas
- Casting



Referencias polimórficas

- El problema de tener todo polimórficamente como Object es que lo que queremos almacenar tiene a perder su verdadera esencia.

```
Object [] objects = new Object[2];
Object object = new Object();
Perro perro = new Perro("Ayudante de Santa");
```

```
objects[0] = object;
objects[1] = perro;
```

```
Object o = objects[0];
int code = o.hashCode();
```

Operación válida

```
o.emitirSonido();
```

Error en tiempo de compilación porque no es un método de la clase Object

Referencias polimórficas (2)

```
Object [] objects = new Object[2];
Object object = new Object();
Perro perro = new Perro("Ayudante de Santa");

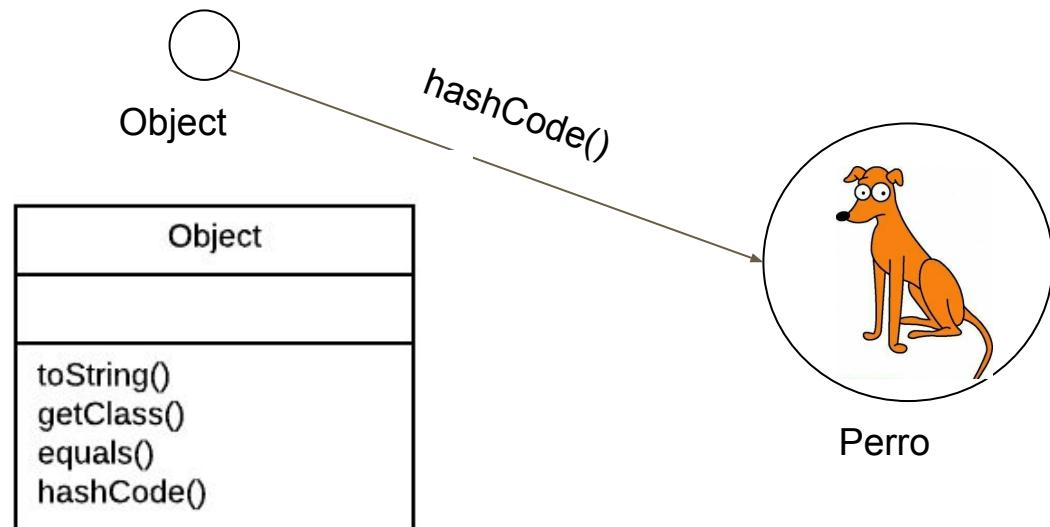
objects[0] = object;
objects[1] = perro;

Object o = objects[1];    int code = o.hashCode();
```

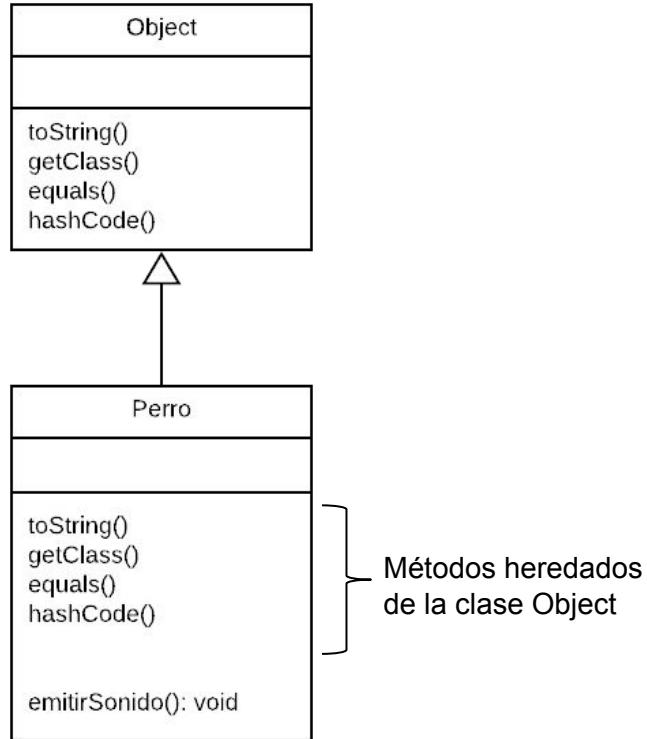
Retorna una referencia de Object de una instancia de Perro

Referencias polimórficas (3)

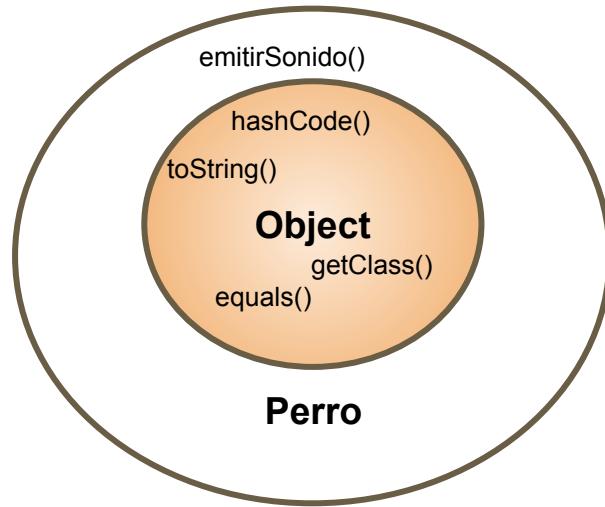
- El método que se invoca debe estar en la clase del tipo declarado, sin importar el tipo del que realmente es el objeto.
- El compilador verifica el tipo de la referencia, no el tipo del objeto.



Referencias polimórficas (4)



objeto Perro

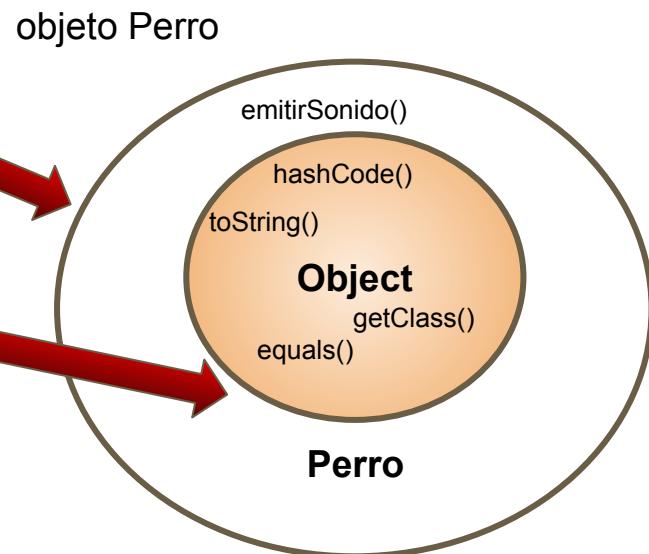


- El objeto Perro contiene la clase **Perro** como parte suya y también la clase **Object**.

Referencias polimórficas (5)

```
Perro p = new Perro("Ayudante de Santa");
```

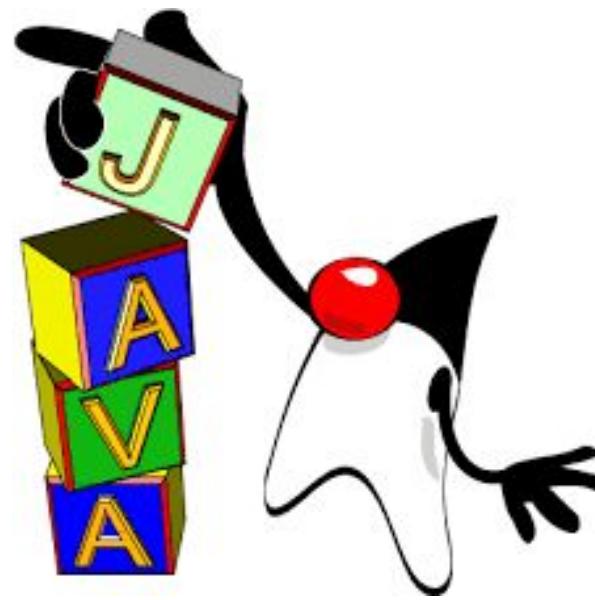
```
Object o = p;
```



- La referencia a Object sólo tiene acceso a los métodos de Object.
- La referencia a Perro tiene acceso a los métodos de Perro y de Object.

Agenda

- ~~Polimorfismo con Object~~
- ~~Referencias polimórficas~~
- Casting



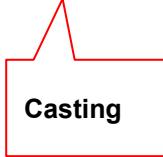
Casting

- Se denomina proceso de casting a la “conversión” de una referencia de un tipo a otro.

```
Object [] objects = new Object[2];  
Perro perro = new Perro("Ayudante de Santa");
```

```
objects[0] = perro;
```

```
Perro p = (Perro) objects[0];
```



Casting

Casting (2)

- IMPORTANTE: Debemos estar seguros de que lo que estamos “casteando” pertenece a la clase de conversión.

```
public static void main(String [] args) {  
  
    Object [] objects = new Object[2];  
    Perro perro = new Perro("Ayudante de Santa");  
  
    objects[0] = perro;  
  
    Gato g = (Gato) objects[0];  
}
```



ERROR EN TIEMPO DE EJECUCIÓN: Exception in thread "main"
java.lang.ClassCastException: Perro cannot be cast to Gato

Casting (3)

- Para asegurarnos del casting correcto, se puede utilizar el operador instanceof.

```
public static void main(String [] args) {  
  
    Object [] objects = new Object[2];  
    Perro perro = new Perro("Ayudante de Santa");  
  
    objects[0] = perro;  
  
    if (objects[0] instanceof Gato) {  
        Gato g = (Gato) objects[0];  
    }  
}
```

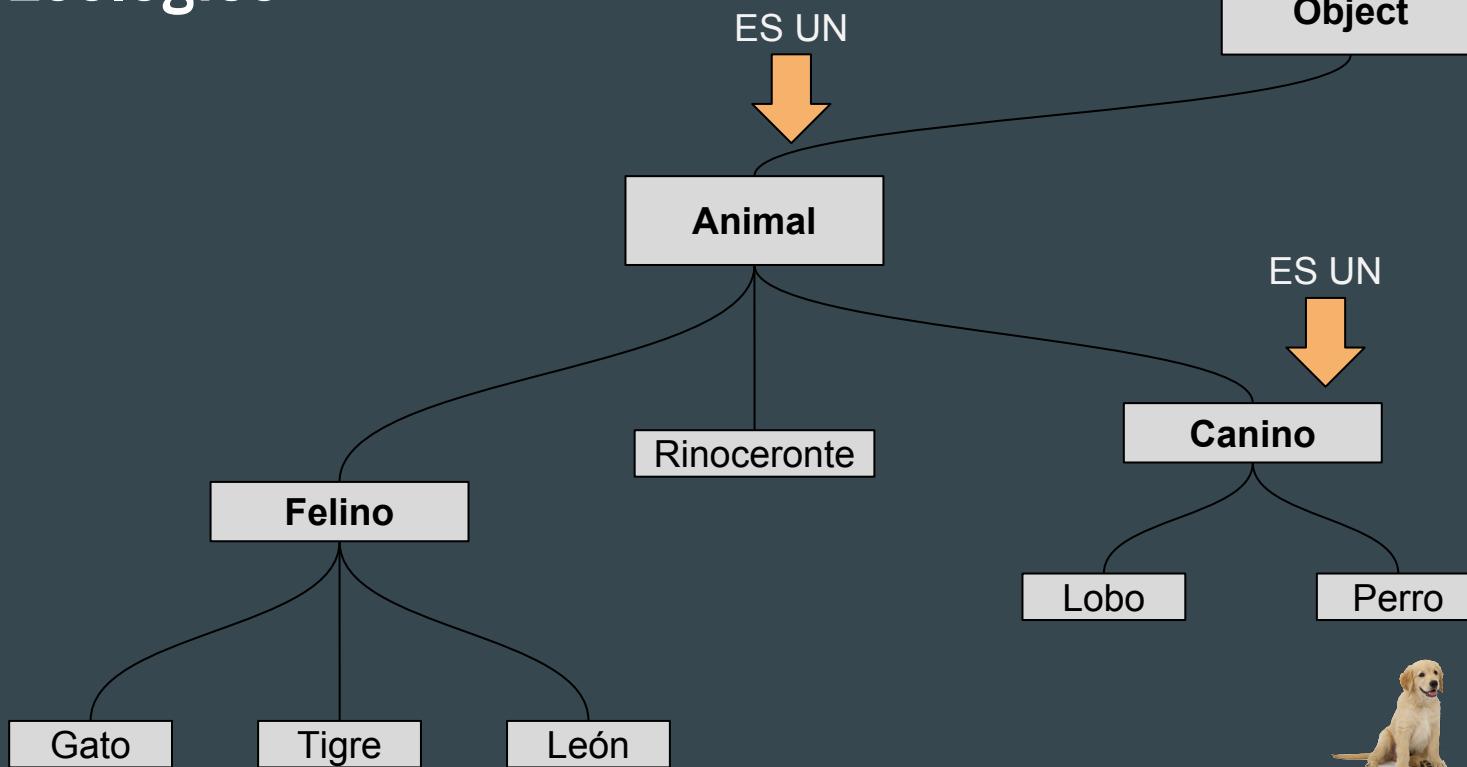
Bibliografía oficial

- <https://docs.oracle.com/javase/specs/jls/se7/html/jls-5.html>
- <https://docs.oracle.com/javase/7/docs/api/java/lang/ClassCastException.html>
- https://docs.oracle.com/cd/E29028_01/wlp.1034/e14255/com/bea/p13n/expression/operator/Instanceof.html
- <https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html>

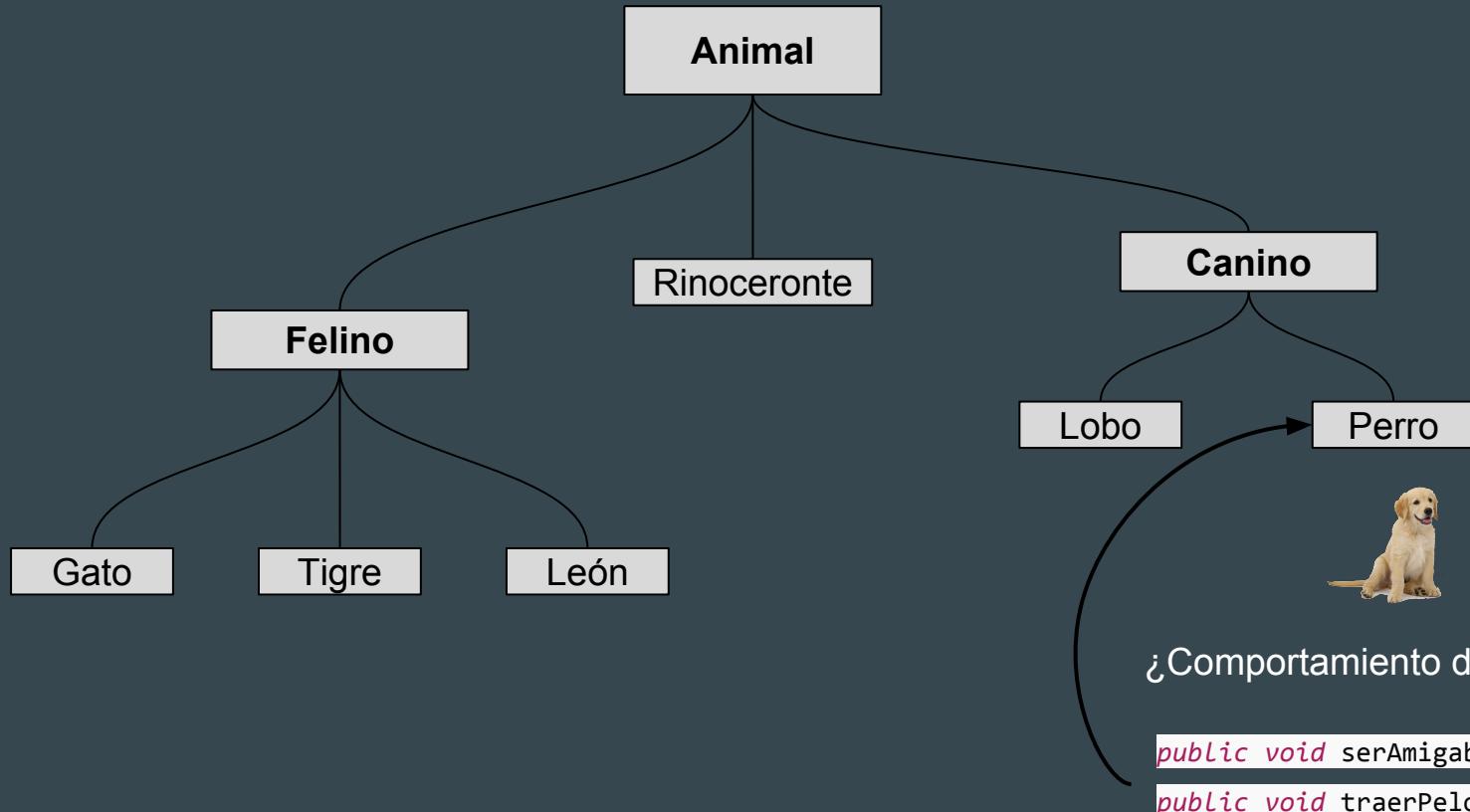
Interfaces

• • •

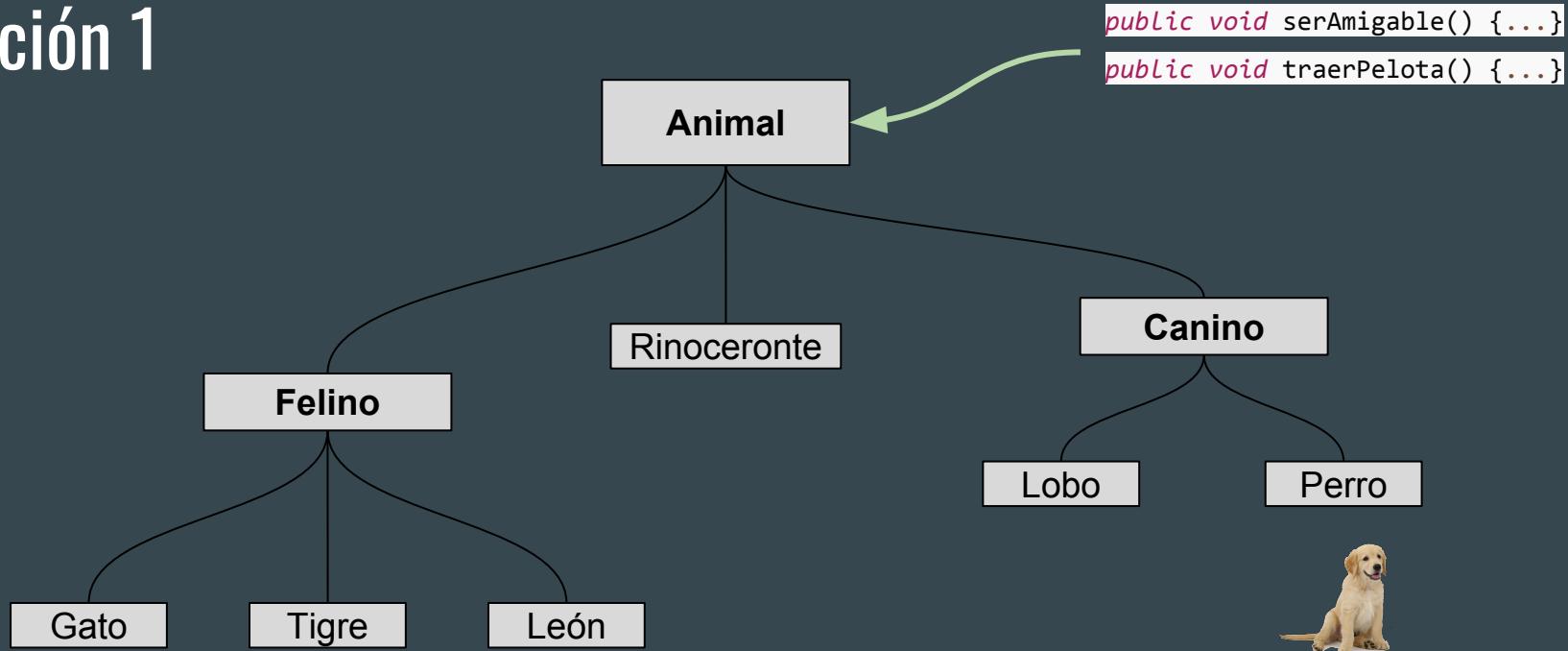
UML Zoológico



UML Veterinaria



Solución 1



Pros:

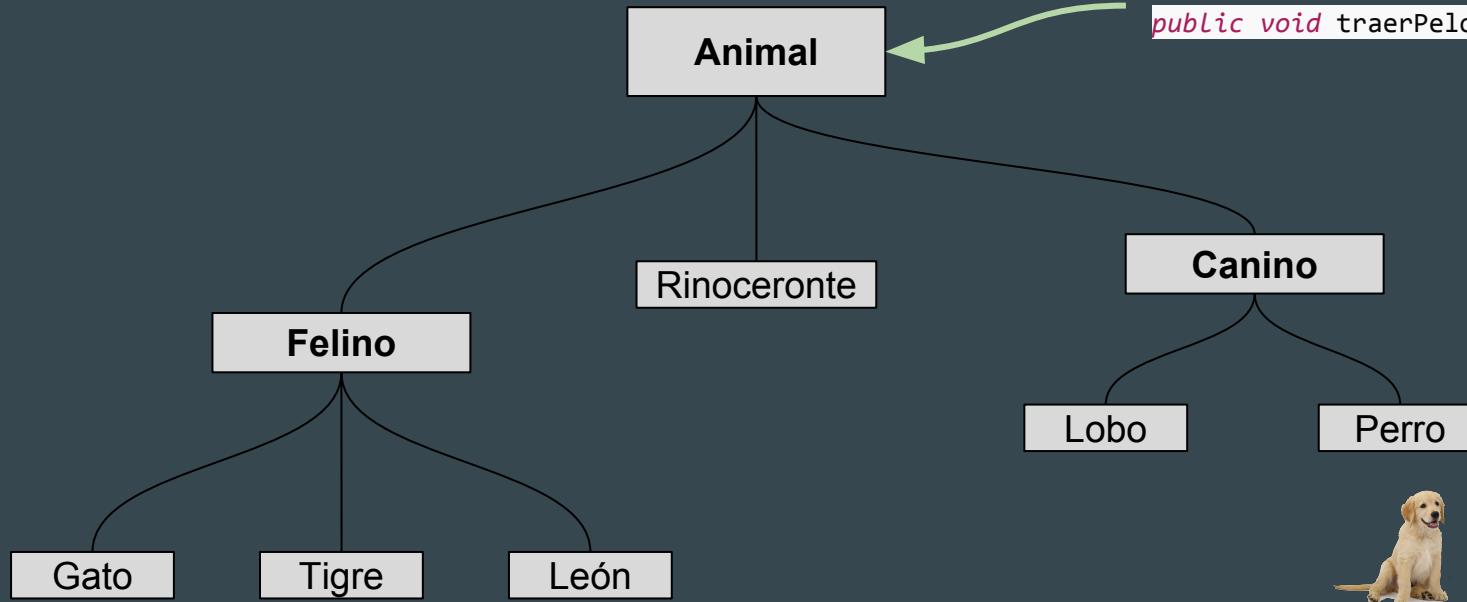
- Todos los animales heredan el comportamiento de mascota.
- No hay que tocar el comportamiento de las subclases existentes.
- Las futuras clases implementadas podrán hacer uso de los nuevos métodos.
- **Animal** puede ser utilizado como tipo polimórfico.





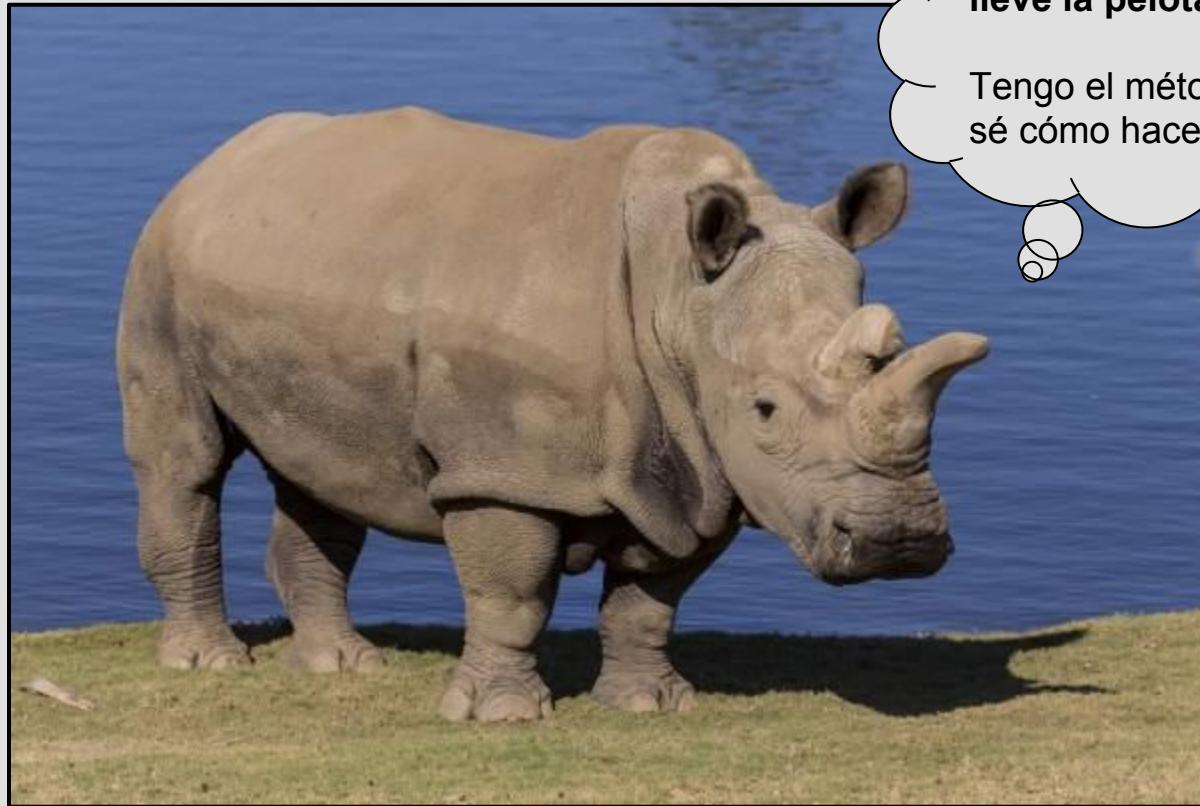
Solución 2

¿Métodos abstractos en la superclase?



Pros:

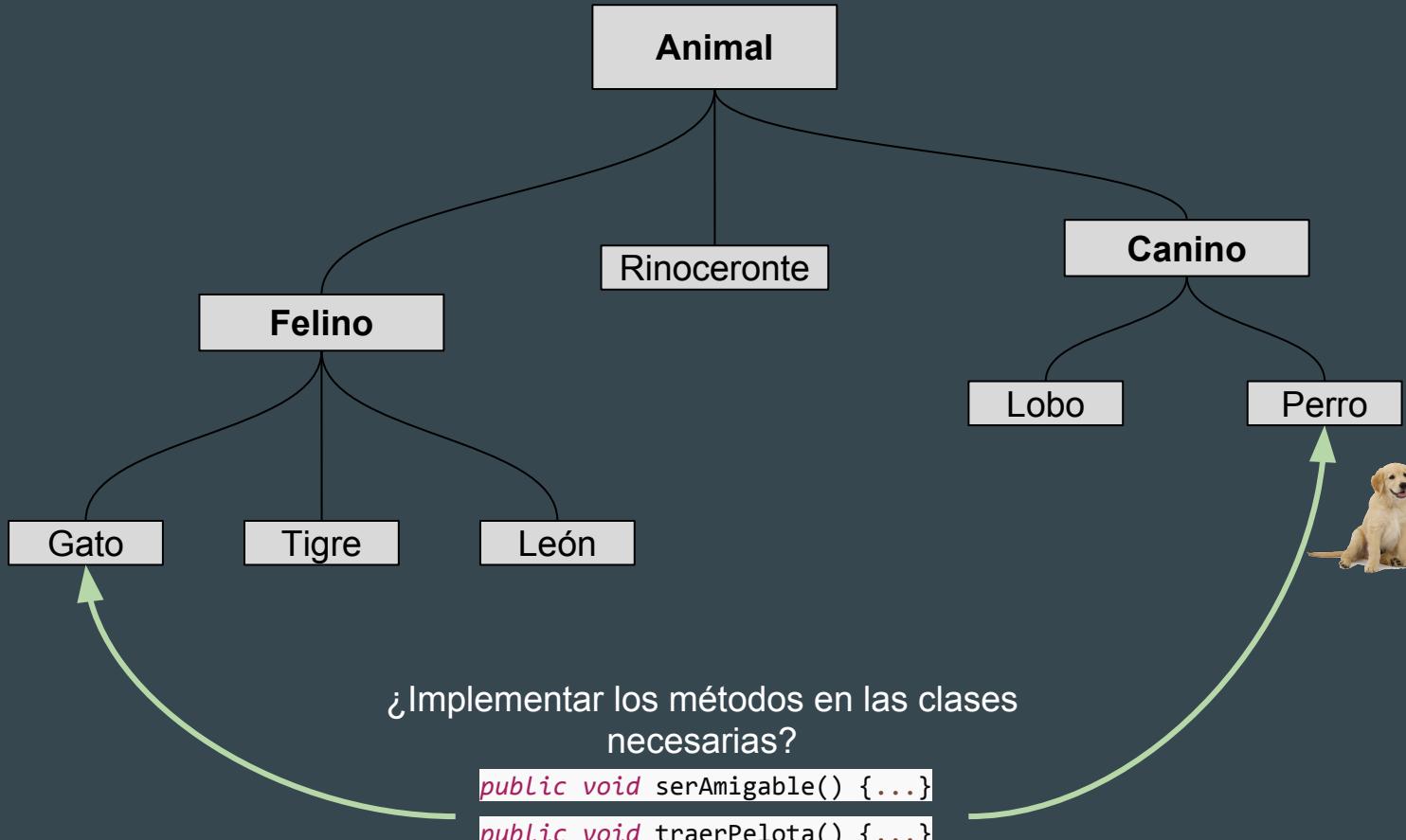
- Todos los beneficios de la solución anterior.
- Se pueden implementar métodos que no hagan nada para los animales que no se consideran mascotas.



.- ¿Querés que
llevé la pelota?

Tengo el método,
sé cómo hacerlo...

Solución 3



Pros:

- No nos tenemos que preocupar por los rinocerontes o los lobos.
- Los métodos están en las clases a las que pertenecen.
- El gato y el perro implementan su comportamiento sin que los demás animales se enteren.

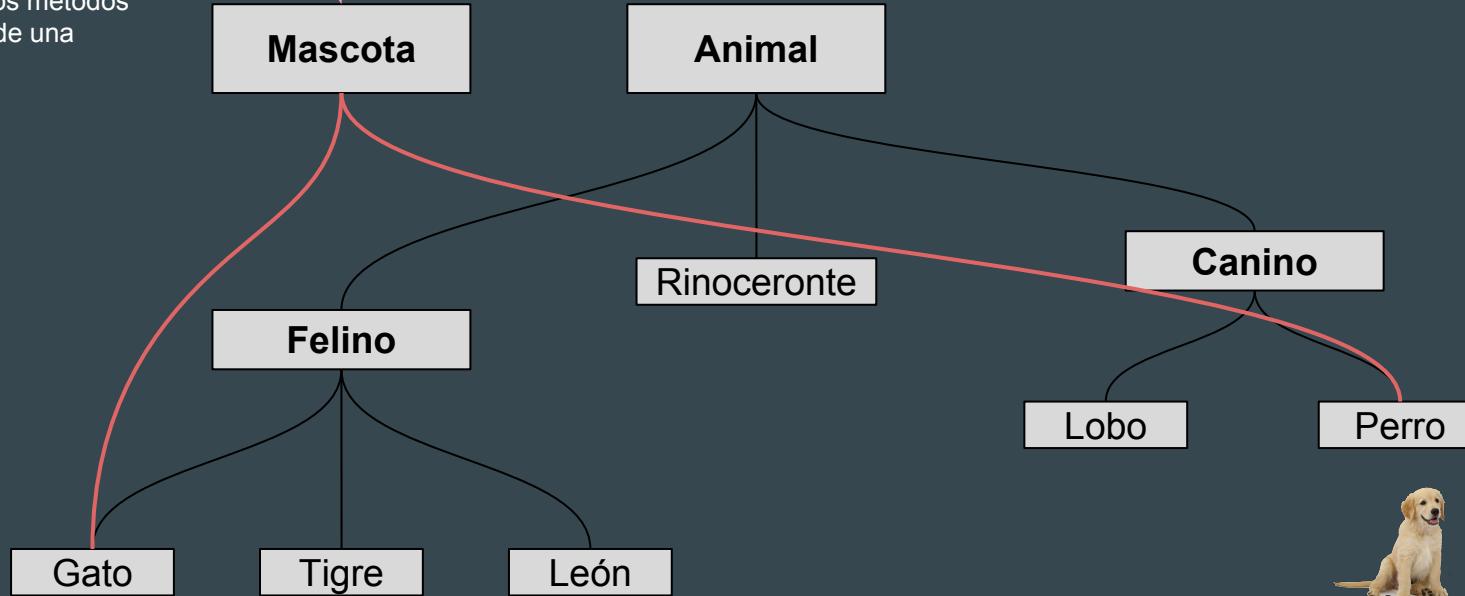
Contras:

- Hay que definir un contrato sobre las acciones que una mascota puede realizar.
- No estamos usando polimorfismo. No podemos usar Animal como tipo polimórfico. Porque no vamos a poder llamar un método de una mascota en un Animal.

¿Y ahora?

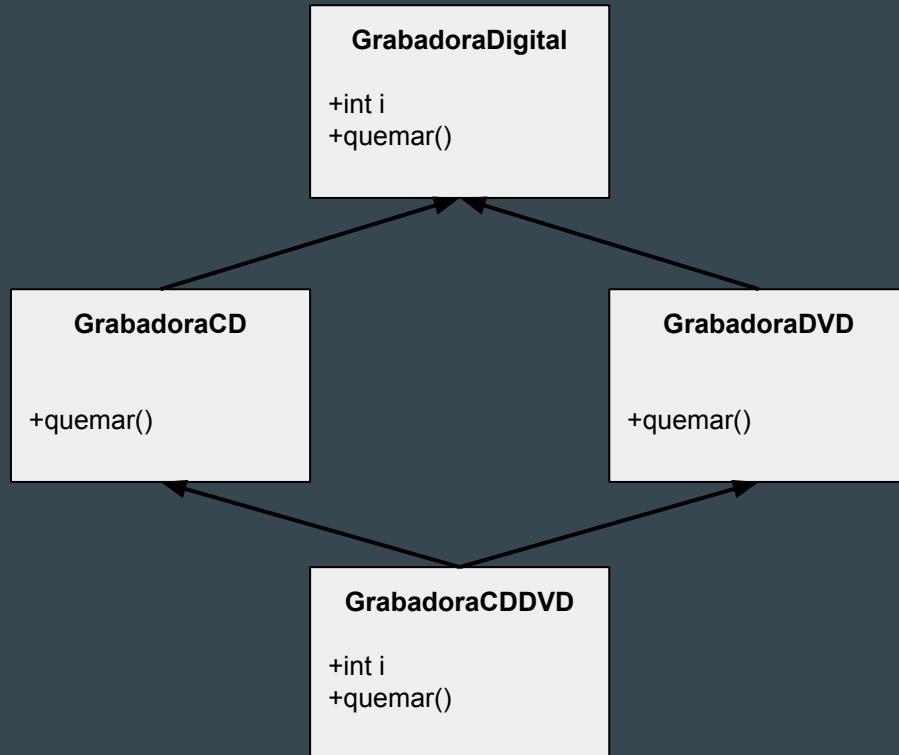
- Una forma de tener el comportamiento de mascotas solo en las clases necesarias.
- Una forma de garantizar que todos los métodos definidos poseen la misma signatura, un “contrato”.
- Una forma de tomar ventaja del polimorfismo.

Una nueva superclase abstracta, Mascota.
Con todos los métodos necesarios de una mascota?

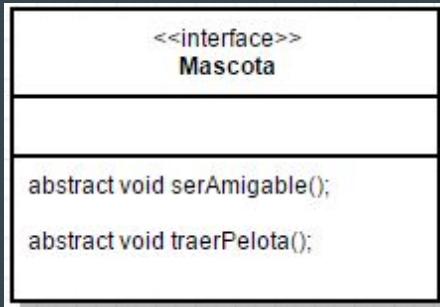


¿Herencia múltiple?

Deadly Diamond of Death (DDD)



Interfaces!!



```
package com.utn.learning.interfaces;

public interface Mascota {

    abstract void serAmigable();
    abstract void traerPelota();
}
```

Todos los métodos dentro de una interfaz son públicos y abstractos por defecto.

De esta forma la clase que implemente dicha interfaz **DEBE** implementar los métodos.

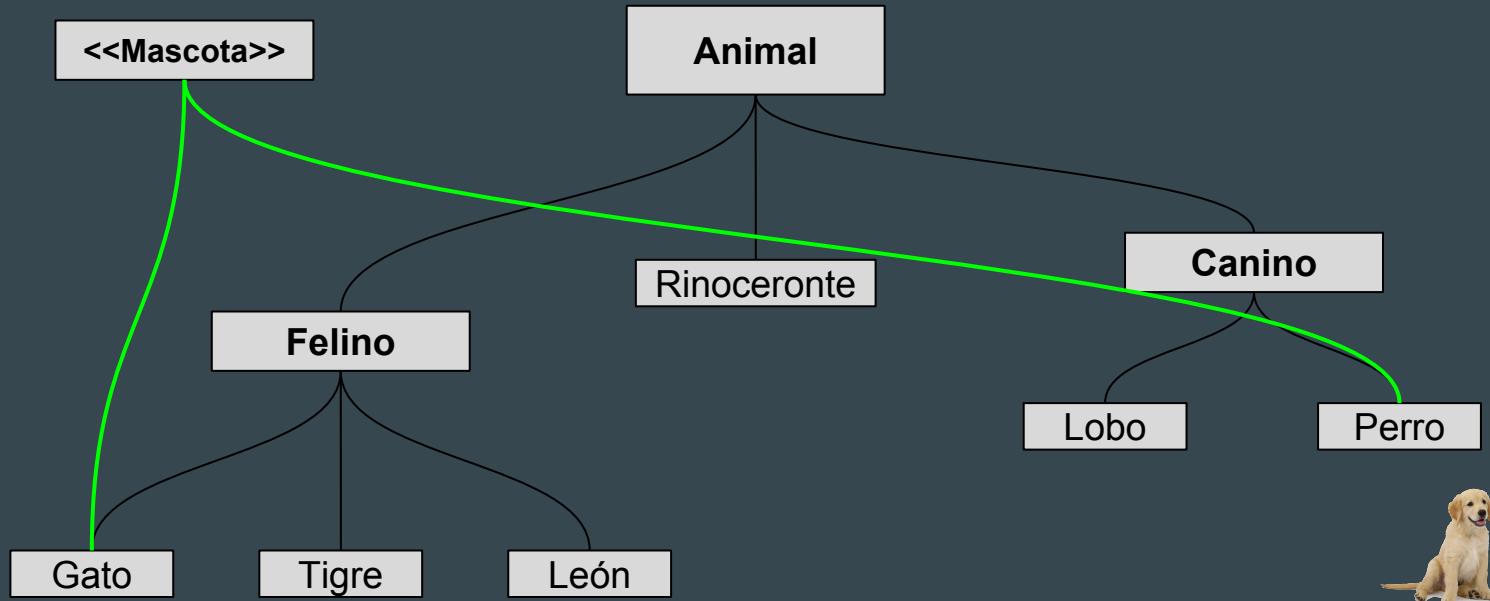
```
package com.utn.learning.interfaces;

public class Perro extends Canino implements Mascota {

    @Override
    public abstract void serAmigable() {...};

    @Override
    public abstract void traerPelota() {...};
}
```

Diagrama con Interfaz



Consideraciones

```
public interface DoIt {  
    void doSomething(int i, double x);  
    int doSomethingElse(String s);  
}
```

Nuevo requerimiento en DoIt:

```
public interface DoIt {  
    void doSomething(int i, double x);  
    int doSomethingElse(String s);  
    boolean didItWork(int i, double x, String s);  
}
```

Extensión de interfaces:

```
public interface DoItPlus extends DoIt {  
    boolean didItWork(int i, double x, String s);  
}
```

Método default, java 8 en adelante:

```
public interface DoIt {  
    void doSomething(int i, double x);  
    int doSomethingElse(String s);  
    default boolean didItWork(int i, double x, String s) {  
        // Method body  
    }  
}
```

Ejemplo

```
public interface Comparable {  
  
    // this (el objeto que llama a esMayorQue)  
    // y otro deben ser instancias de la misma  
    // clase devuelve 1, 0, -1 si this  
    // es mayor que, igual a, o menor a otro.  
  
    public int esMayorQue(Comparable otro);  
}
```

```
public Object encontrarMayor(Object o1, Object o2) {  
    Comparable obj1 = (Comparable) o1;  
    Comparable obj2 = (Comparable) o2;  
  
    if ((obj1).esMayorQue(obj2) > 0)  
        return obj1;  
    else  
        return obj2;  
}
```

```
public class Rectangulo implements Comparable {  
    public int base = 0;  
    public int altura = 0;  
  
    public int getArea() {  
        return base * altura;  
    }  
  
    // Método requerido de la interface Comparable  
    public int esMayorQue(Relatable otro) {  
        Rectangulo otroRect = (Rectangulo) otro;  
  
        if (this.getArea() < otroRect.getArea())  
            return -1;  
        else if (this.getArea() > otroRect.getArea())  
            return 1;  
        else  
            return 0;  
    }  
}
```

Clase 10: Collections - Listas

— Programación & Laboratorio III —

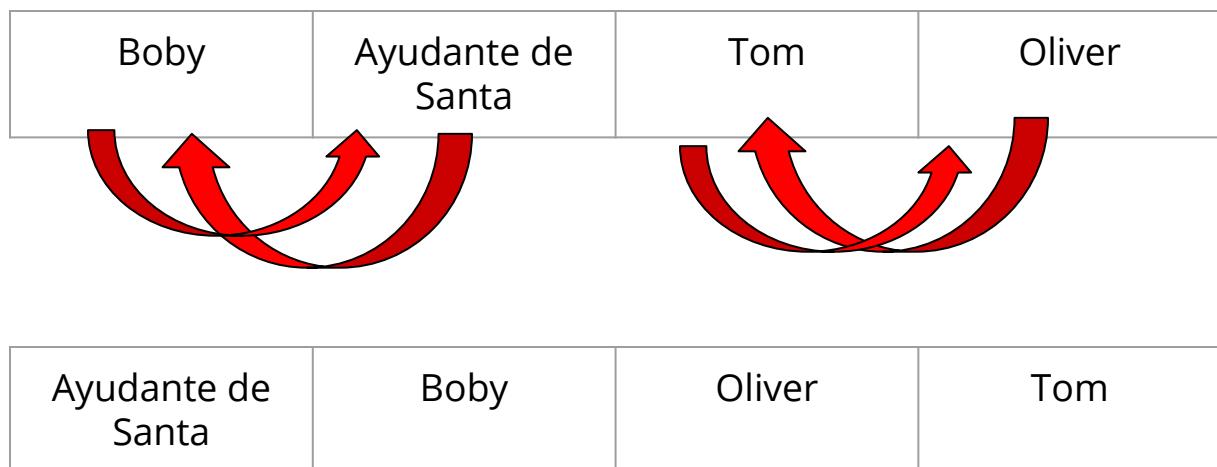
Ejemplo

```
public static void main(String [] args) {  
  
    Perro [] perros = new Perro[4];  
  
    Perro p1 = new Perro("Boby");  
    Perro p2 = new Perro("Ayudante de Santa");  
    Perro p3 = new Perro("Tom");  
    Perro p4 = new Perro("Oliver");  
  
    perros[0] = p1;  
    perros[1] = p2;  
    perros[2] = p3;  
    perros[3] = p4;  
  
}
```

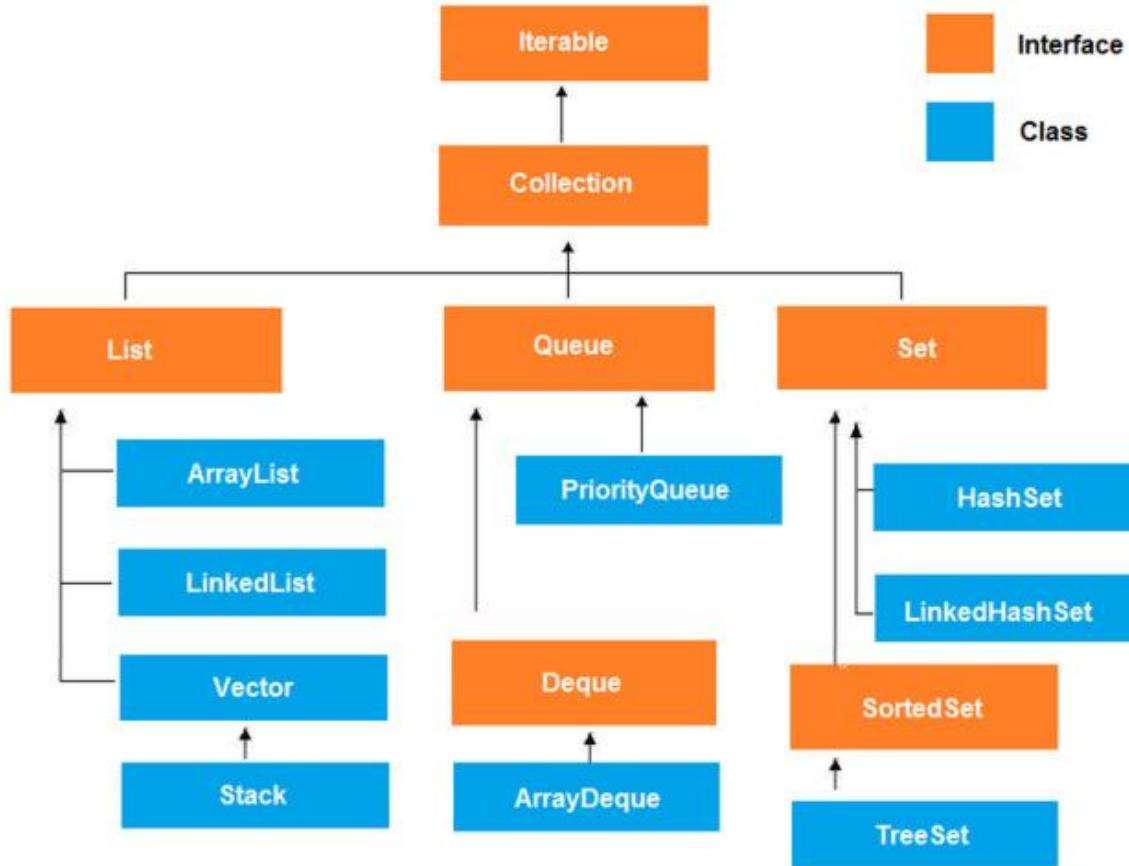
PREVIOUSLY ON...

Problema: Ordenar el arreglo?

- Posible solución:
 - Buscar algoritmos de ordenamiento que utilicen variables auxiliares e implementarlos.

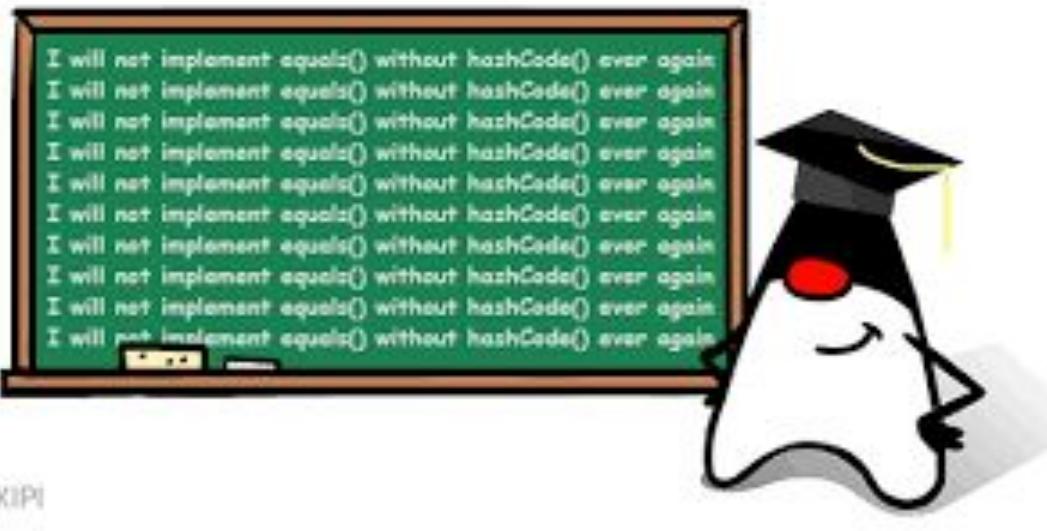


Solución: Collection API



Agenda

- **Collection API**
- Interfaz Collection
- List
- ArrayList
 - Insertar elemento
 - Lectura de un elemento
 - Eliminar elemento
- Método equals()
- Método hashCode()
- Método equals() - Ejemplo



Collection API

- Una Collection es un objeto que representa un grupo de objetos.
- El framework Collection es una arquitectura unificada para representar y manipular colecciones, lo que permite manipularlas independientemente de los detalles de la implementación.

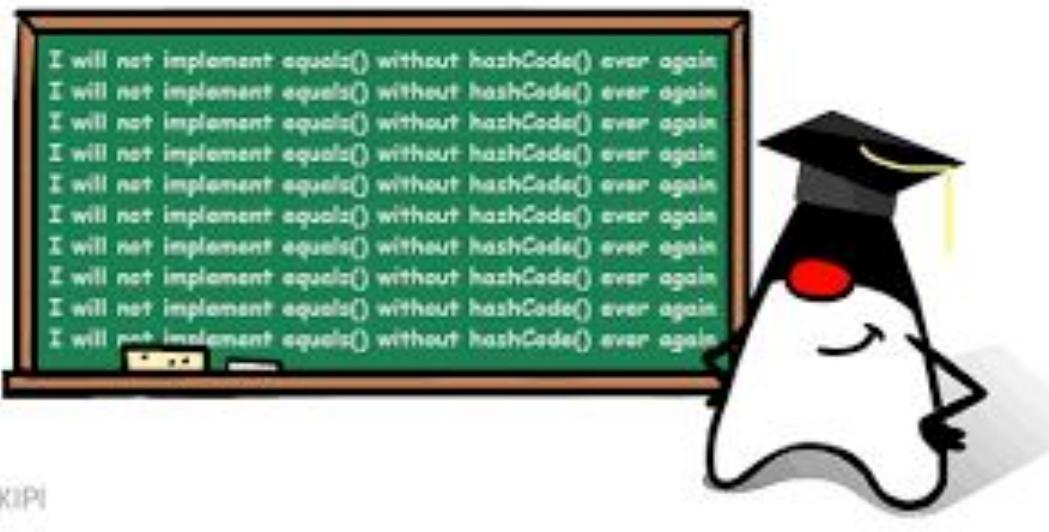


- Reduce los esfuerzos de programación al proveer estructuras de datos y algoritmos que no tenemos que escribir.
- Provee implementaciones de alta performance.
- Fomenta la reutilización del software al proporcionar una interfaz para colecciones y algoritmos para manipularlas.

Agenda

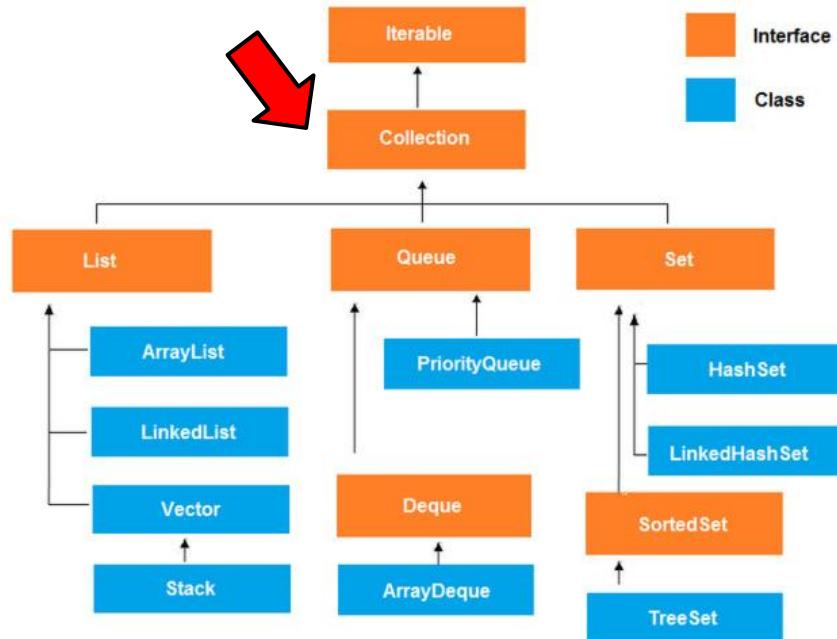
— Collection API

- **Interfaz Collection**
- List
- ArrayList
 - Insertar elemento
 - Lectura de un elemento
 - Eliminar elemento
- Método equals()
- Método hashCode()
- Método equals() - Ejemplo



Interfaz Collection

- Es la raíz de todas las interfaces y clases relacionadas con colecciones de elementos.
- Algunas colecciones permiten elementos duplicados mientras que otras no.
- Otras colecciones pueden tener los elementos ordenados mientras que en otras no existe orden alguno.



Interfaz Collection (2)

- **Por qué es una interfaz?**
- Es la manera más genérica para representar un grupo de elementos.
- Puede ser usada para pasar colecciones de elementos o manipularlas de la manera más general.
- Gracias a esta interfaz, podemos almacenar cualquier tipo de objeto y podemos usar una serie de métodos comunes, como pueden ser: añadir, eliminar, obtener el tamaño de la colección.
- Se trata de métodos definidos por la interfaz que obligatoriamente han de implementar las subinterfaces o clases que hereden de ella.

Interfaz Collection - Algunos métodos definidos

- boolean add(E e)
- int size()
- boolean isEmpty()
- boolean remove(E e)
- void clear()
- Object[] toArray()

Agenda

~~Collection API~~

~~Interfaz Collection~~

- **List**

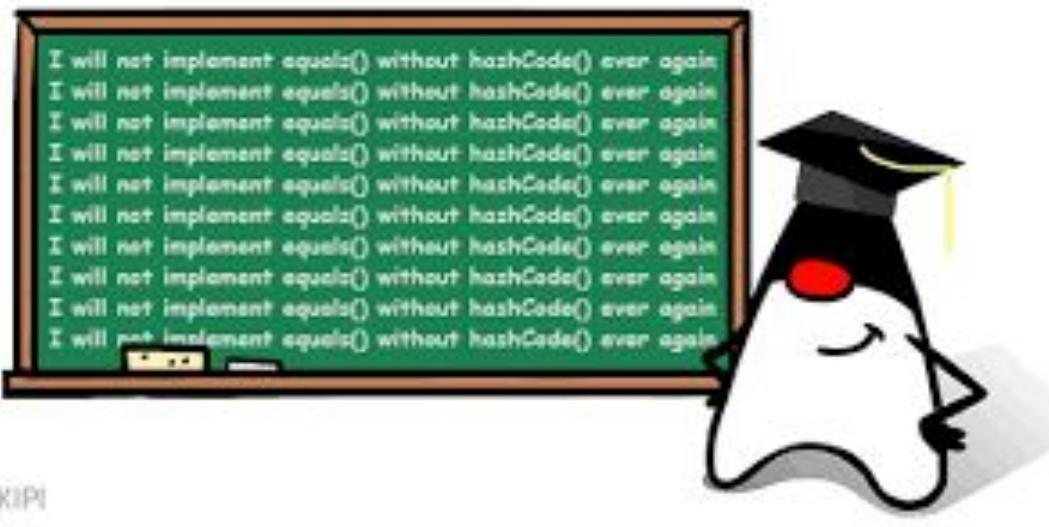
- **ArrayList**

- Insertar elemento
 - Lectura de un elemento
 - Eliminar elemento

- Método **equals()**

- Método **hashCode()**

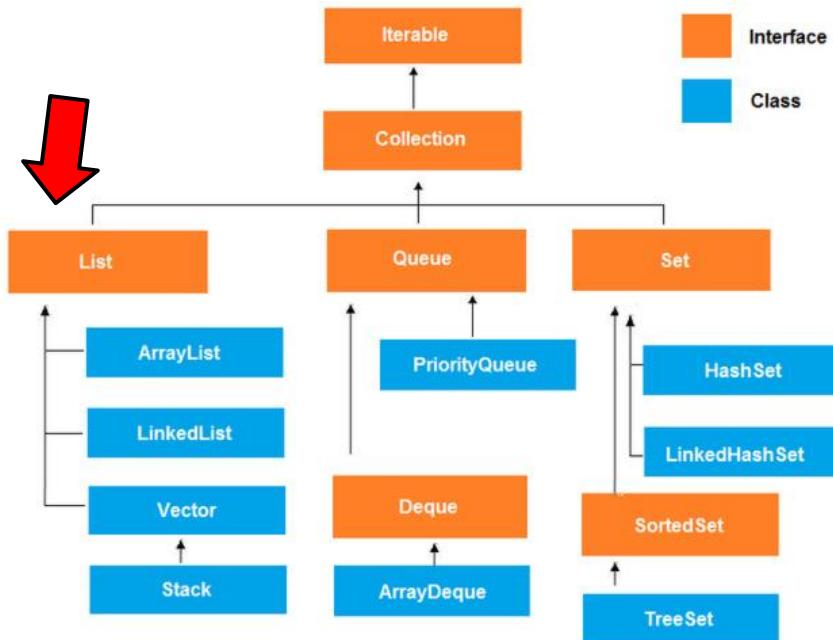
- Método **equals()** - Ejemplo



TAKIPI
2010

List

- Es la interfaz encargada de agrupar una colección de elementos en forma de lista, es decir uno detrás de otro.
- Acepta elementos duplicados.
- Al igual que en los arreglos, el primer elemento está en la posición 0.



List - Algunos métodos definidos

- E get(int index)
- E set(int index, E element)
- E add(int index, E element)
- E remove(int index)
- int indexOf(Object o)
- Object[] toArray()

Agenda

— Collection API

— Interfaz Collection

— List

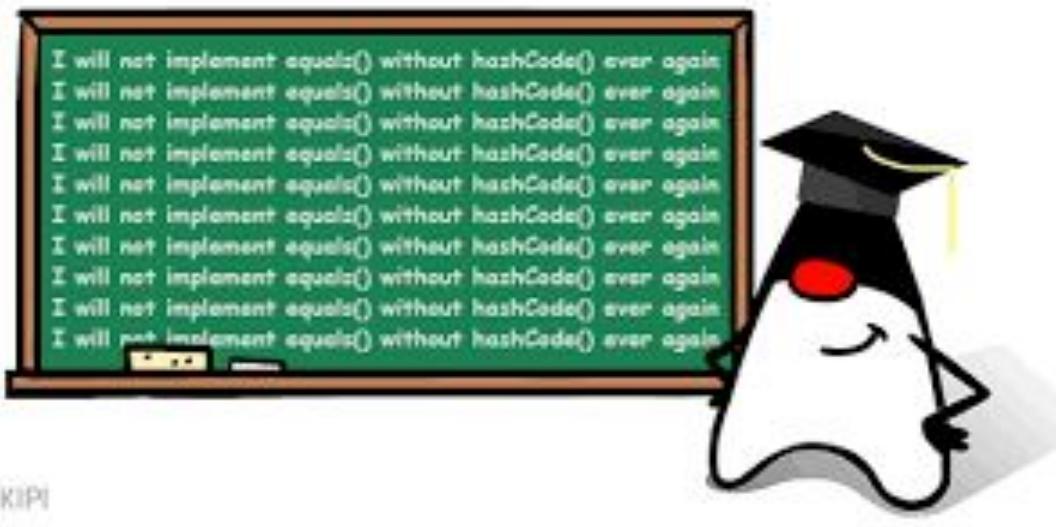
- **ArrayList**

- Insertar elemento
- Lectura de un elemento
- Eliminar elemento

- Método equals()

- Método hashCode()

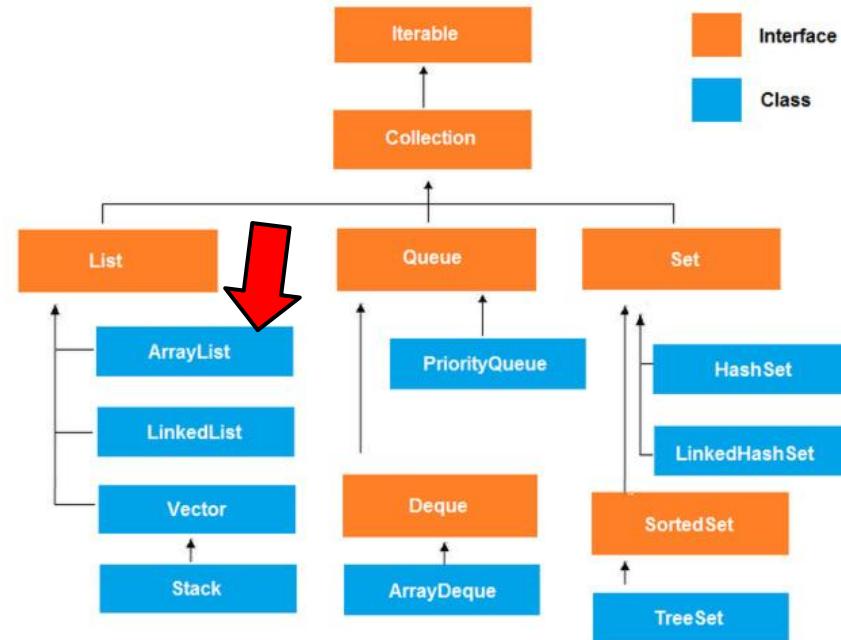
- Método equals() - Ejemplo



TAKIPI
2010

ArrayList

- Basa la implementación de la lista en un array de tamaño variable.
- Un beneficio de usar esta implementación es que las operaciones de acceso a elementos, capacidad y saber si es vacía se realizan de forma eficiente y rápida.
- Todo arraylist tiene una propiedad de capacidad, aunque cuando se añade un elemento esta capacidad puede incrementarse.



Agenda

— Collection API

— Interfaz Collection

— List

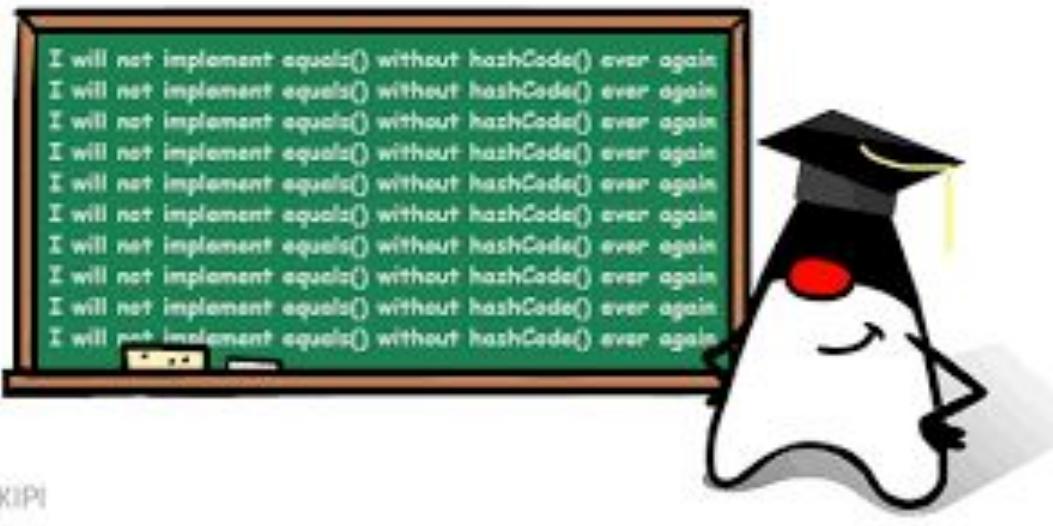
- **ArrayList**

- **Insertar elemento**
- Lectura de un elemento
- Eliminar elemento

- Método equals()

- Método hashCode()

- Método equals() - Ejemplo



TAKIPI
2010

ArrayList - Insertar elemento

```
public static void main(String [] args) {  
    List<Perro> perros = new ArrayList<>();  
    Perro p1 = new Perro("Boby");  
    Perro perro2 = new Perro("Ayudante de Santa");  
  
    perros.add(perro1);  
    perros.add(perro2);  
}
```

Antes de agregar un elemento al arreglo.

Construye un arreglo interno con una capacidad de 10.

```
Perro p1 = new Perro("Boby");
Perro perro2 = new Perro("Ayudante de Santa");
```

```
perros.add(perro1);  
perros.add(perro2);
```

Antes de agregar un elemento, se valida la capacidad del arreglo.

ArrayList - Insertar elemento (2)

```
public static void main(String [] args) {  
  
    List<Perro> perros = new ArrayList<>();  
  
    Perro p1 = new Perro("Boby");  
    Perro p2 = new Perro("Ayudante de Santa");  
    ...  
    Perro p11 = new Perro("Rob")  
  
    perros.add(p1);  
    ...  
    perros.add(p11);  
}
```

Se valida la capacidad del arreglo y como no es suficiente se crea un nuevo arreglo con el doble de la capacidad y se copia el anterior.

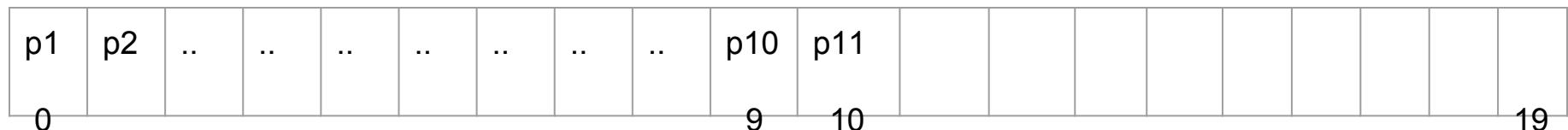
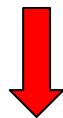


0

9

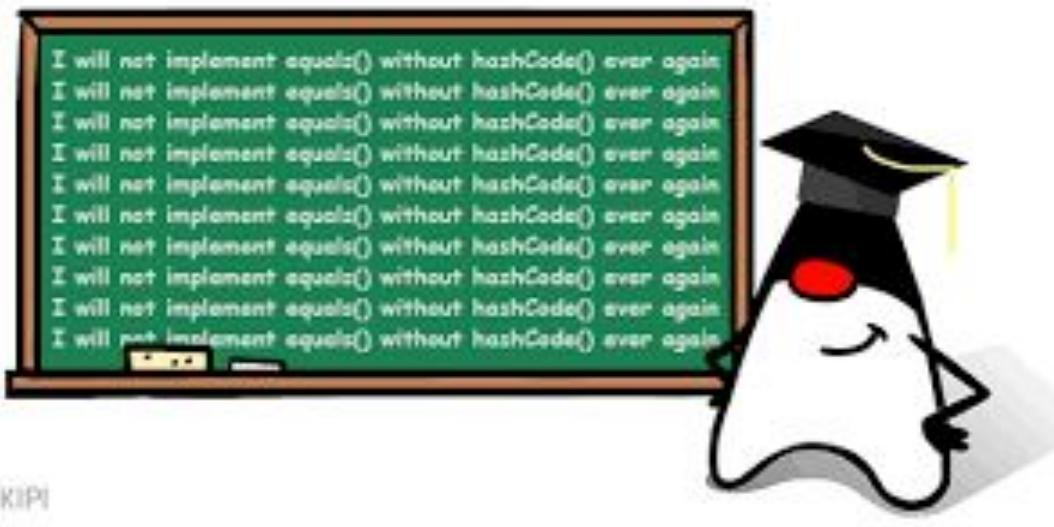
p11

ArrayList - Insertar elemento (3)



Agenda

- ~~Collection API~~
- ~~Interfaz Collection~~
- ~~List~~
- **ArrayList**
 - ~~Insertar elemento~~
 - **Lectura de un elemento**
 - ~~Eliminar elemento~~
- Método equals()
- Método hashCode()
- Método equals() - Ejemplo



ArrayList - Lectura de un elemento

- Como la estructura interna es un arreglo, se accede al elemento a través del índice correspondiente.

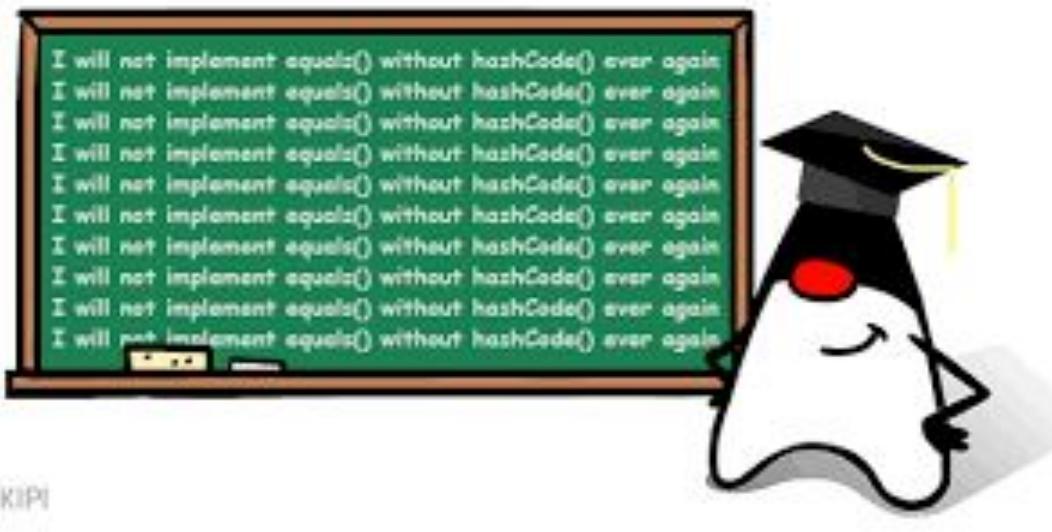
```
public static void main(String [] args) {  
  
    List<Perro> perros = new ArrayList<>();  
  
    Perro p1 = new Perro("Boby");  
    Perro p2 = new Perro("Ayudante de Santa");  
  
    perros.add(p1);  
  
    System.out.println(perros.get(0).getNombre());  
}
```



Es lo mismo que: p1.getNombre()

Agenda

- ~~Collection API~~
- ~~Interfaz Collection~~
- ~~List~~
- **ArrayList**
 - ~~Insertar elemento~~
 - ~~Lectura de un elemento~~
 - **Eliminar elemento**
- Método equals()
- Método hashCode()
- Método equals() - Ejemplo

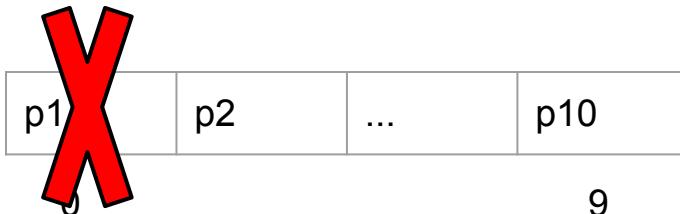


ArrayList - Eliminar elemento

1) A través del índice:

```
public static void main(String [] args) {  
  
    List<Perro> perros = new ArrayList<>();  
  
    Perro p1 = new Perro("Boby");  
    Perro p2 = new Perro("Ayudante de Santa");  
  
    perros.remove(0);  
  
}
```

ArrayList - Eliminar elemento (2)



Se deben correr los elementos



ArrayList - Eliminar elemento (3)

2) A través de igualdad:

```
public static void main(String [] args) {  
  
    List<Perro> perros = new ArrayList<>();  
  
    Perro p1 = new Perro("Boby");  
    Perro p2 = new Perro("Ayudante de Santa");  
  
    perros.remove(p1);  
  
}
```

ArrayList - Eliminar elemento (4)



Se verifica que cada elemento sea igual
al que se quiere eliminar.
`if (e.equals(p1))`



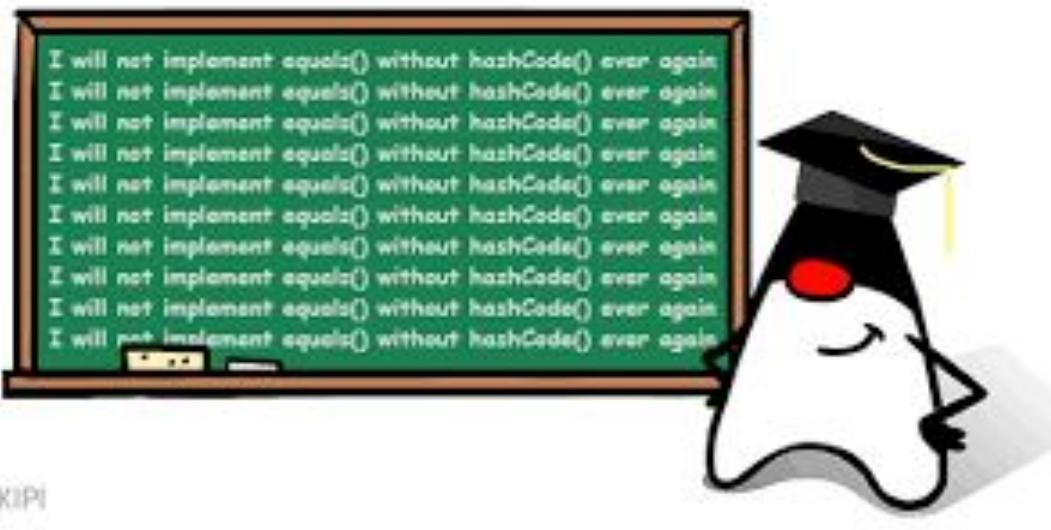
Se deben correr los elementos



IMPORTANTE!!!
Método equals()

Agenda

- ~~Collection API~~
- ~~Interfaz Collection~~
- ~~List~~
- ~~ArrayList~~
 - Insertar elemento
 - Lectura de un elemento
 - Eliminar elemento
- **Método equals()**
- **Método hashCode()**
- **Método equals() - Ejemplo**

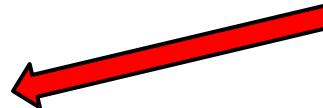


TAKIPI
TAKIPI

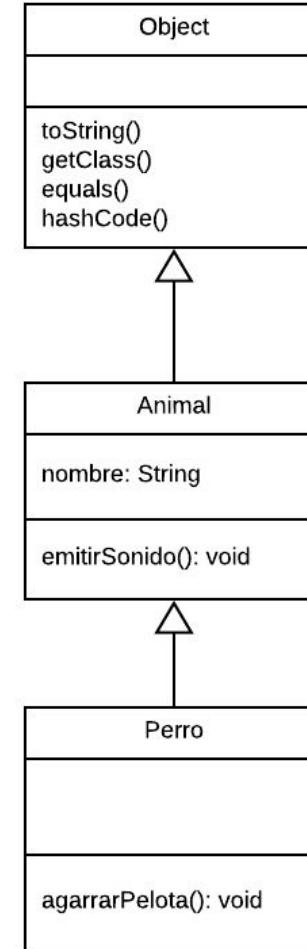
Método equals()

```
public boolean equals(Object obj) {  
    return (this == obj);  
}
```

Se compara la identidad



- Se debe redefinir el método equals() para comparar el contenido de los objetos que queremos evaluar.
- **Igualdad != Identidad**



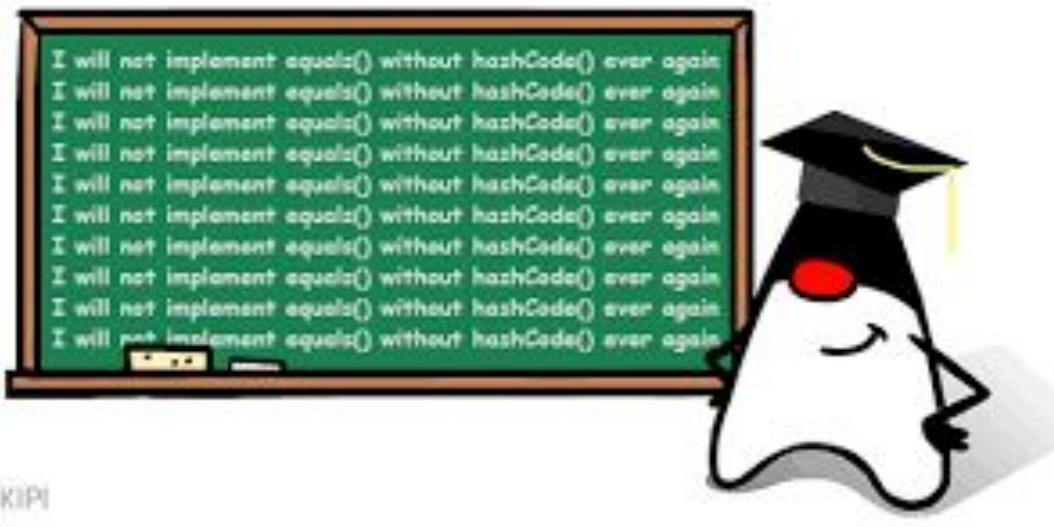
Método equals() (2)

- El método equals() está estrechamente relacionado con la función hashCode()
- Si sobreescribimos equals deberemos de sobrescribir también hashCode.
- hashCode debe cumplir que si dos objetos son iguales, según la función equals, debe de dar el mismo valor para ambos objetos.

Si `a.equals(b)` entonces `a.hashCode() == b.hashCode()`

Agenda

- ~~Collection API~~
- ~~Interfaz Collection~~
- ~~List~~
- ~~ArrayList~~
 - Insertar elemento
 - Lectura de un elemento
 - Eliminar elemento
- ~~Método equals()~~
- **Método hashCode()**
- Método equals() - Ejemplo



Método hashCode()

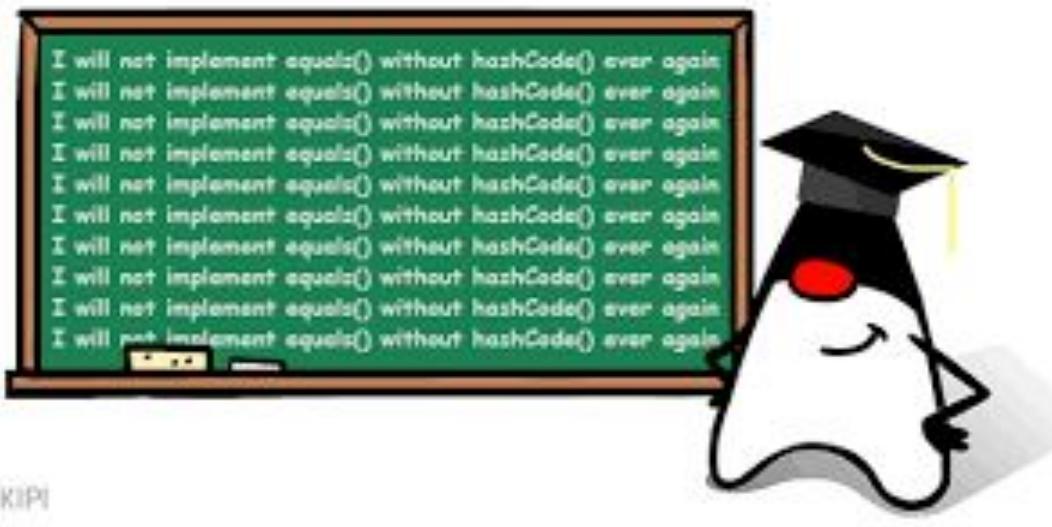
```
public int hashCode()
```

- Este método viene a complementar al método equals y sirve para comparar objetos de una forma más rápida en estructuras Hash ya que únicamente nos devuelve un número entero.

“To be continued...”

Agenda

- ~~Collection API~~
- ~~Interfaz Collection~~
- ~~List~~
- ~~ArrayList~~
 - Insertar elemento
 - Lectura de un elemento
 - Eliminar elemento
- ~~Método equals()~~
- ~~Método hashCode()~~
- **Método equals() - Ejemplo**



TAKIPI

Método equals() - Ejemplo

```
public class Perro extends Animal {  
  
    ...  
    @Override  
    public boolean equals(Object obj) {  
        if (this == obj)  
            return true;  
        if (obj == null)  
            return false;  
        if (getClass() != obj.getClass())  
            return false;  
        Perro otroPerro = (Perro) obj;  
        if (nombre.equals(otroPerro.nombre))  
            return true;  
        return false;  
    }  
}
```

Bibliografía oficial

- <https://docs.oracle.com/javase/tutorial/collections/interfaces/collection.html>
- <https://docs.oracle.com/javase/tutorial/collections/interfaces/list.html>
- <https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html>
- <https://docs.oracle.com/javase/tutorial/java/landl/objectclass.html>

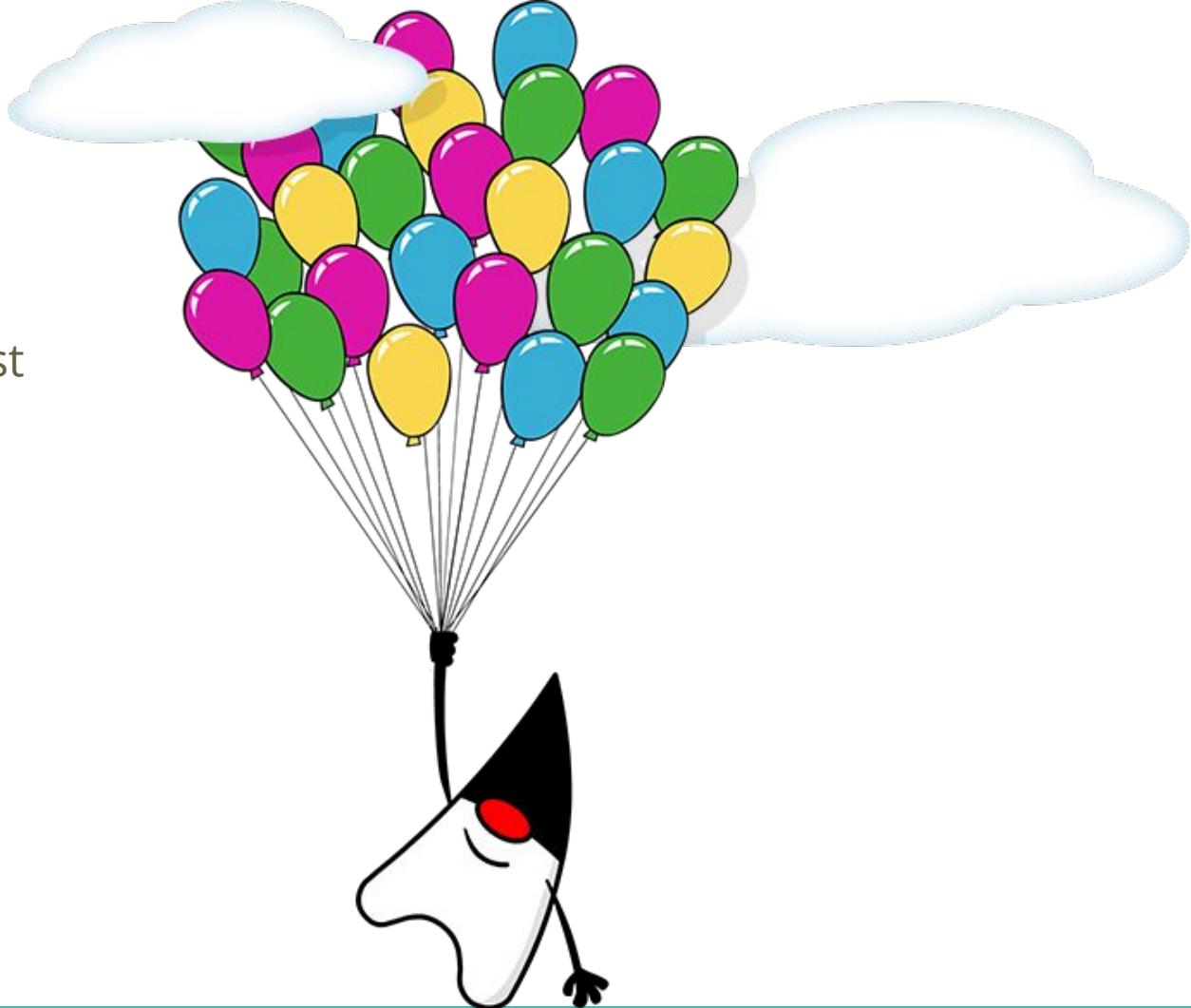
Clase 11: Collection - Listas

Parte II

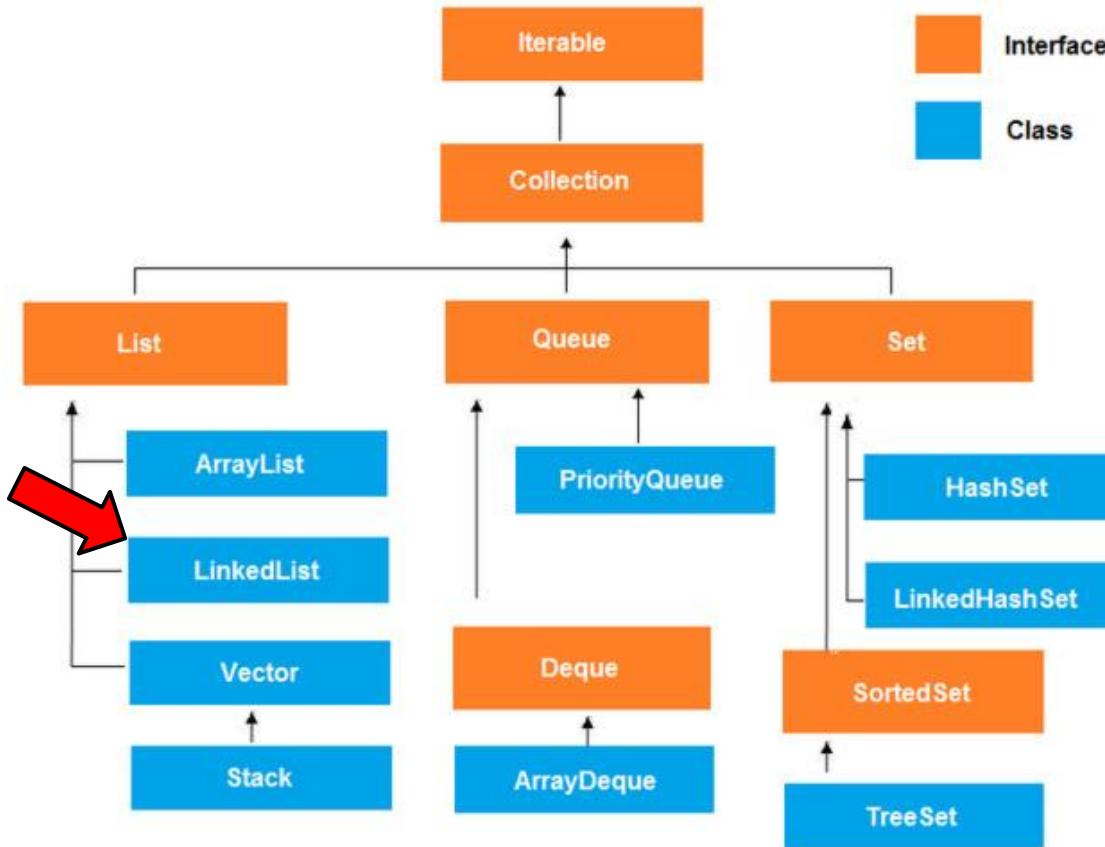
— Programación III —

Agenda

- **LinkedList (Repaso)**
- ArrayList vs. LinkedList
- Vector
- Vector vs. ArrayList
- Stack

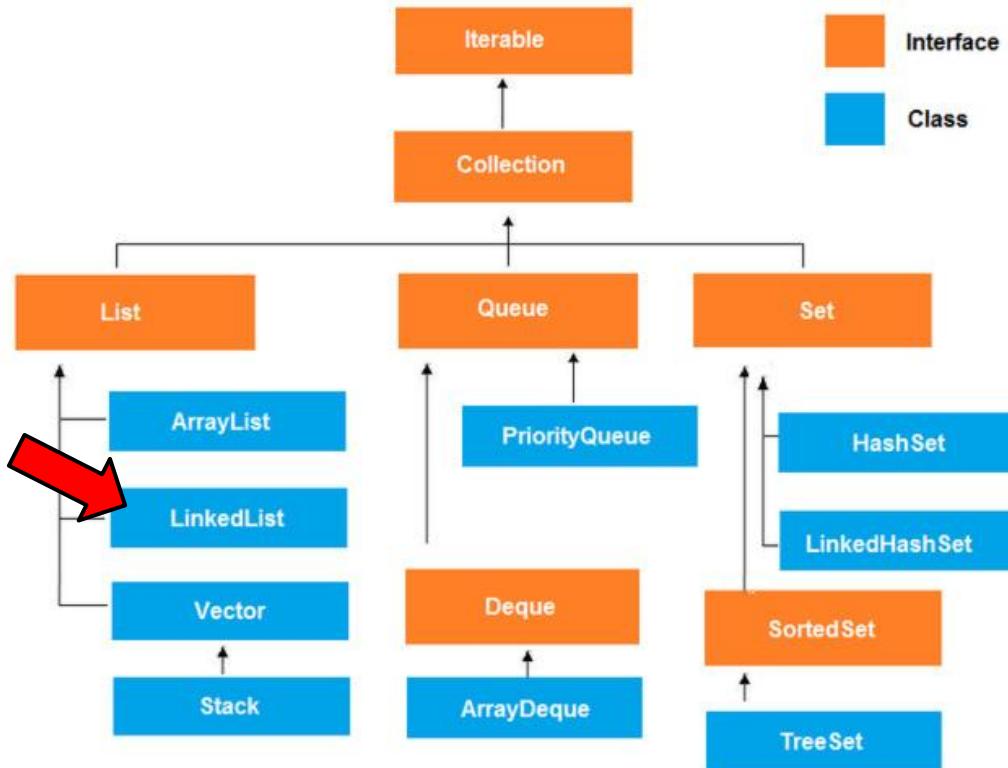


Collection API



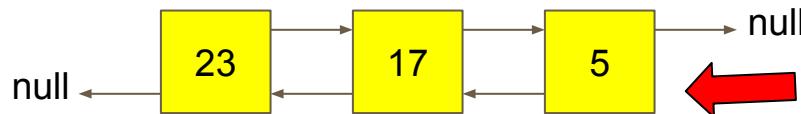
LinkedList

- Su implementación se base en una lista doblemente vinculada de tamaño ilimitado.
- Al igual que ArrayList, también implementa la interfaz List.
- Su estructura está formada por Nodos, cada nodo contiene dos enlaces: uno a su nodo predecesor y otro a su nodo sucesor.

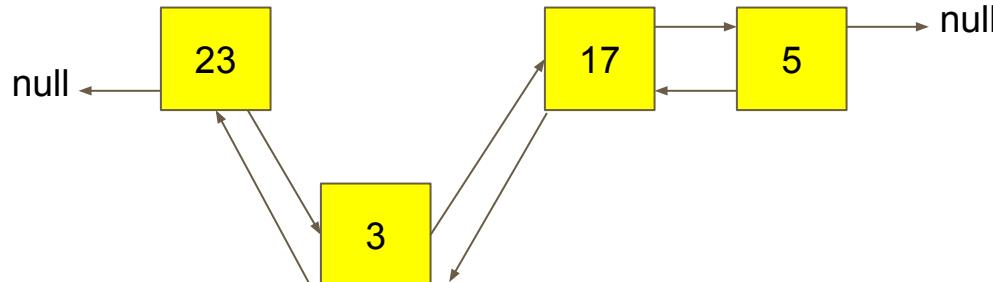


LinkedList - Insertar elemento

- 1) Agregar al final → boolean add(E e)

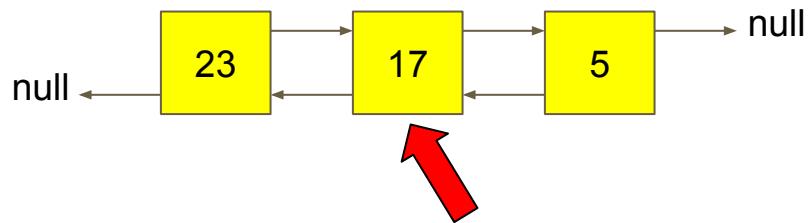


- 2) Agregar con índice → void add(int index, E element)



LinkedList - Leer elemento

- E get(int index)

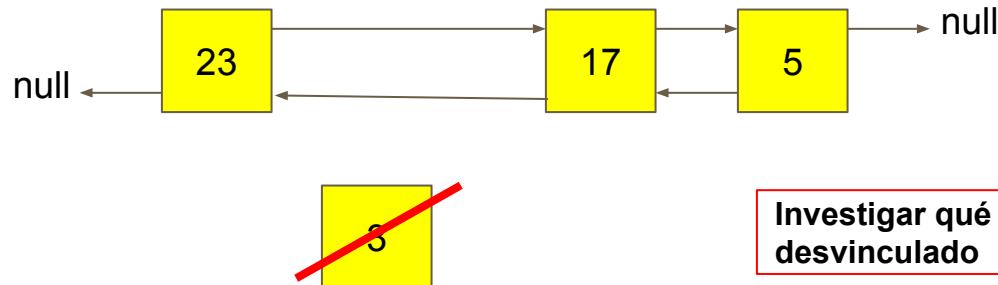


Investigar implementación
(link)



LinkedList - Eliminar elemento

- E remove(int index)



Investigar qué pasa con el nodo
desvinculado



Agenda

- ~~LinkedList (Repaso)~~
- **ArrayList vs. LinkedList**
- Vector
- Vector vs. ArrayList
- Stack



ArrayList vs LinkedList

- ArrayList está basada en una estructura de datos del tipo arreglo, mientras que LinkedList está basada en una lista doblemente vinculada.

0	1	2	3	4
23	3	17	9	42

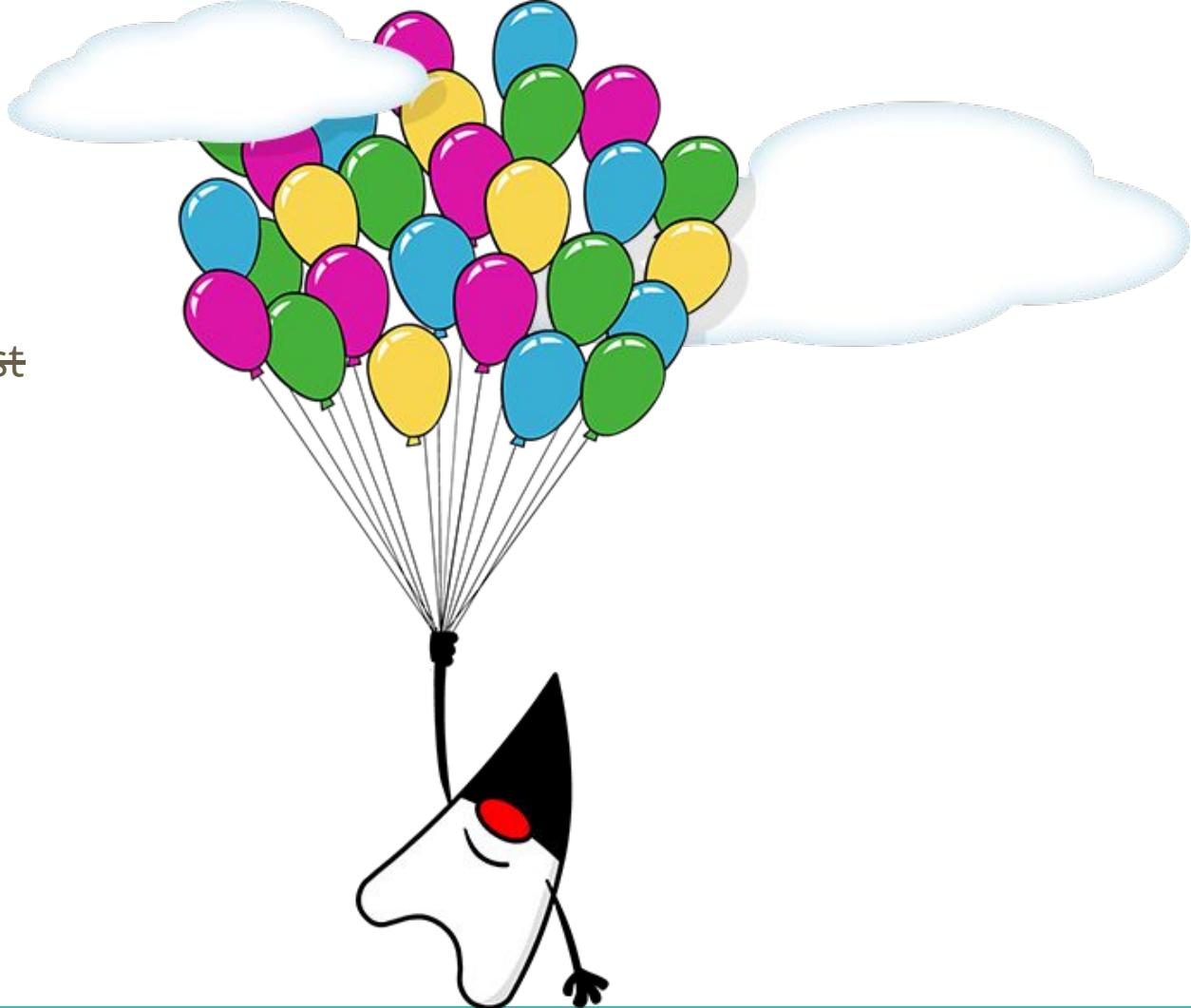


ArrayList vs LinkedList(2)

- Almacenar elementos en un ArrayList consume menos memoria y generalmente es más rápido en tiempos de acceso.
- Agregar o eliminar elementos usualmente es más rápido en LinkedList, pero como normalmente se debe iterar hasta la posición en la que se desea agregar o eliminar el elemento, la pérdida de rendimiento a veces es más grande que la ganancia (no siempre).

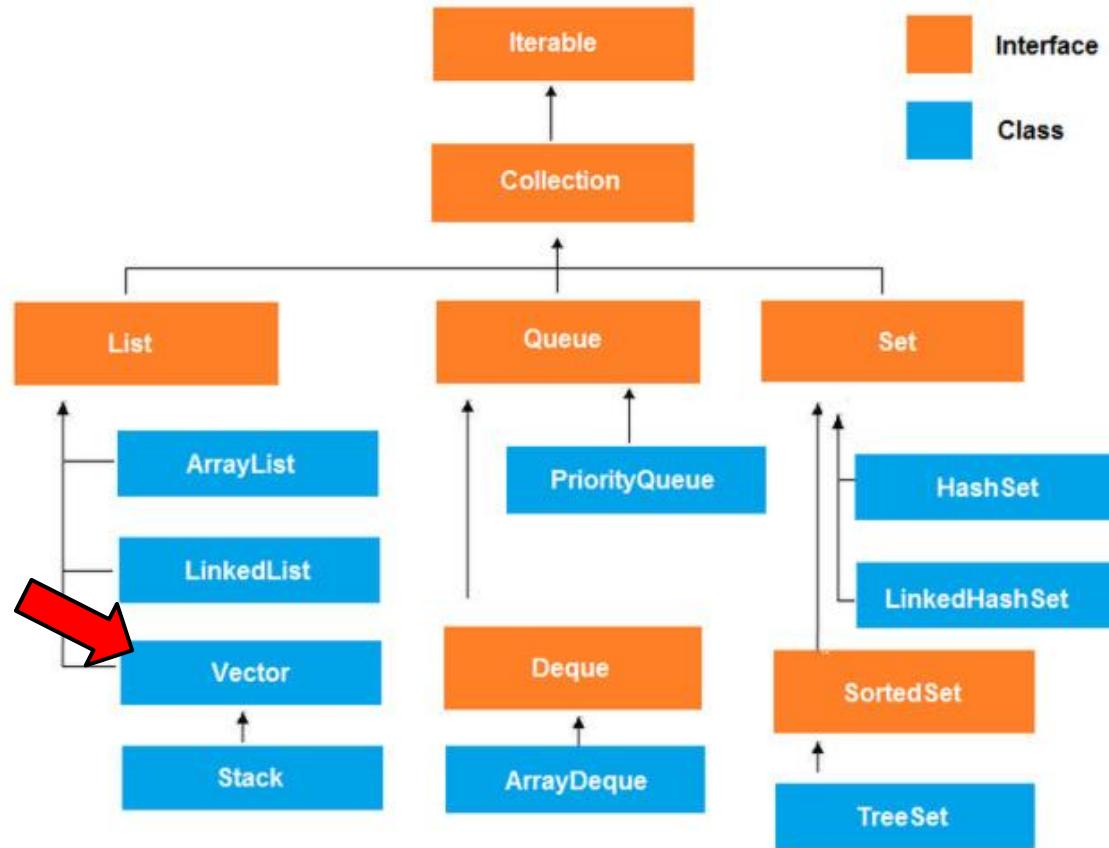
Agenda

- ~~LinkedList (Repaso)~~
- ~~ArrayList vs. LinkedList~~
- **Vector**
- Vector vs. ArrayList
- Stack



Vector

- Un Vector es similar a un array que crece automáticamente cuando alcanza la capacidad inicial máxima.
- También puede reducir su tamaño.
- La capacidad siempre es al menos tan grande como el tamaño del vector.



Vector (2)

```
Vector vector = new Vector(20, 5);
```

- El vector se inicializa con una dimensión inicial de 20 elementos. Si rebasamos dicha dimensión y guardamos 21 elementos la dimensión del vector crece a 25.

```
Vector vector=new Vector(20);
```

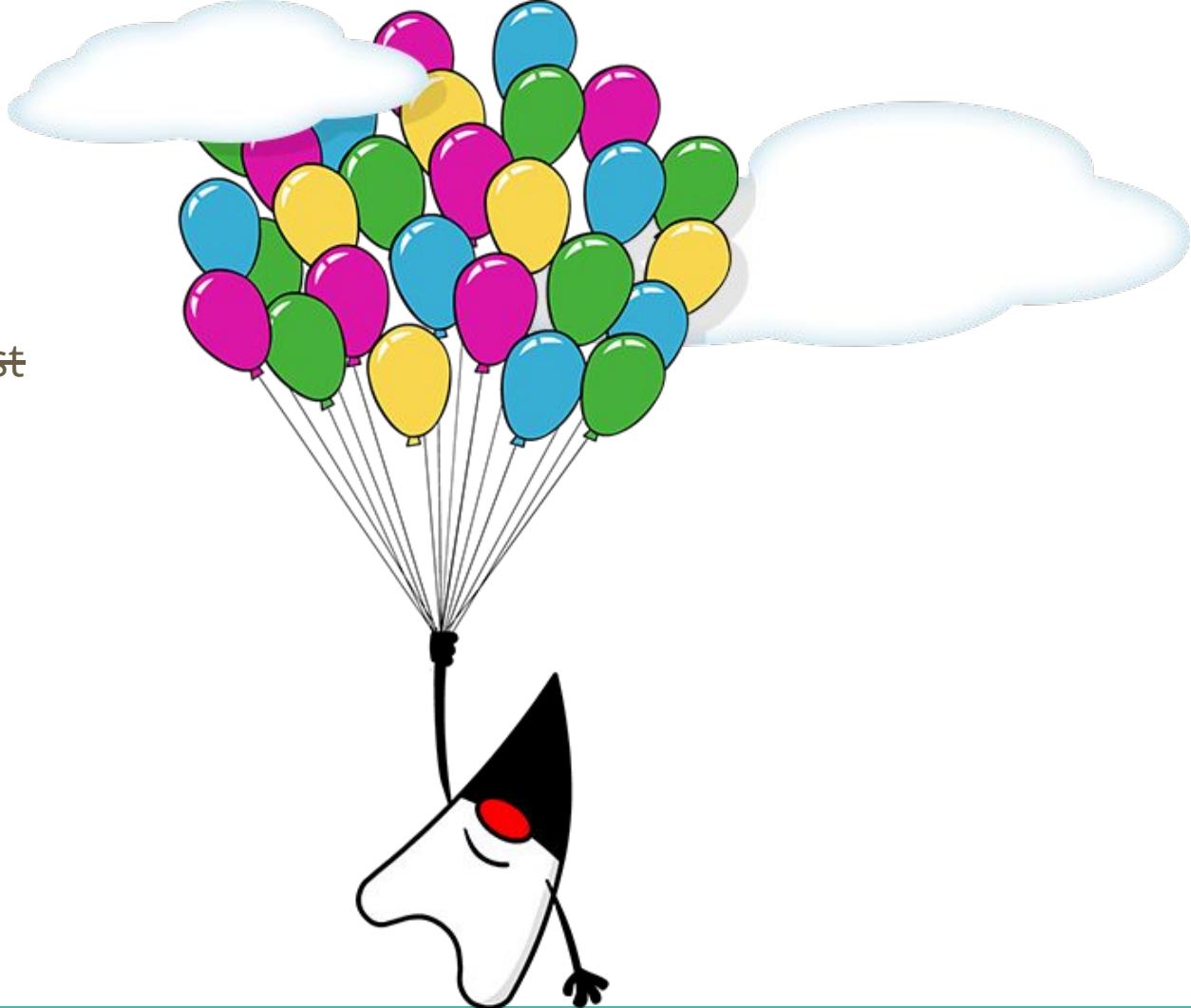
- Si se rebasa la dimensión inicial guardando 21 elementos, la dimensión del vector se duplica.

```
Vector vector=new Vector();
```

- Se inicializa con dimensión incial de 10 elementos. La dimensión del vector se duplica si se rebasa la dimensión inicial

Agenda

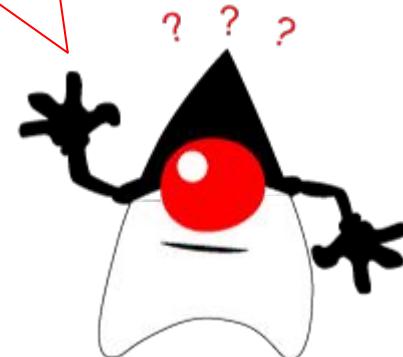
- ~~LinkedList (Repaso)~~
- ~~ArrayList vs. LinkedList~~
- ~~Vector~~
- **Vector vs. ArrayList**
- Stack



Vector vs. ArrayList

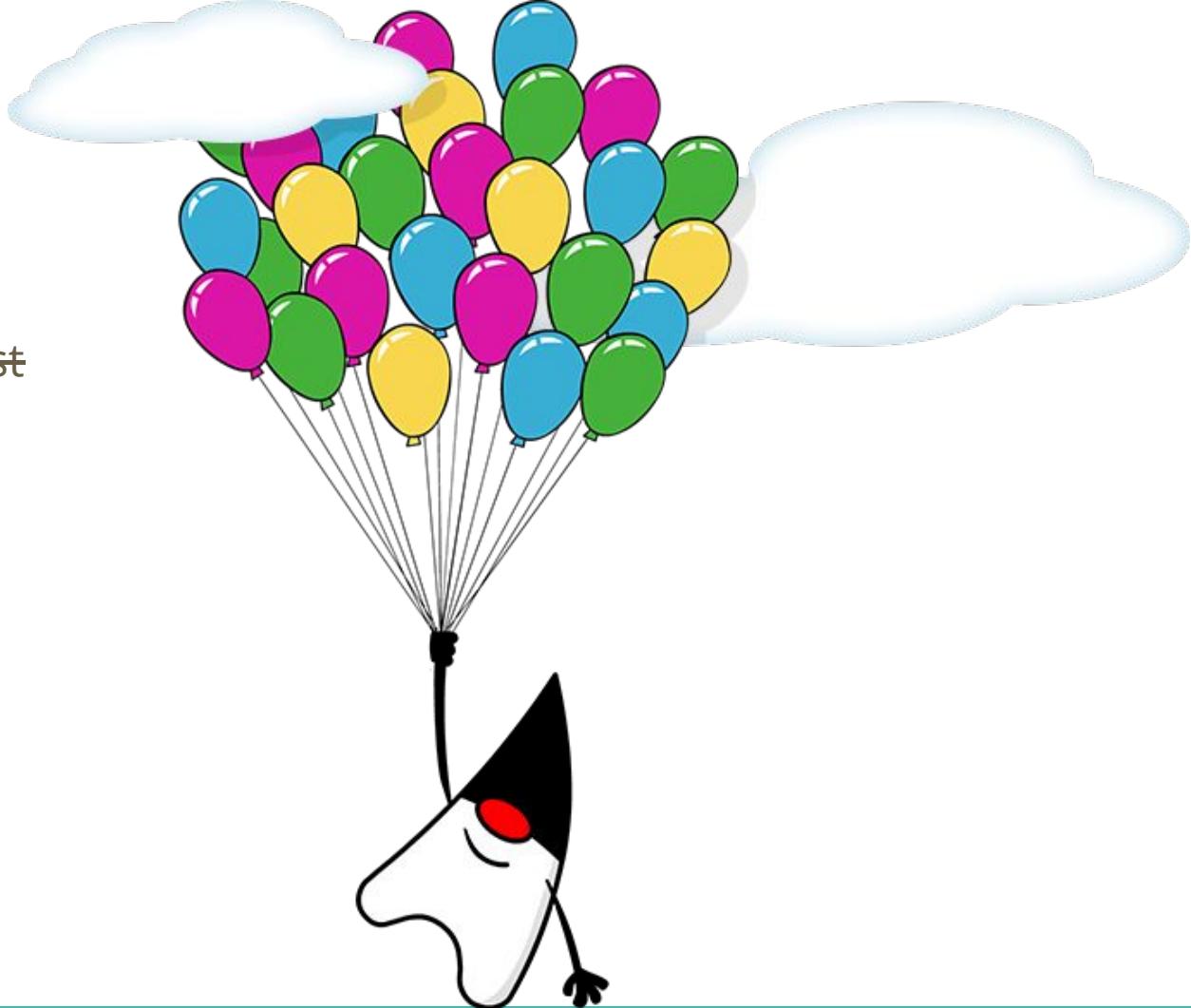
- Vector es sincronizada mientras que ArrayList no lo es. Si no trabajamos en un entorno multihilos utilizar ArrayList.
- La estructura de ambos está basada en array.
- Ambos pueden crecer y reducirse en forma dinámica, sin embargo la forma en la que se redimensionan es diferente.

Investigar
Collections.synchronizedList



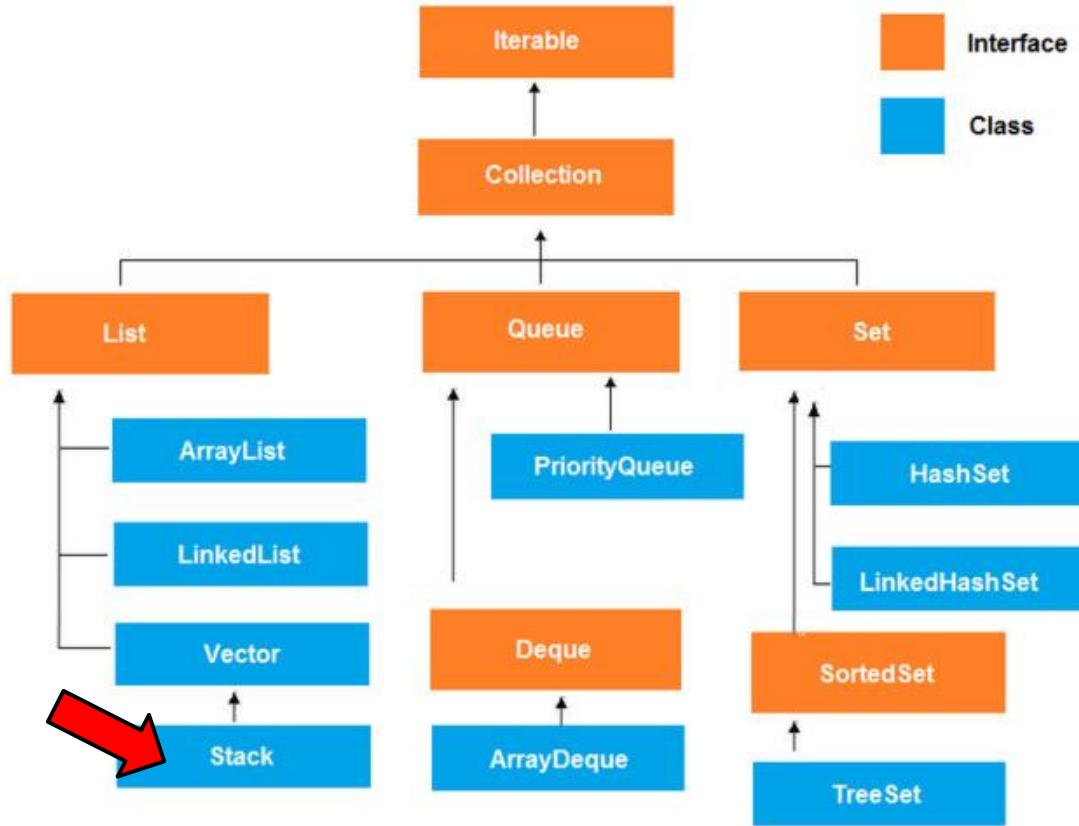
Agenda

- ~~LinkedList (Repaso)~~
- ~~ArrayList vs. LinkedList~~
- ~~Vector~~
- ~~Vector vs. ArrayList~~
- Stack



Stack

- Representa una estructura LIFO (last in - first out).
- Es una subclase de Vector, por lo tanto su estructura también está basada en un array.
- Al igual que Vector, Stack es sincronizada.



Stack - Operaciones básicas

- **push** → introduce un elemento en la pila.
- **pop** → saca un elemento de la pila.
- **peek** → consulta el primer elemento de la cima de la pila.
- **empty** → comprueba si la pila está vacía.
- **search** → busca un determinado elemento dentro de la pila y devuelve su posición dentro de ella.

Bibliografía oficial

- <https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html>
- <https://docs.oracle.com/javase/8/docs/api/java/util/Vector.html>
- <https://docs.oracle.com/javase/7/docs/api/java/util/Stack.html>
- <https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html>