
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.**



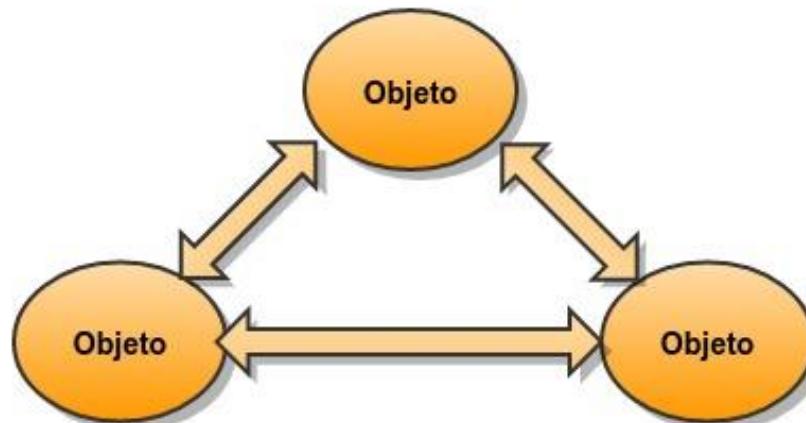
Agenda

- ~~¿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

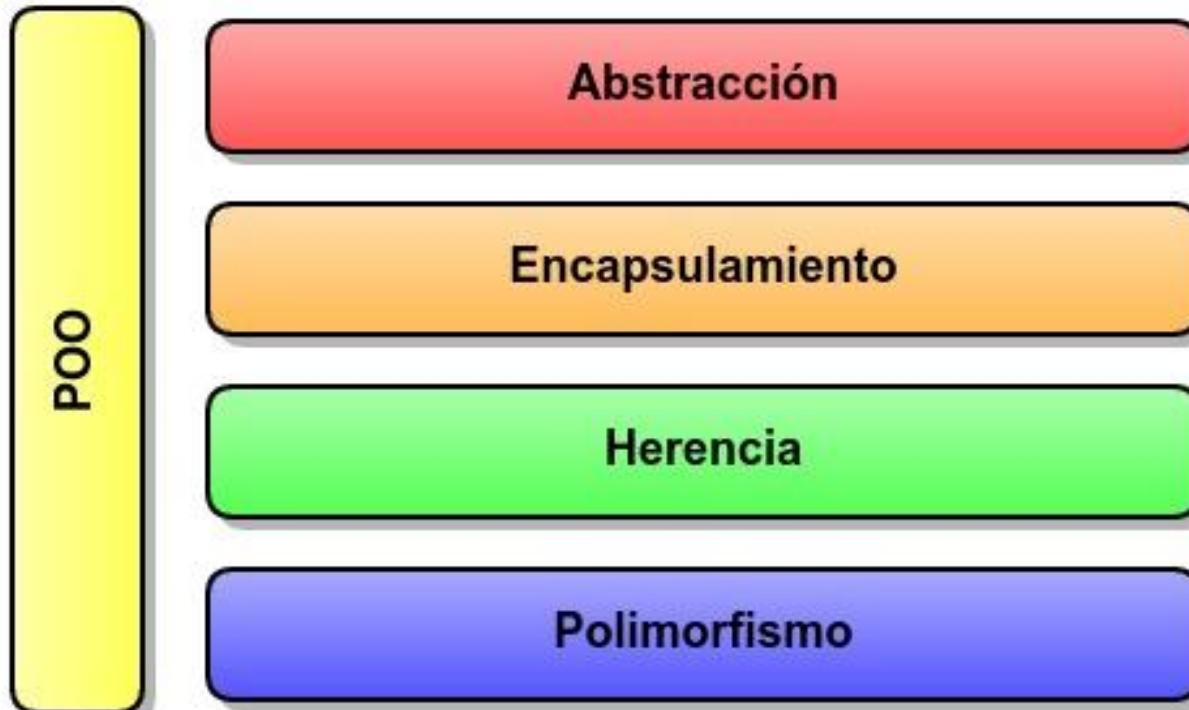
No basta un lenguaje OO para programar orientado a objetos, para eso hay que seguir un paradigma orientado a objetos.

- 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

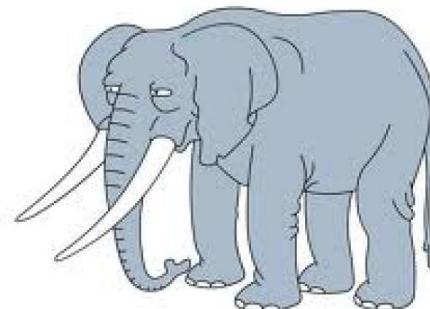
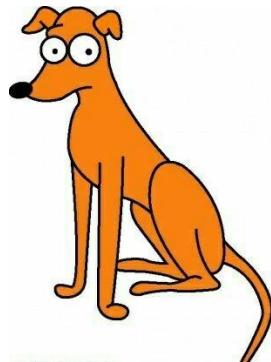
- Consiste en **aislar** un elemento de su contexto → ¿Qué hace?
- Se enfoca en la visión externa de un objeto → Separar el comportamiento específico.
- Quitar las propiedades y acciones de un objeto para dejar solo aquellas que sean necesarias.

La abstracción es clave para diseñar un buen software.



Abstracción - Ejemplo

¿Qué características podemos abstraer de los animales?



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

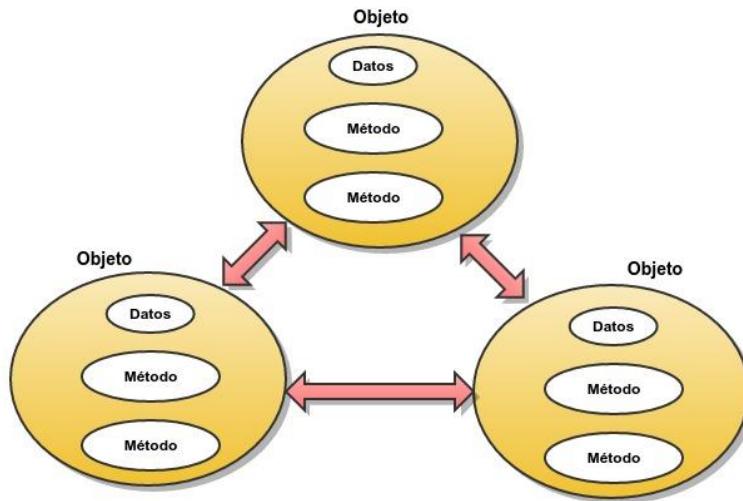
Agenda

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

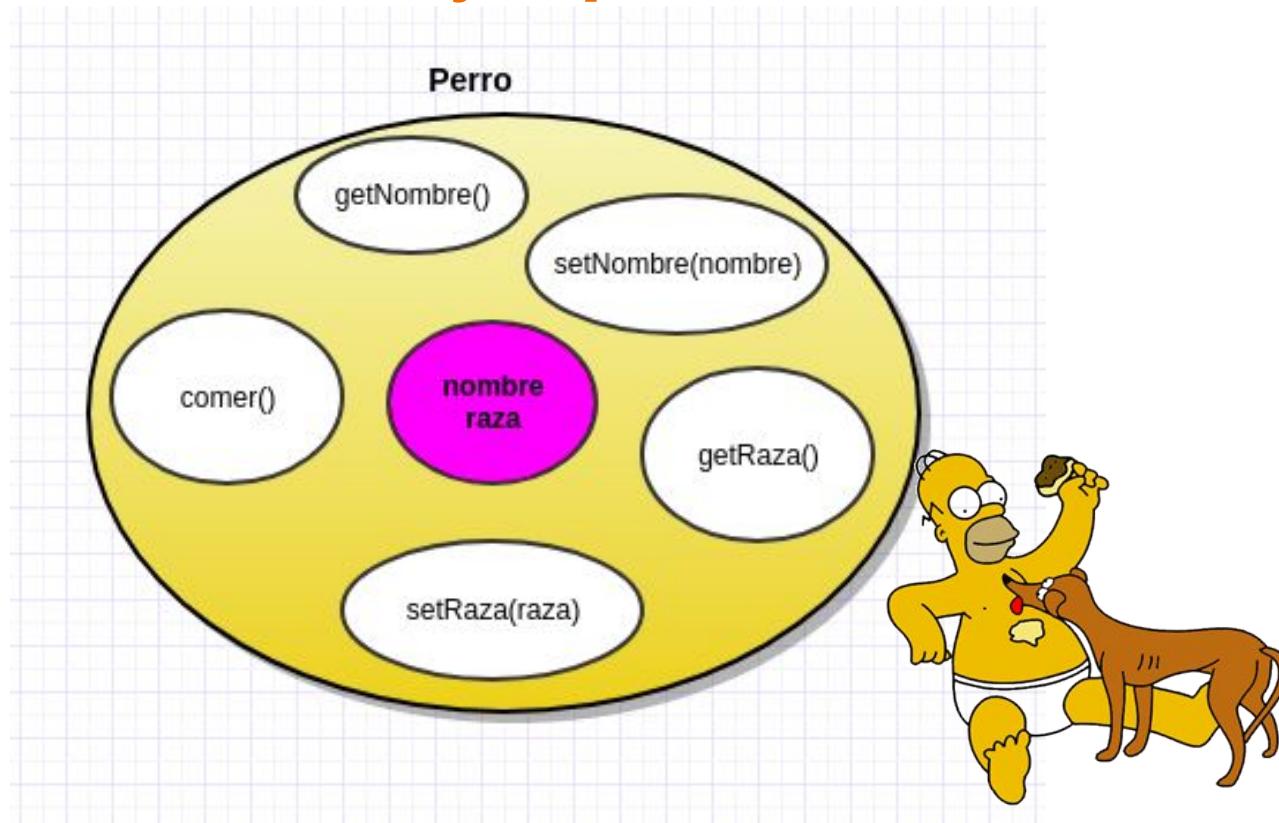


Encapsulamiento

- 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

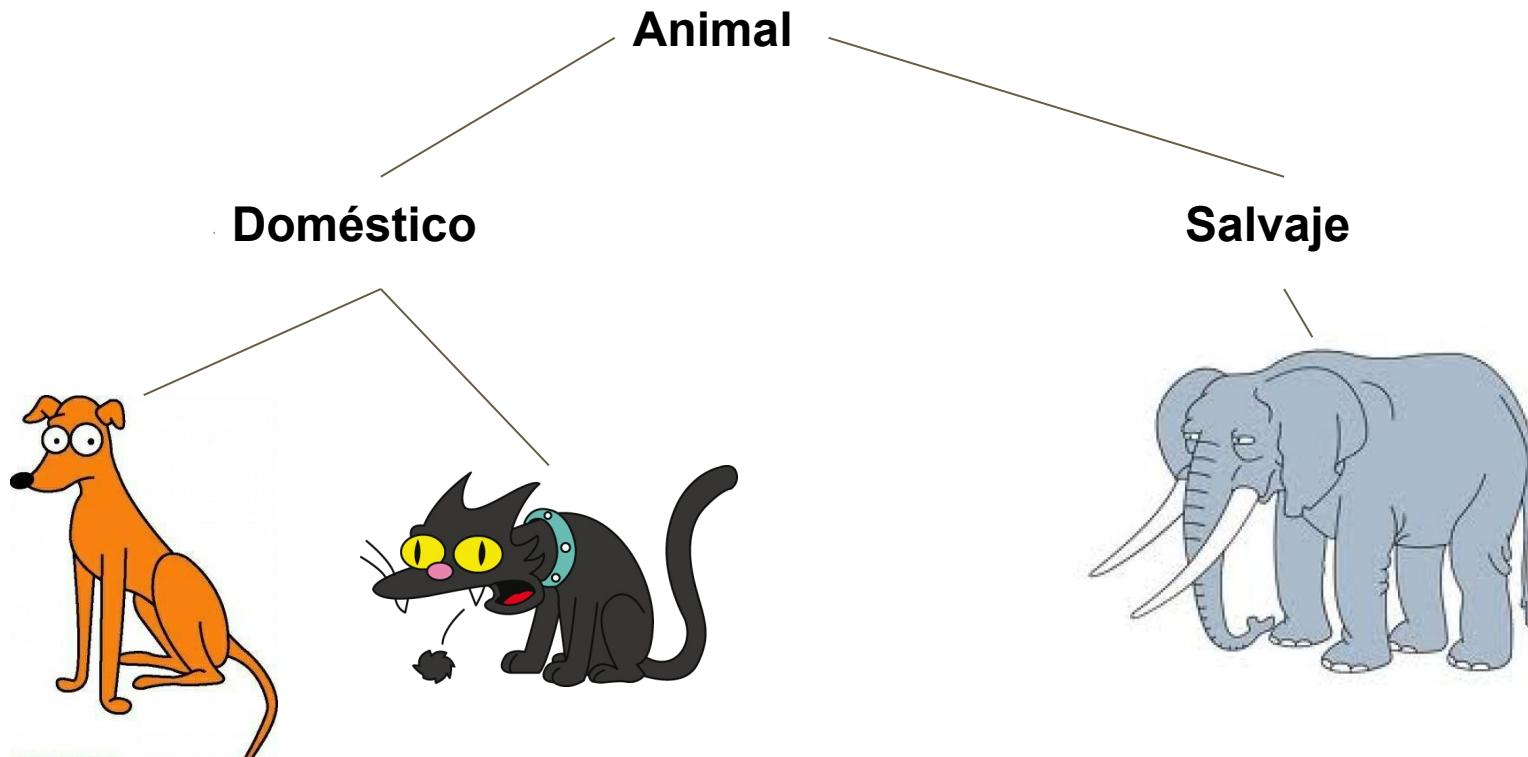


Agenda

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



Herencia - Ejemplo



Agenda

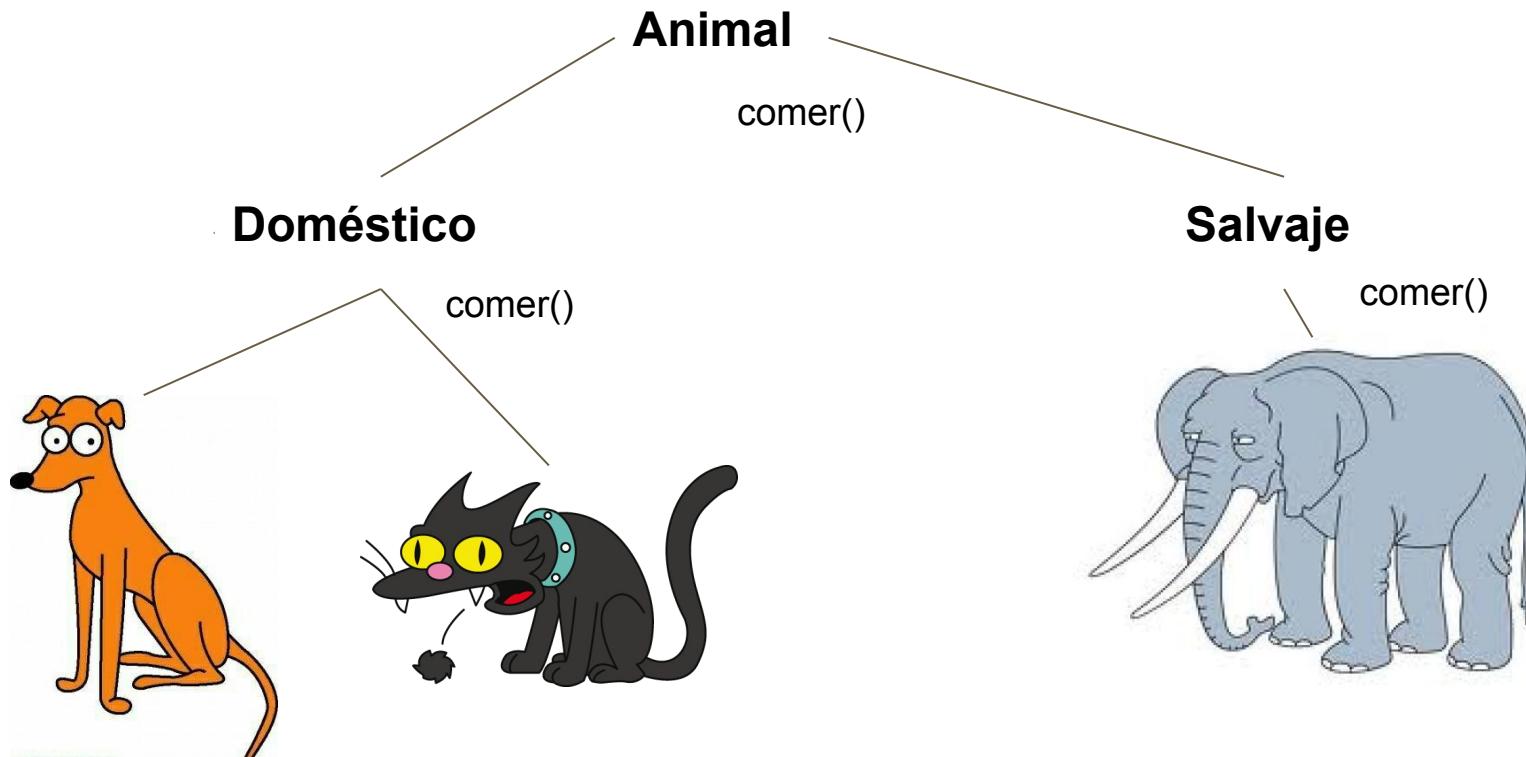
- ~~¿Qué es paradigma?~~
- ~~Paradigma Orientado a Objetos~~
- ~~Lenguaje de Programación Orientado a Objetos~~
- Programación Orientada a Objetos:
 - Abstracción
 - Encapsulamiento
 - Herencia
 - Polimorfismo



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:

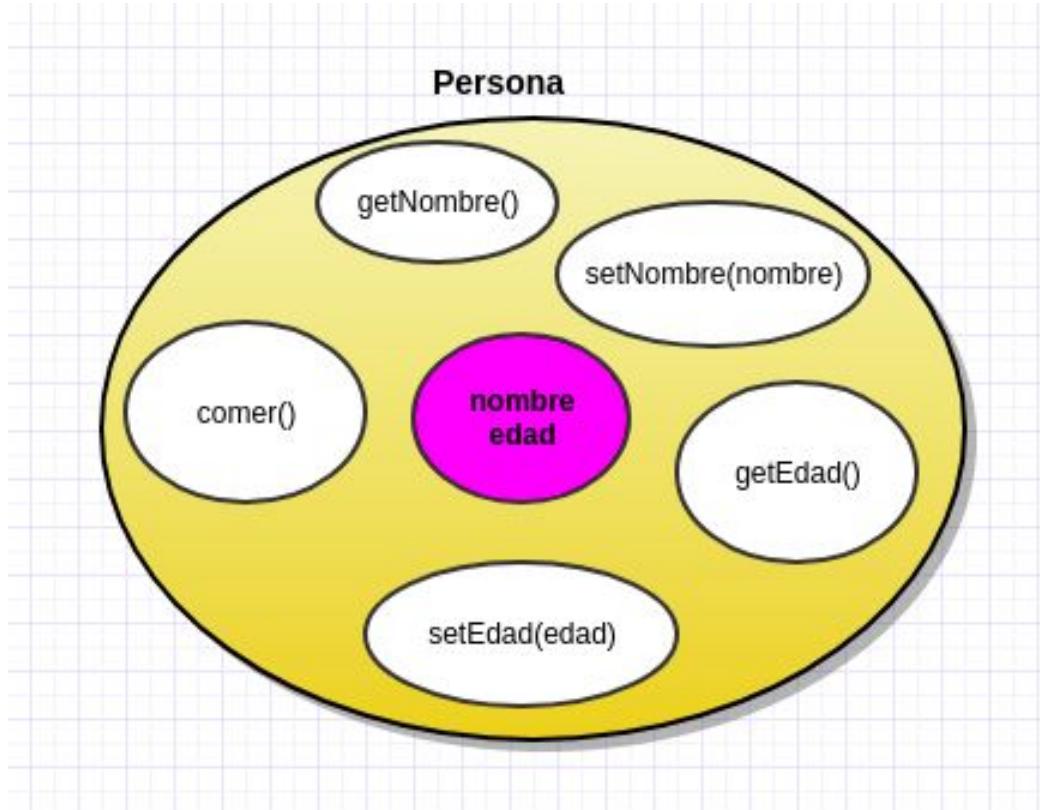
- Nombre
- Edad
- Peso
- Altura
- Fecha de nacimiento
- etc.

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

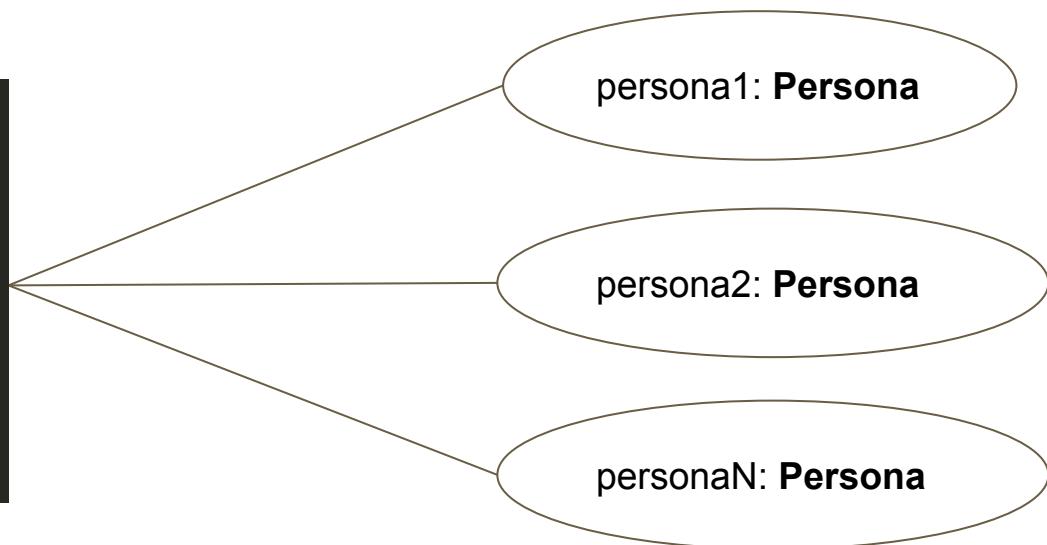


Instancia

- 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;  
    }  
}
```

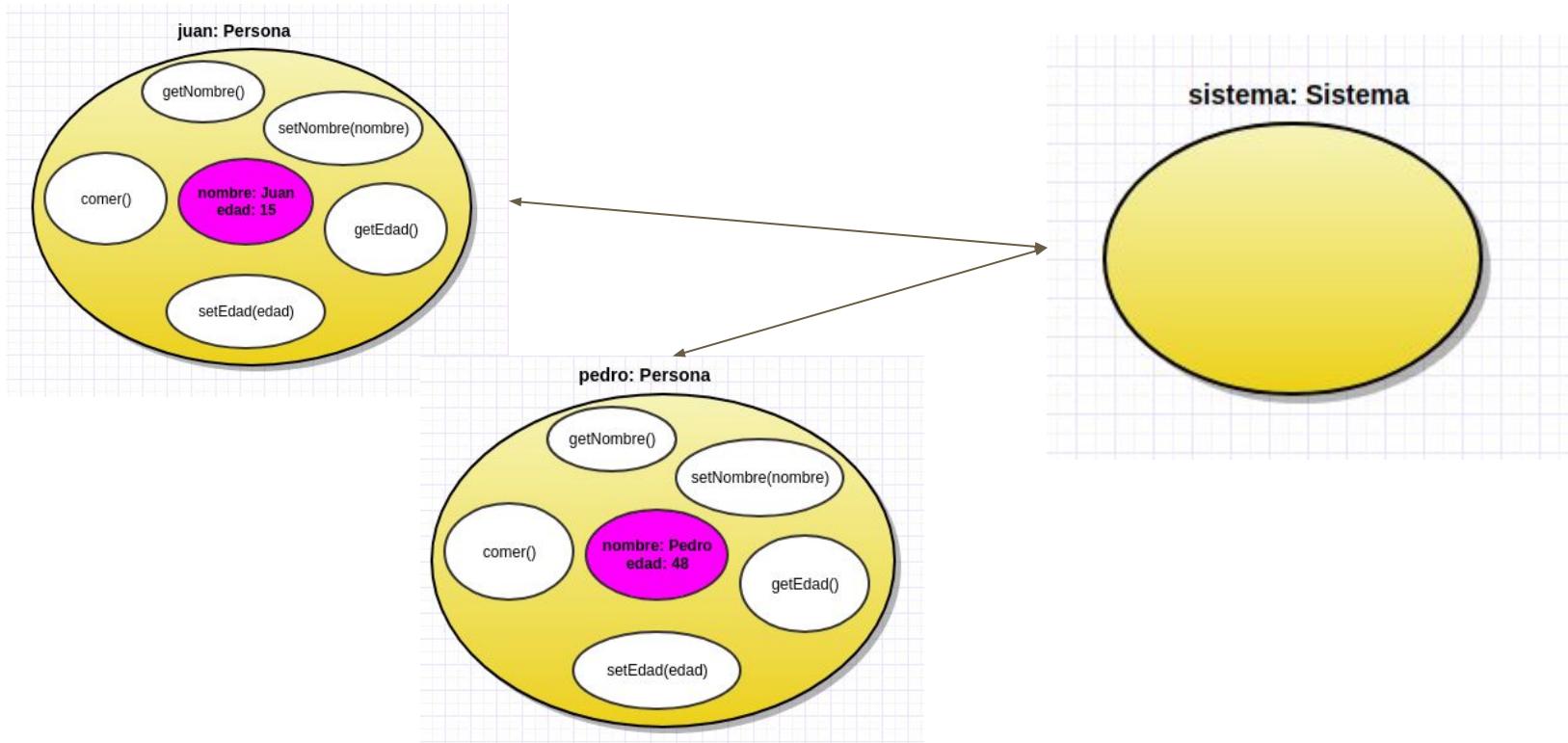


Agenda

- 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>;`

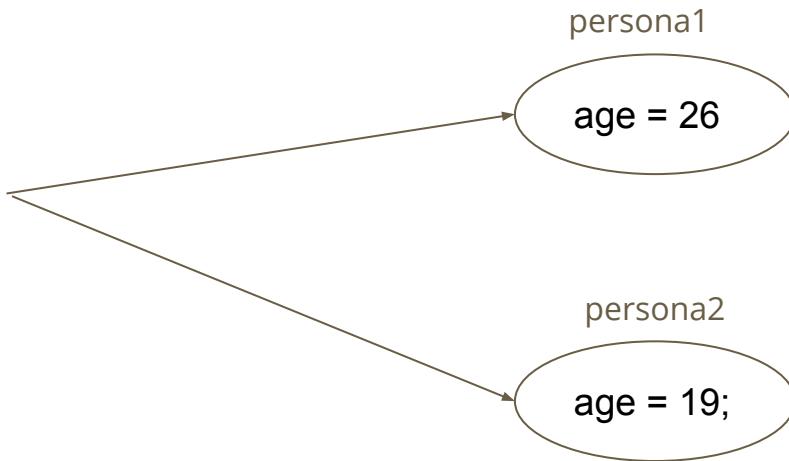


Variables - Clasificación

- **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:

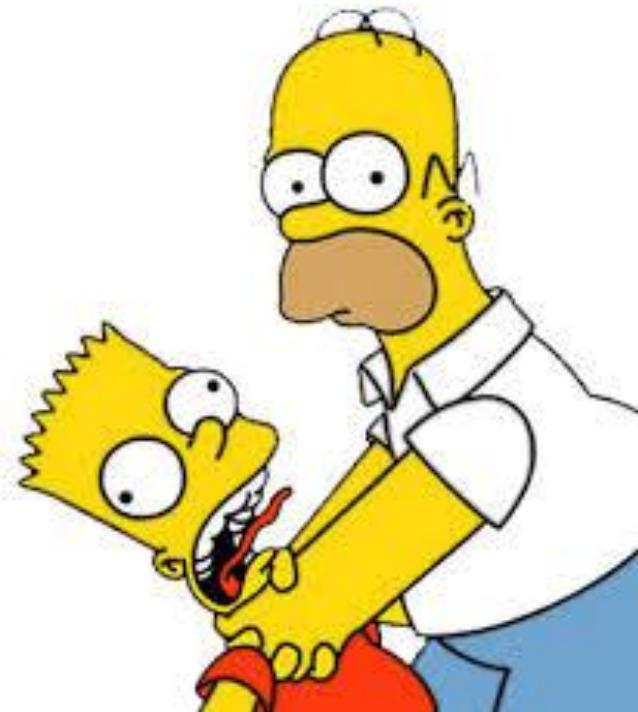
Identificadores válidos	Identificadores no válidos
customerValidObject	7world
\$rate, £Value, _sine	%value
happy2Help, nullValue	Digital!, books@manning
Constant	null, true, false, goto

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

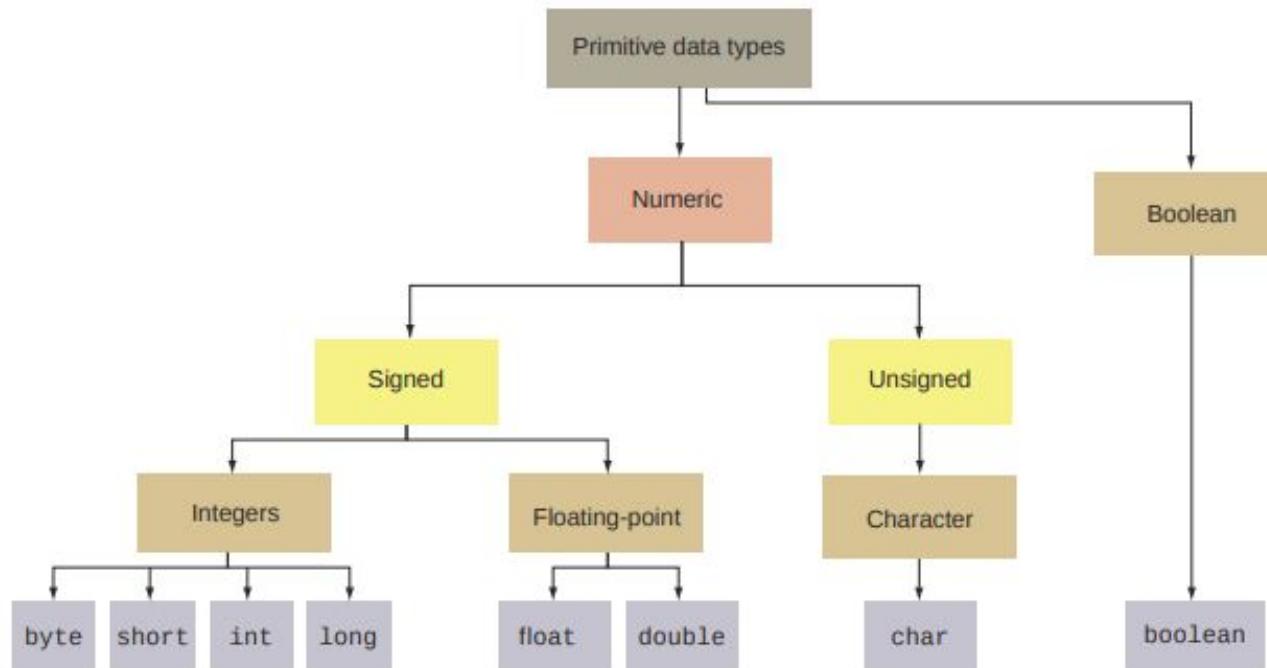
- Variables
 - Clasificación
 - Nomenclatura de identificador
- Tipos primitivos
 - Categorización
 - Ejemplos
 - Valores por defecto
- Java Virtual Machine
 - Secciones
 - Tareas principales
 - Garbage Collector
- Tipo objeto
 - Creación
- Diferencias entre variables de tipo objeto y tipo primitivo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

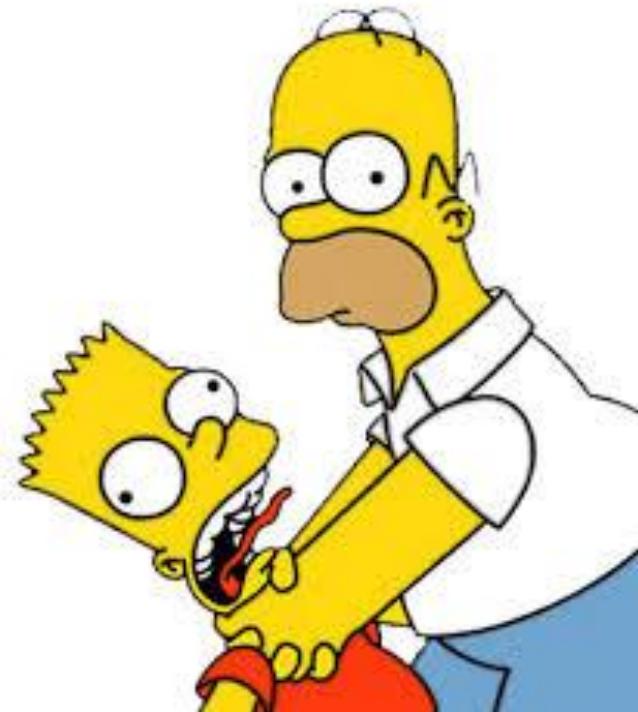
char	byte (8 bits)	short (16 bits)	int (32 bits)	long (64 bits)	float (32 bits)	double (64 bits)	boolean
'a' 'D' '122'	100	1240	48764 0413 0x10B 0b100001011	214748368	20.12F 1765.65f 120.1762	120.176D 120.1762	true false

Tipo primitivo - Valores por defecto

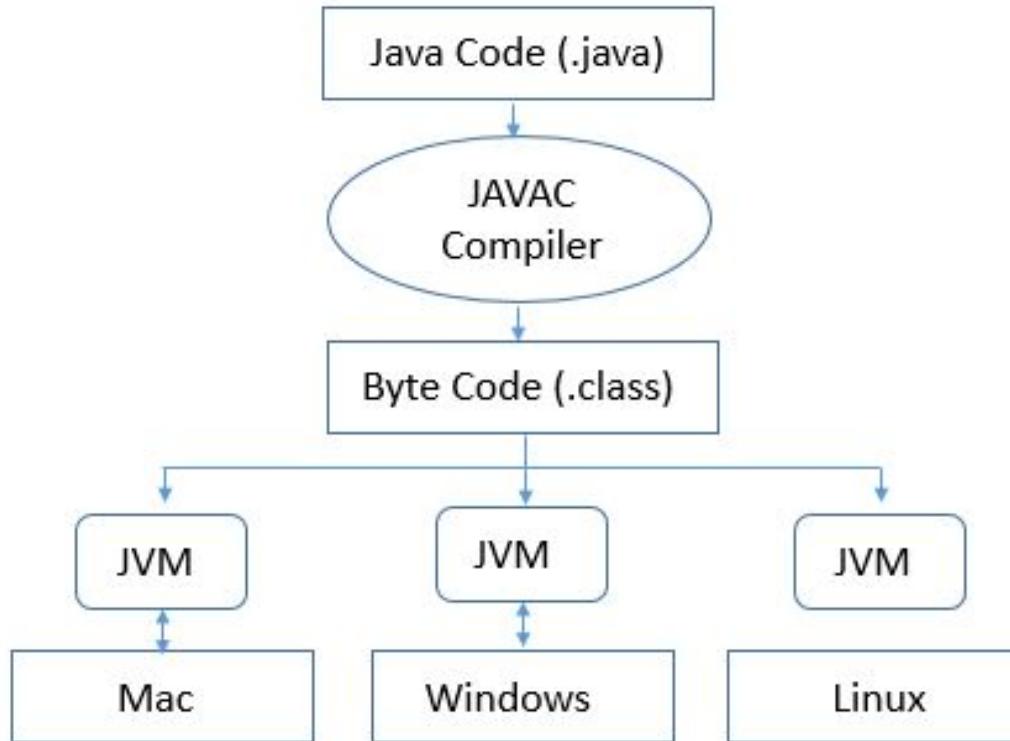
Tipo	Valor por defecto
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
boolean	false

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

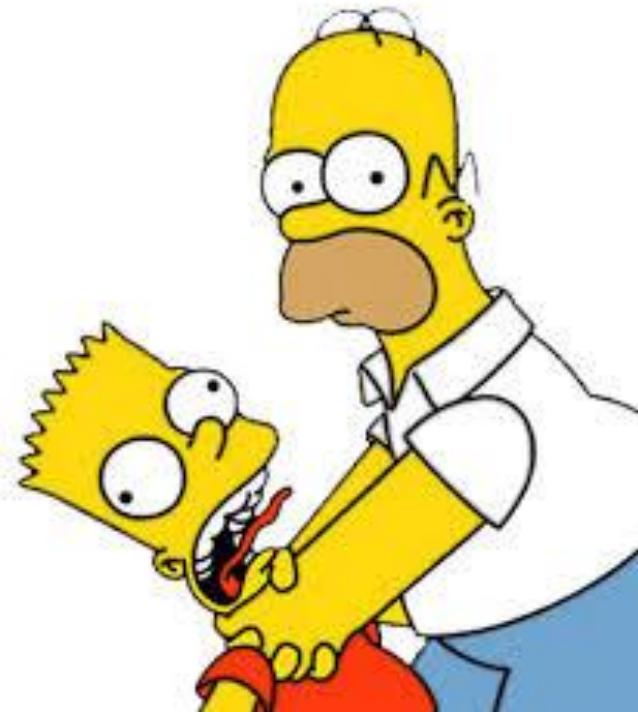
- Reservar espacio en memoria para los objetos creados.
- Liberar la memoria no usada.
- Asignar variables a registros y pilas.
- Llamar al sistema huésped para ciertas funciones, como los accesos a los dispositivos.
- Vigilar el cumplimiento de las normas de seguridad de las aplicaciones Java. Por ejemplo:
 - 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

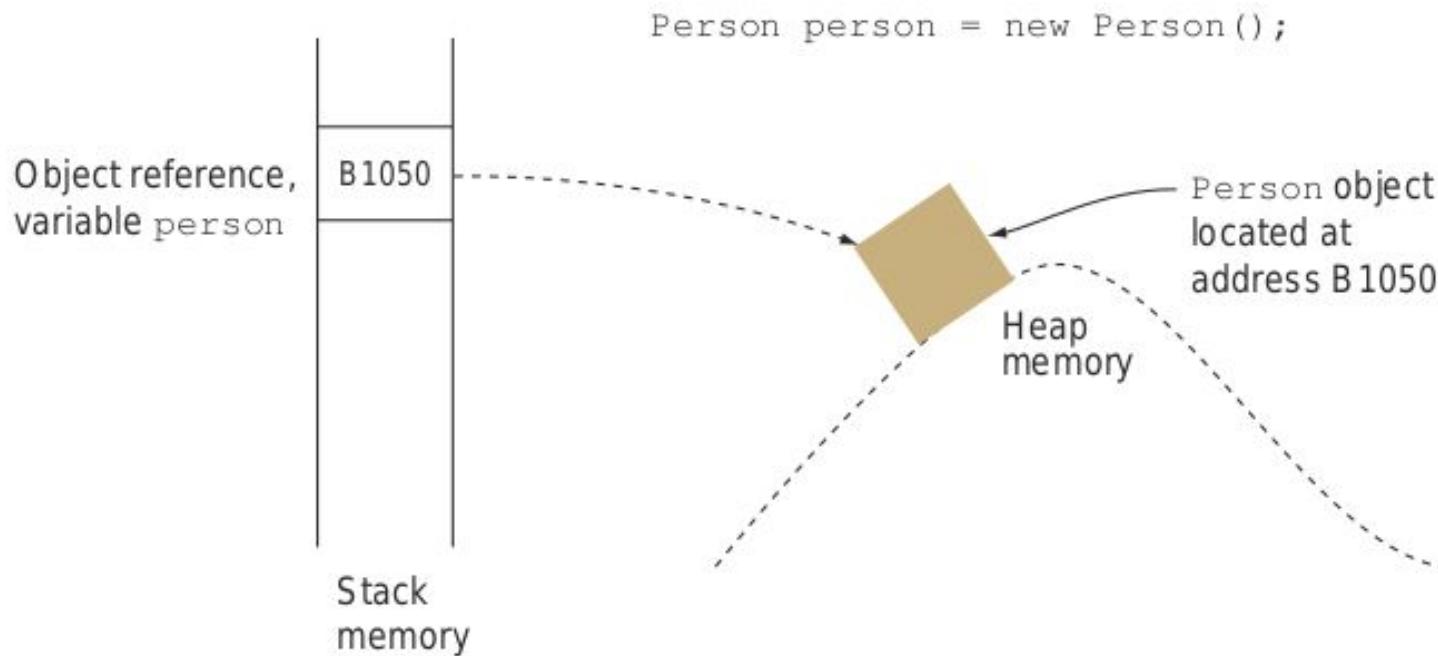


Tipo objeto

- 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

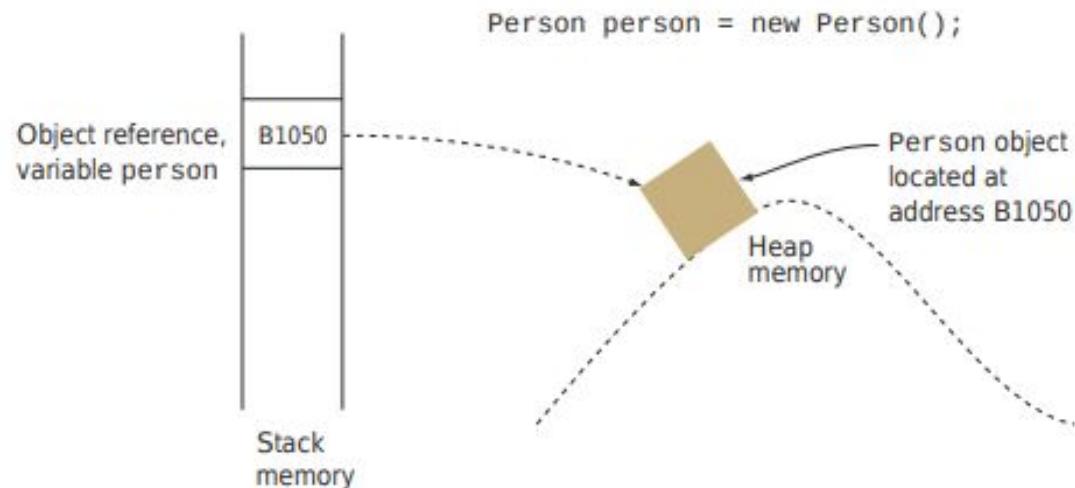
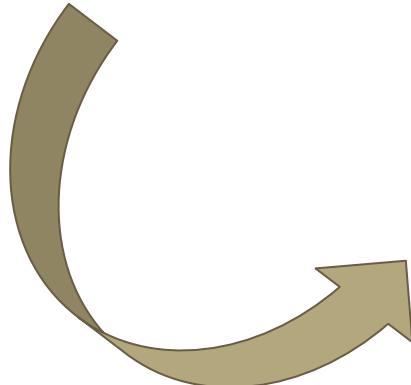
- Variables
 - Clasificación
 - Nomenclatura de identificador
- Tipos primitivos
 - Categorización
 - Ejemplos
 - Valores por defecto
- Java Virtual Machine
 - Secciones
 - Tareas principales
 - Garbage Collector
- Tipo objeto
 - Creación
- **Diferencias entre variables de tipo objeto y tipo primitivo**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



Diferencias entre variables de tipo objeto y tipo primitivo

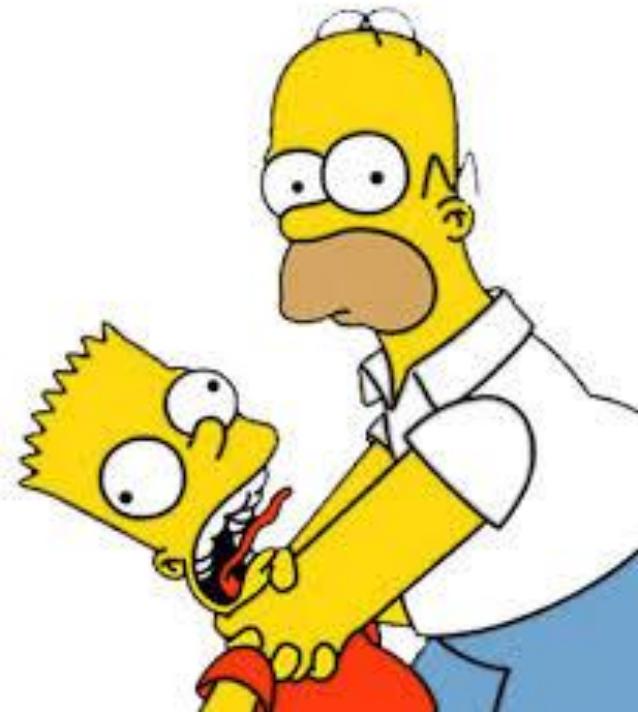
```
int a = 77;
```

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



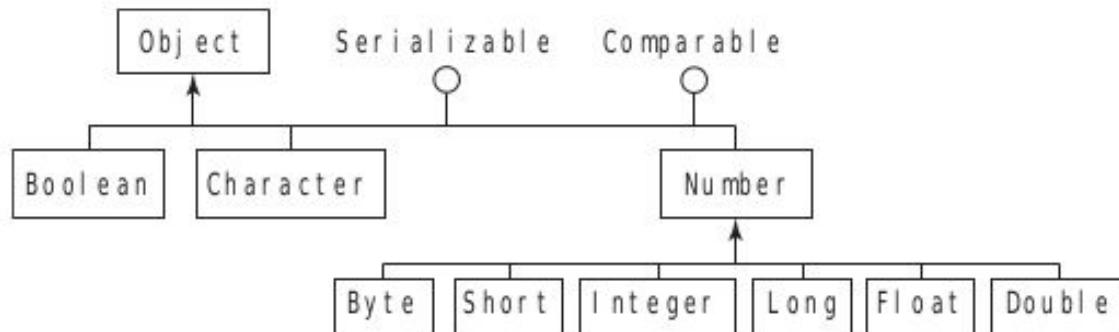
Agenda

- 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';
```

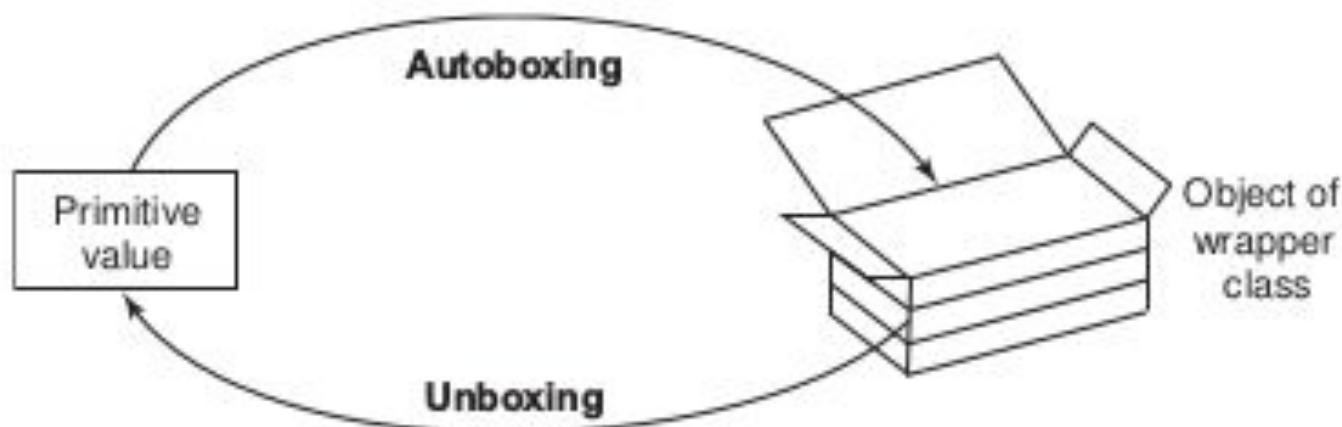
```
Double double1 = 10.98;
```

```
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

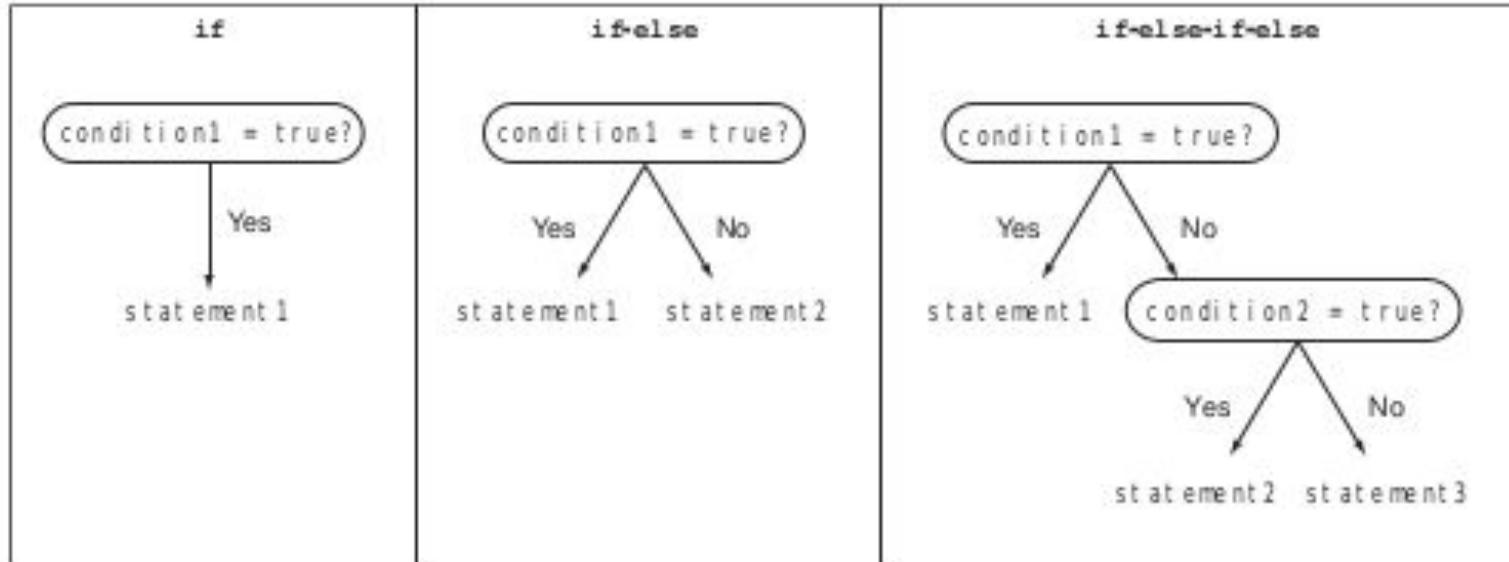
- Control de flujo

- 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;
}

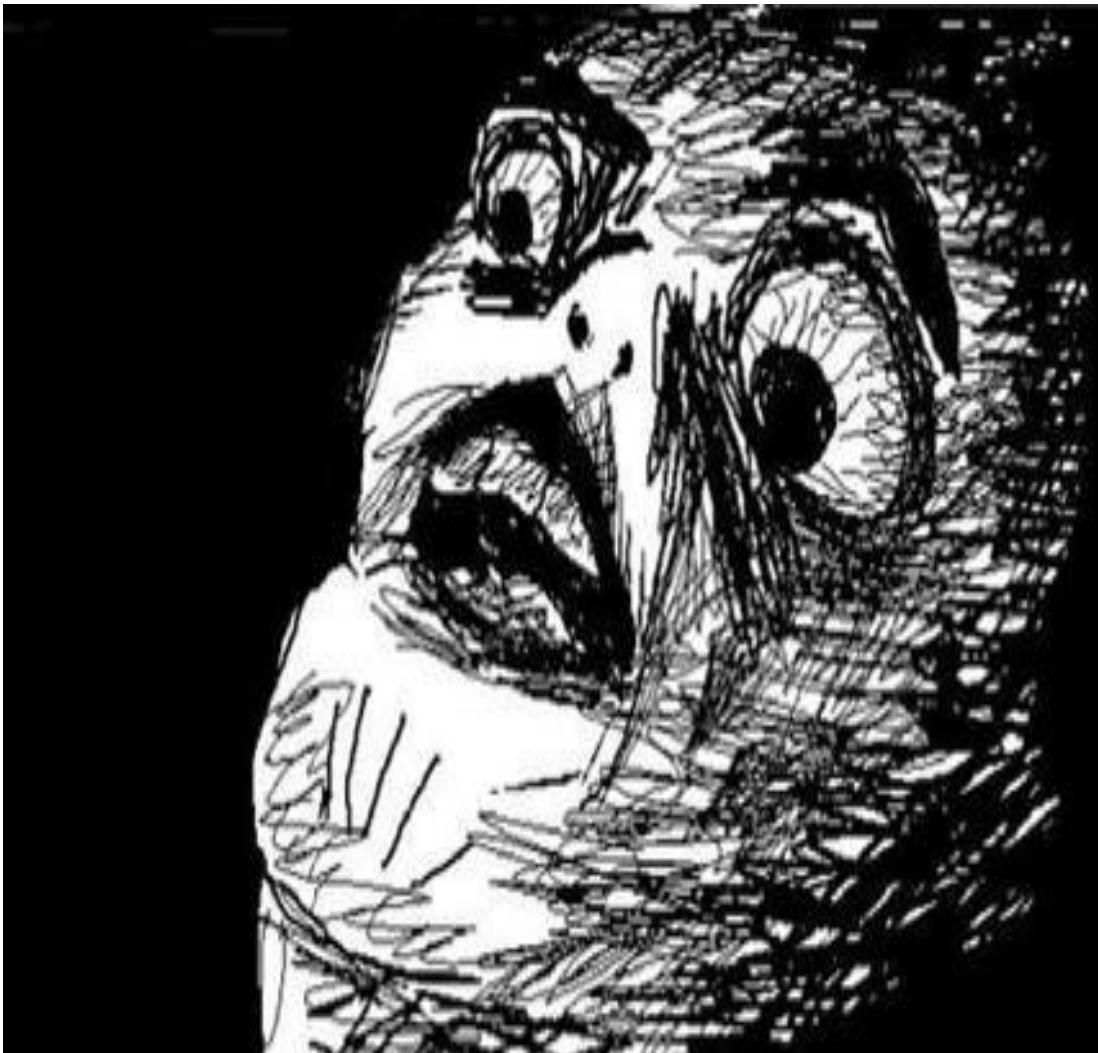
int num = 9;
boolean divisibleBy7 = false;
do {
    System.out.println(num);
    if (num % 7 == 0) {
        divisibleBy7 = true;
    }
    num--;
} while (divisibleBy7 == false);
```

static & non-static

— Programación y Laboratorio III —

Métodos que no usan
valores variables de
instancia.

No es necesario crear
una instancia de la
clase.

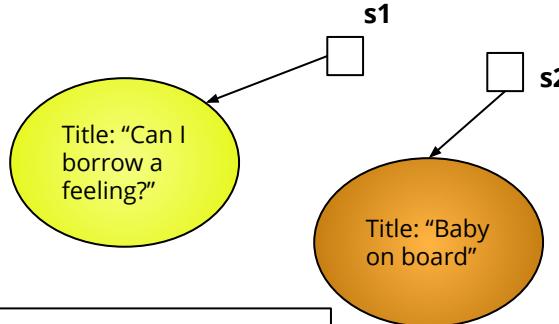
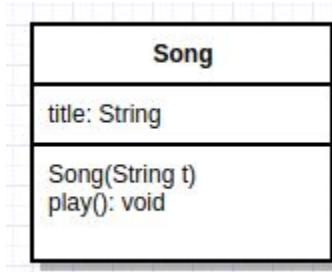


Agenda

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



Método regular (non-static)



```
Song s1 = new Song();
s1.play();
Song s2 = new Song();
s2.play();
```

Variable de instancia

```
public class Song {
    private String title;

    public Song(String t) {
        title = t;
    }

    public void play() {
        SoundPlayer player = new SoundPlayer();
        player.playSound(title);
    }
}
```

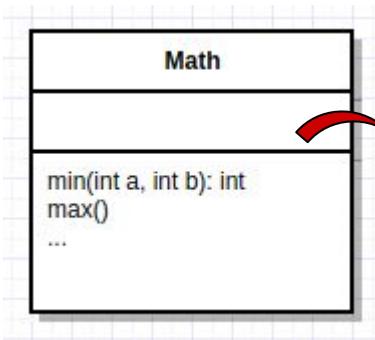


Agenda

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

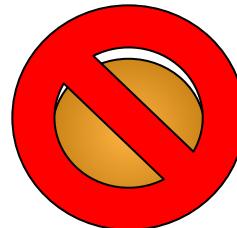


Método estático



No hay variables de
instancia

```
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”

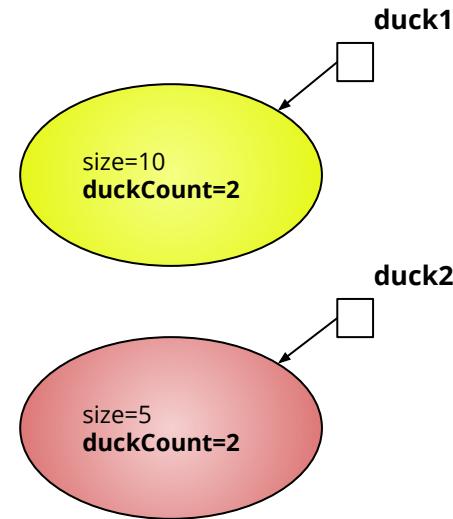


Variables estáticas

```
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

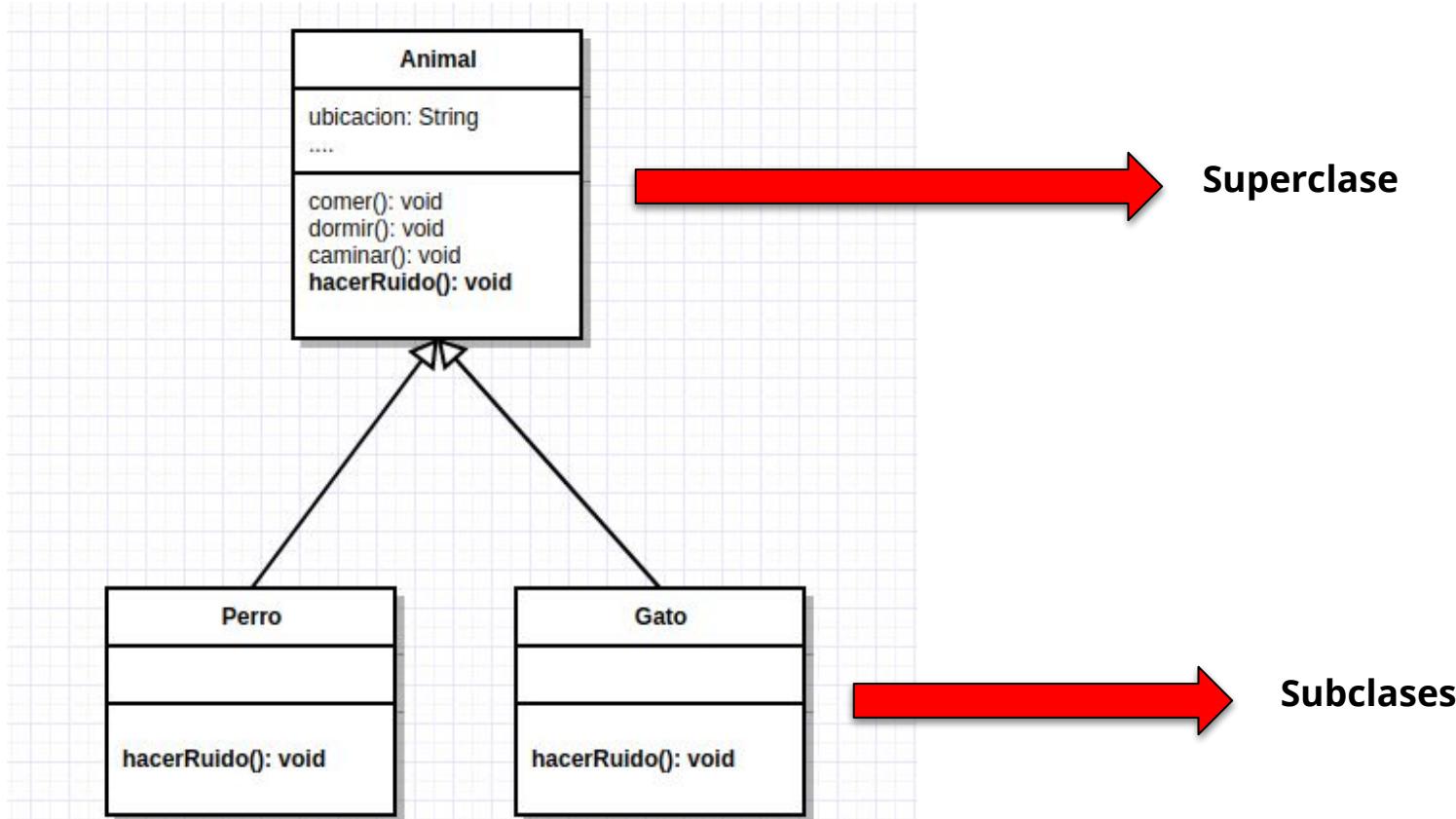
— Programación y Laboratorio III —

Agenda

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



Herencia UML - Ejemplo práctico



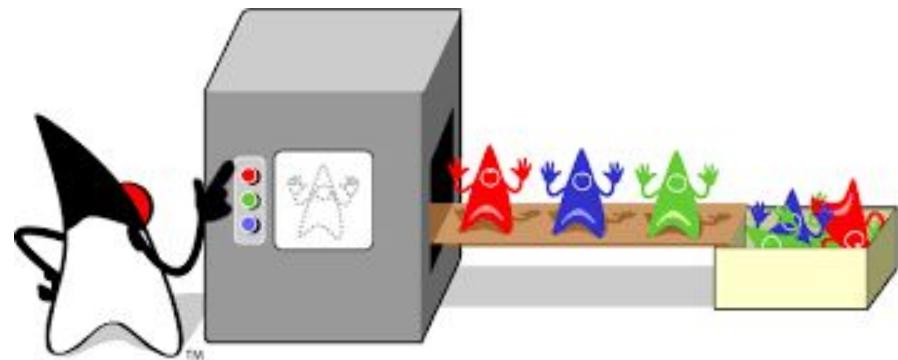
Agenda

- **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.



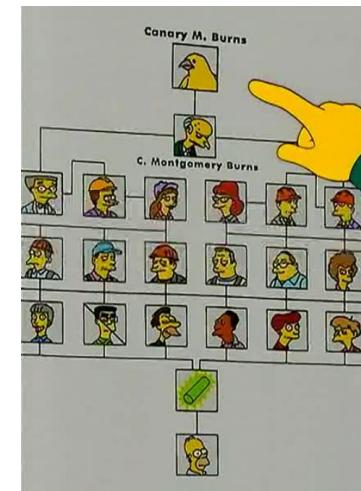
Agenda

- **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

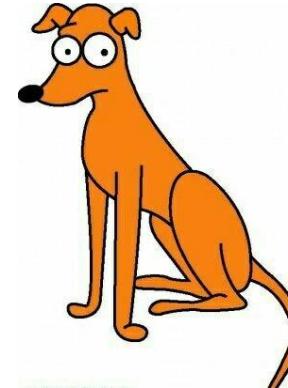


Herencia en Java

```
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

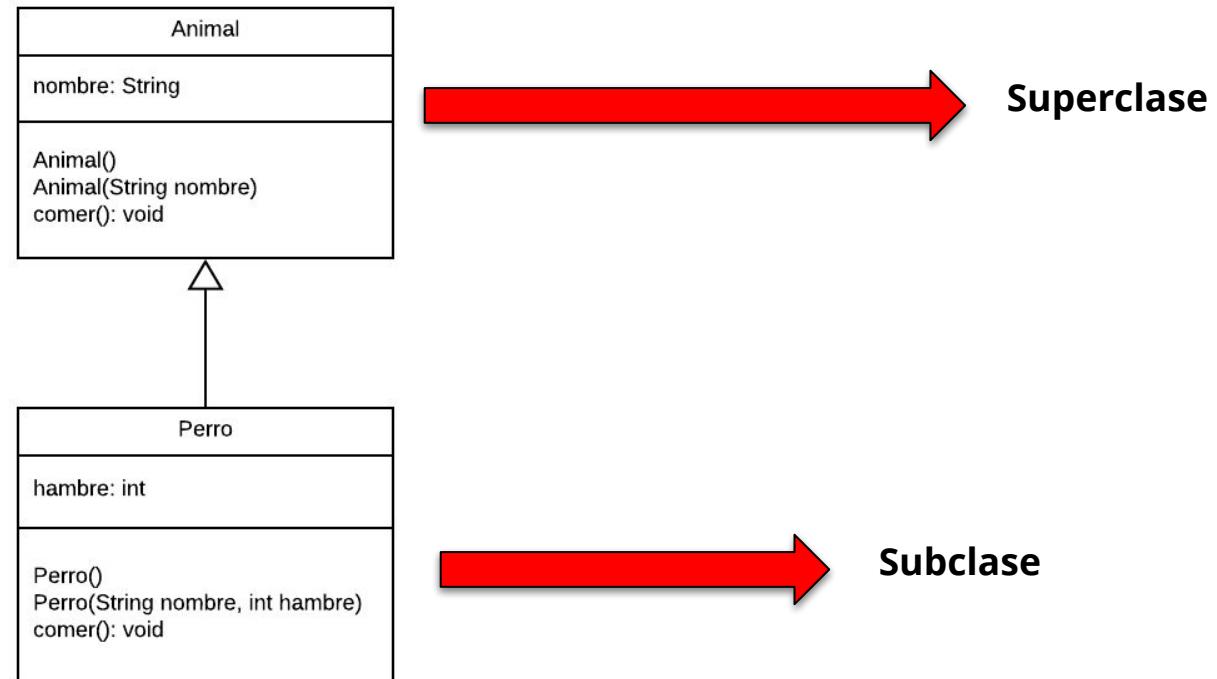
— Programación y Laboratorio III —

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

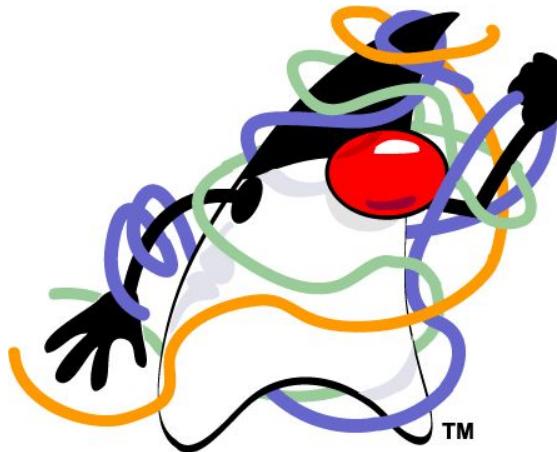
Agenda

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

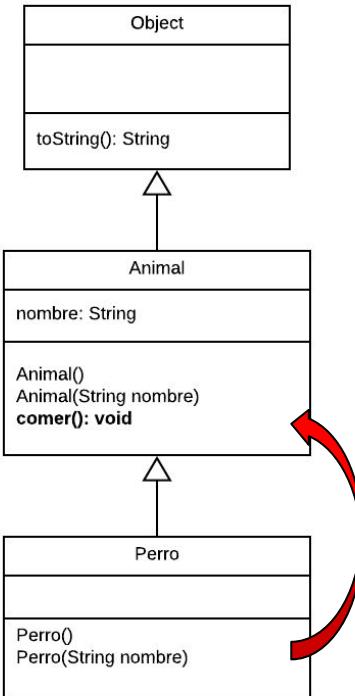


Ejecución dinámica de métodos

- 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)



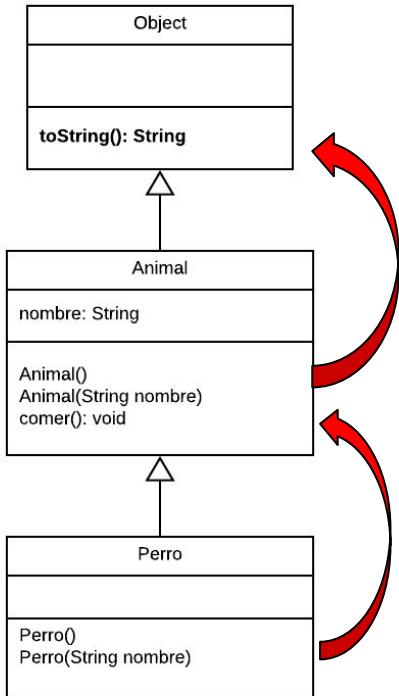
Ejemplo 1: Se quiere ejecutar el método `comer()` de la clase `Perro`.

```
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.

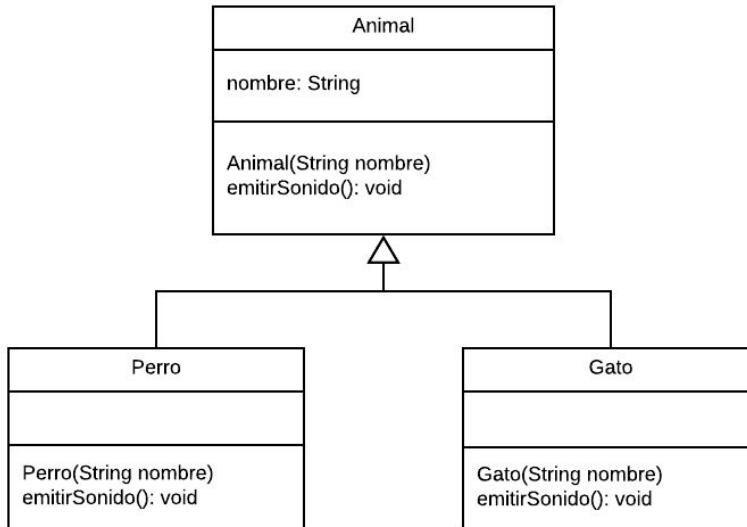
- 1) Se busca el método **`toString()`** en la subclase `Perro`.
- 2) Como no se encuentra la sobreescritura, se sube en la jerarquía hasta que se encuentra y se ejecuta.

Agenda

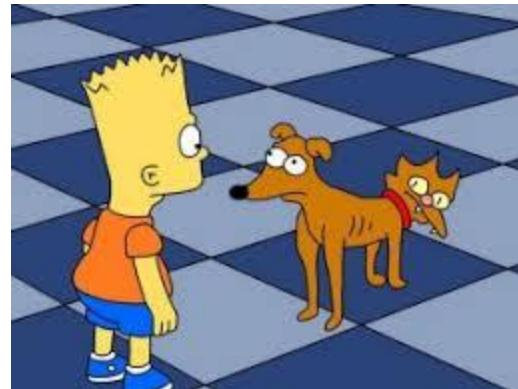
- ~~Ejemplo UML~~
- ~~Palabra reservada super~~
 - Ejemplo
- ~~Modificador de acceso: protected~~
- ~~Ejecución dinámica de métodos~~
- **Polimorfismo**
 - Definición
 - Ejemplo
 - Warning
- UML - Ejemplo completo para practicar



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

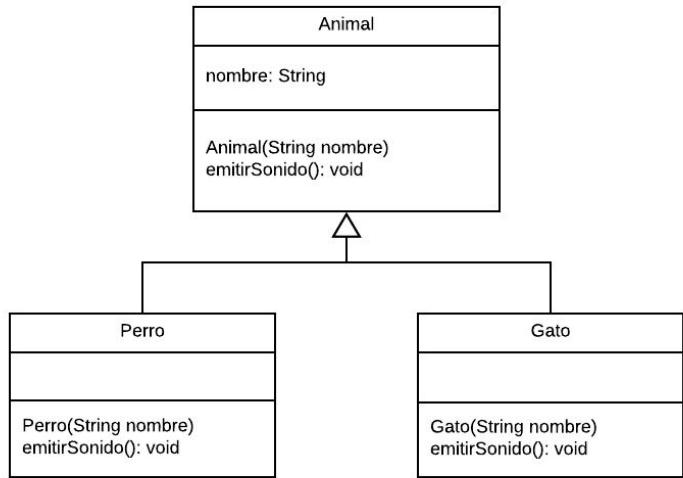


Polimorfismo - Ejemplo

```
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!");  
    }  
}
```

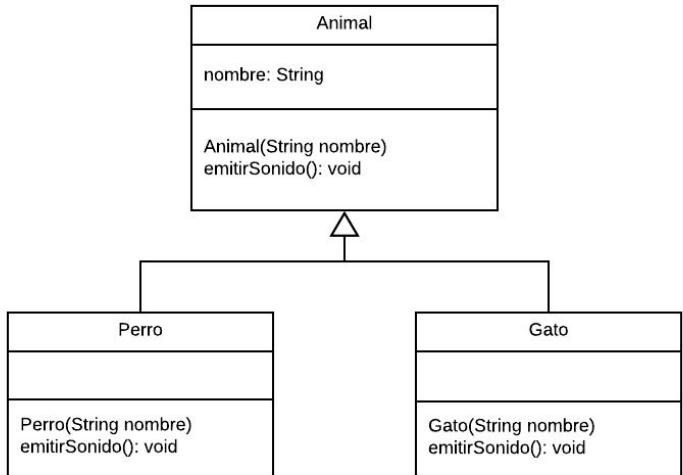
Polimorfismo - Ejemplo (2)



```
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();
```



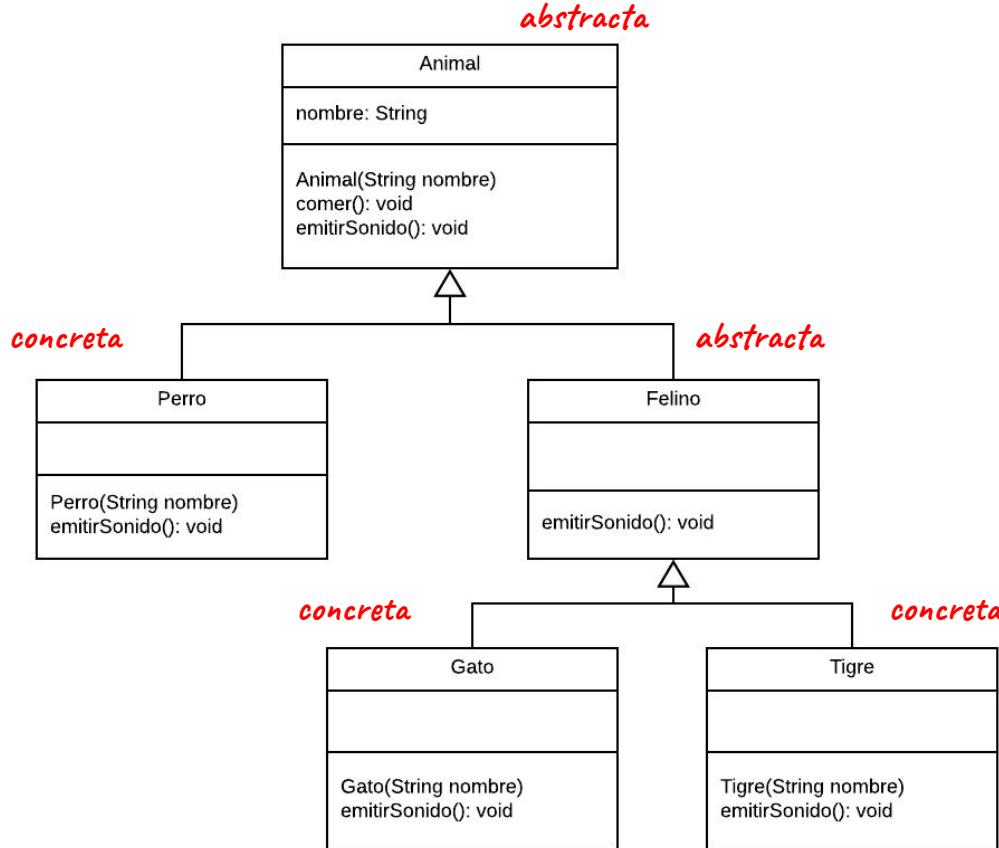
El método no está definido en la clase Animal!!

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



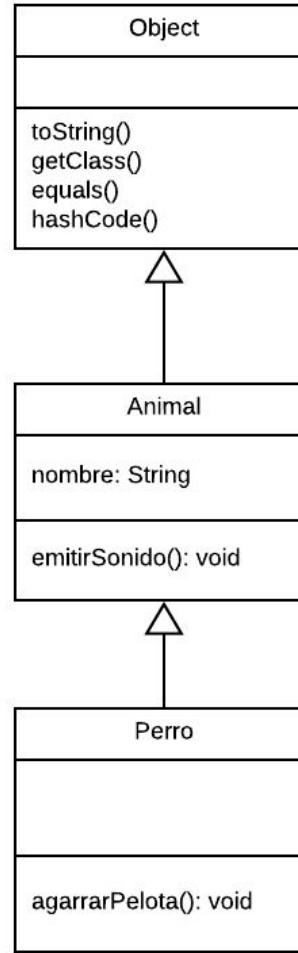
Bibliografía oficial

- <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

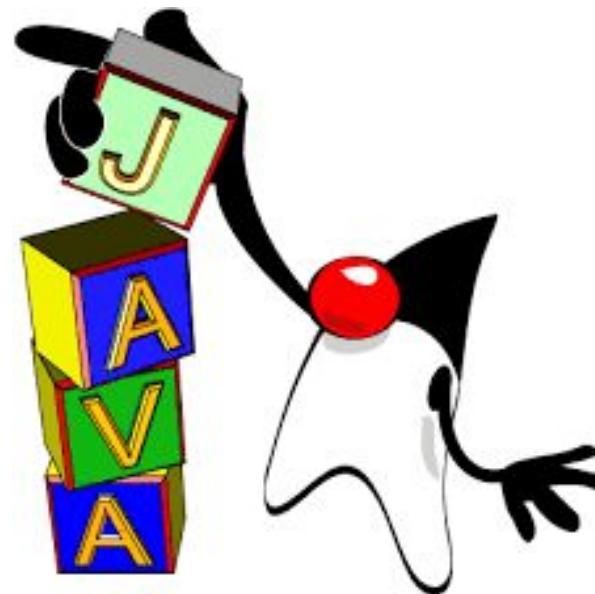
— Programación & Laboratorio III —

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

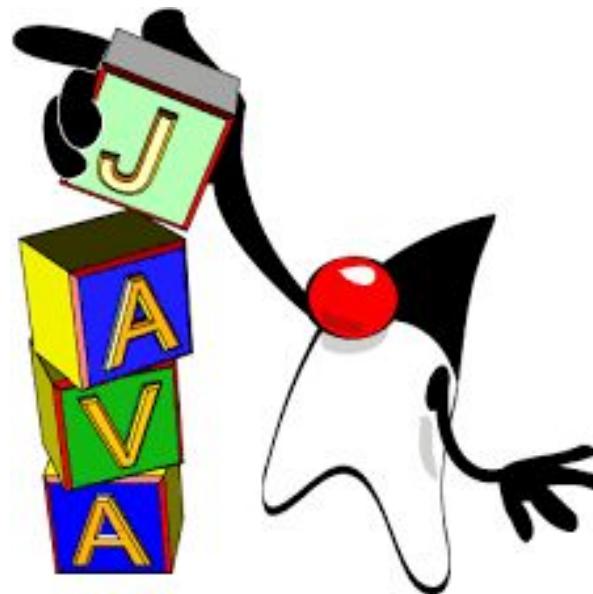
Polimorfismo con Object - Ejemplo (2)

```
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");

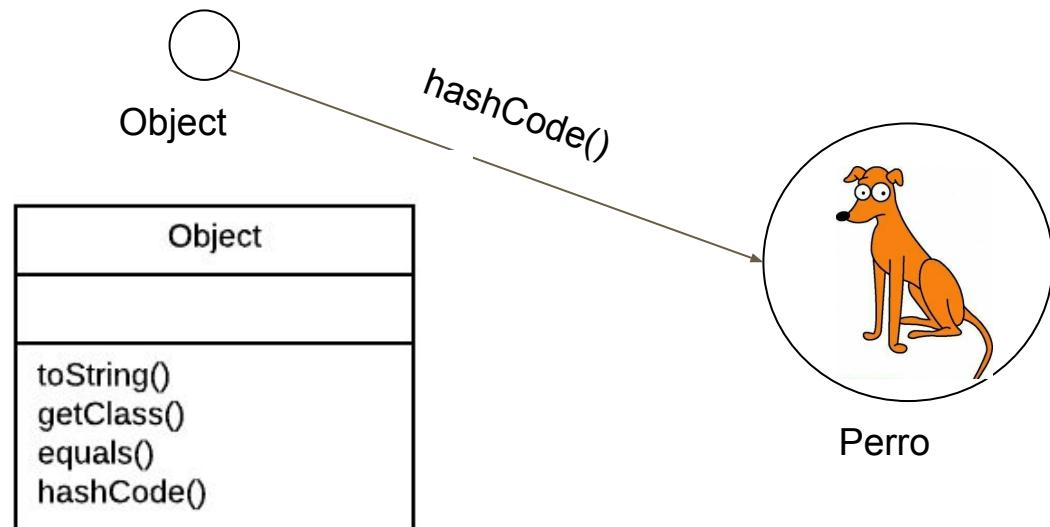
objects[0] = object;
objects[1] = perro;

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

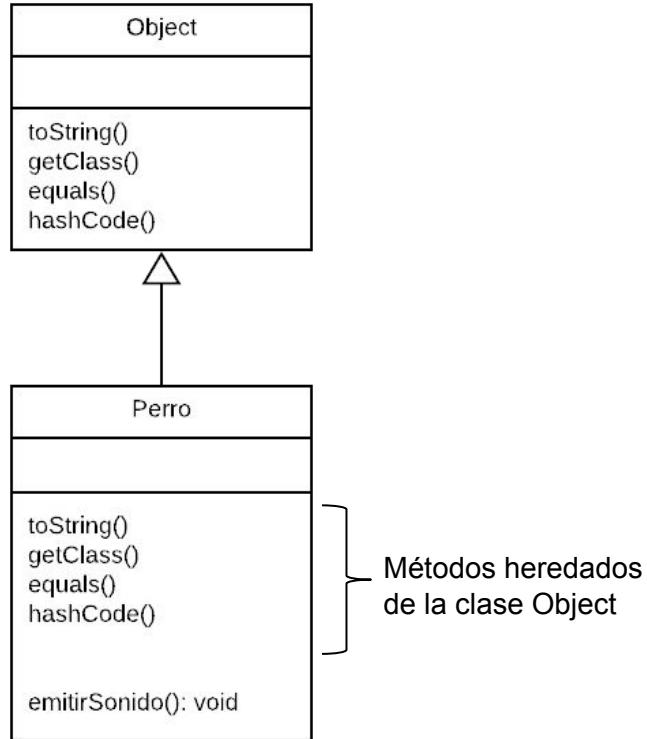
Retorna una referencia de Object de una instancia de Perro

Referencias polimórficas (3)

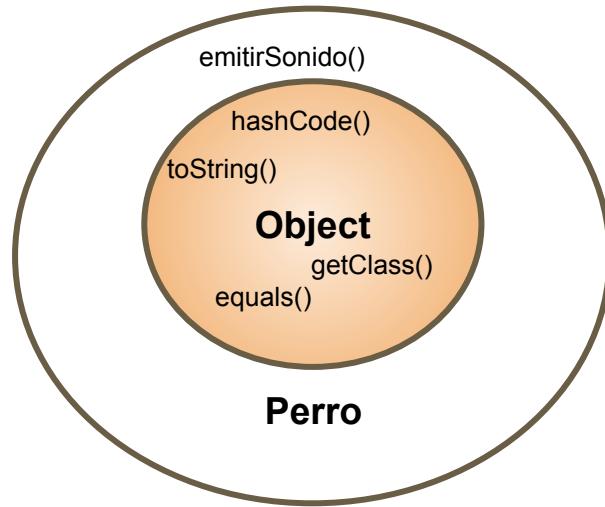
- El método que se invoca debe estar en la clase del tipo declarado, sin importar el tipo del que realmente es el objeto.
- El compilador verifica el tipo de la referencia, no el tipo del objeto.



Referencias polimórficas (4)



objeto Perro

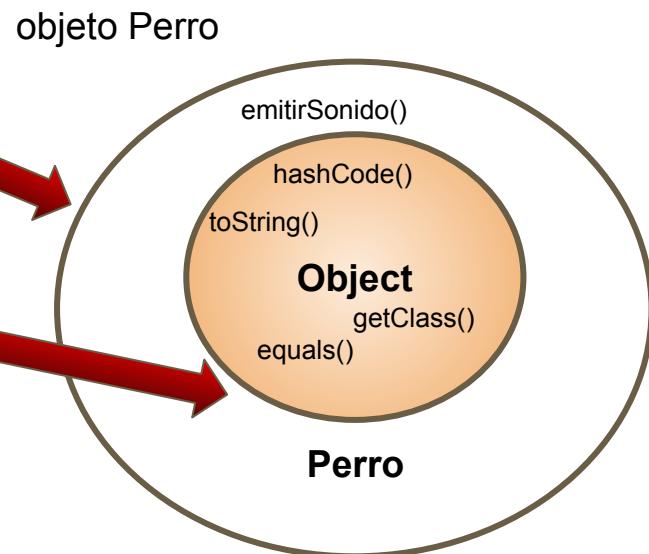


- El objeto Perro contiene la clase **Perro** como parte suya y también la clase **Object**.

Referencias polimórficas (5)

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

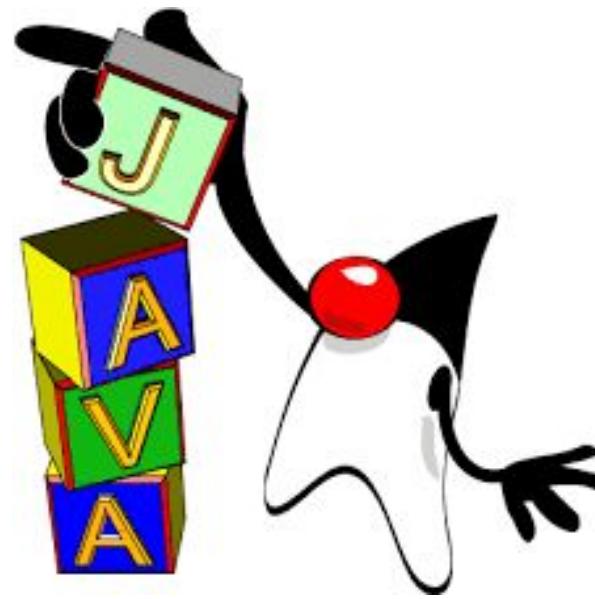
```
Object o = p;
```



- 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];  
    }  
}
```

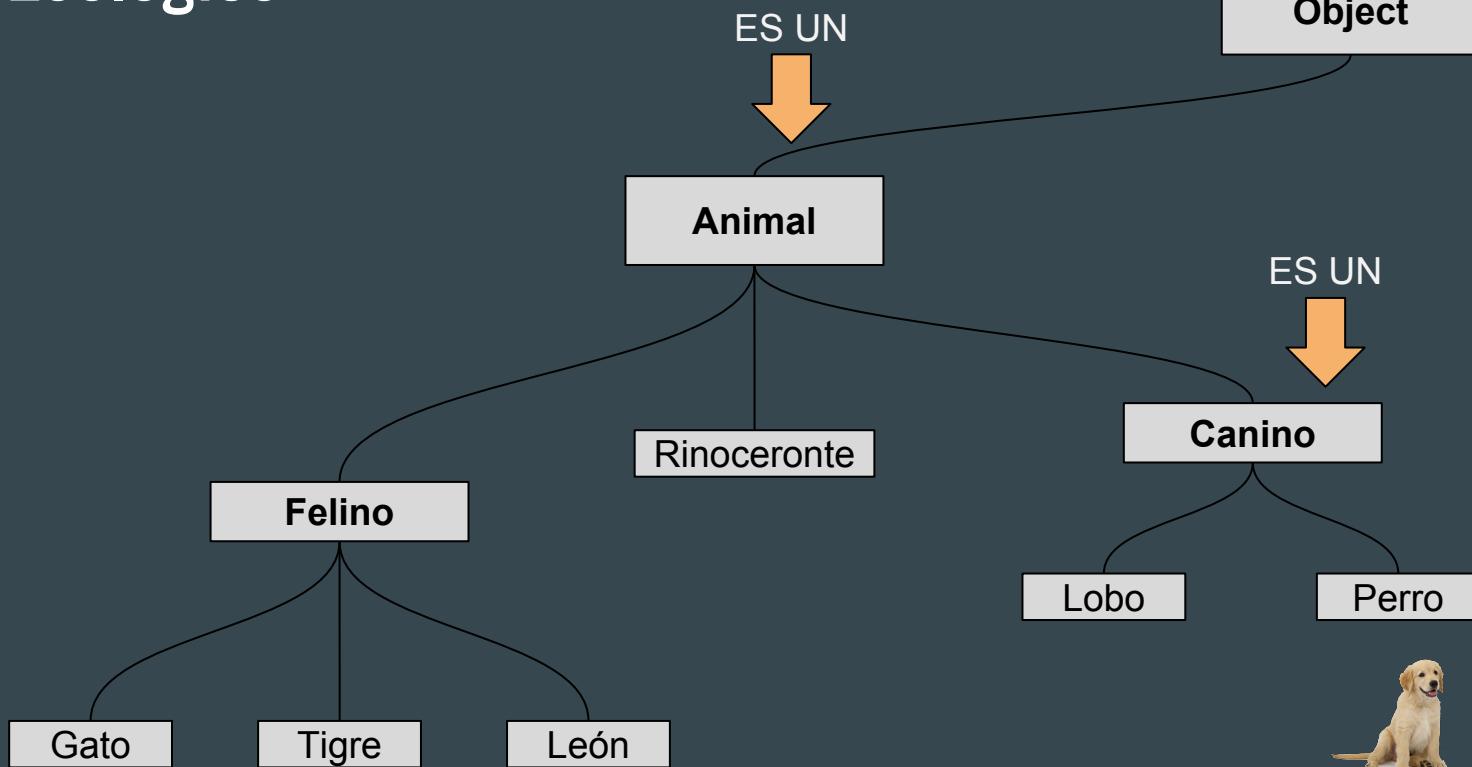
Bibliografía oficial

- <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>

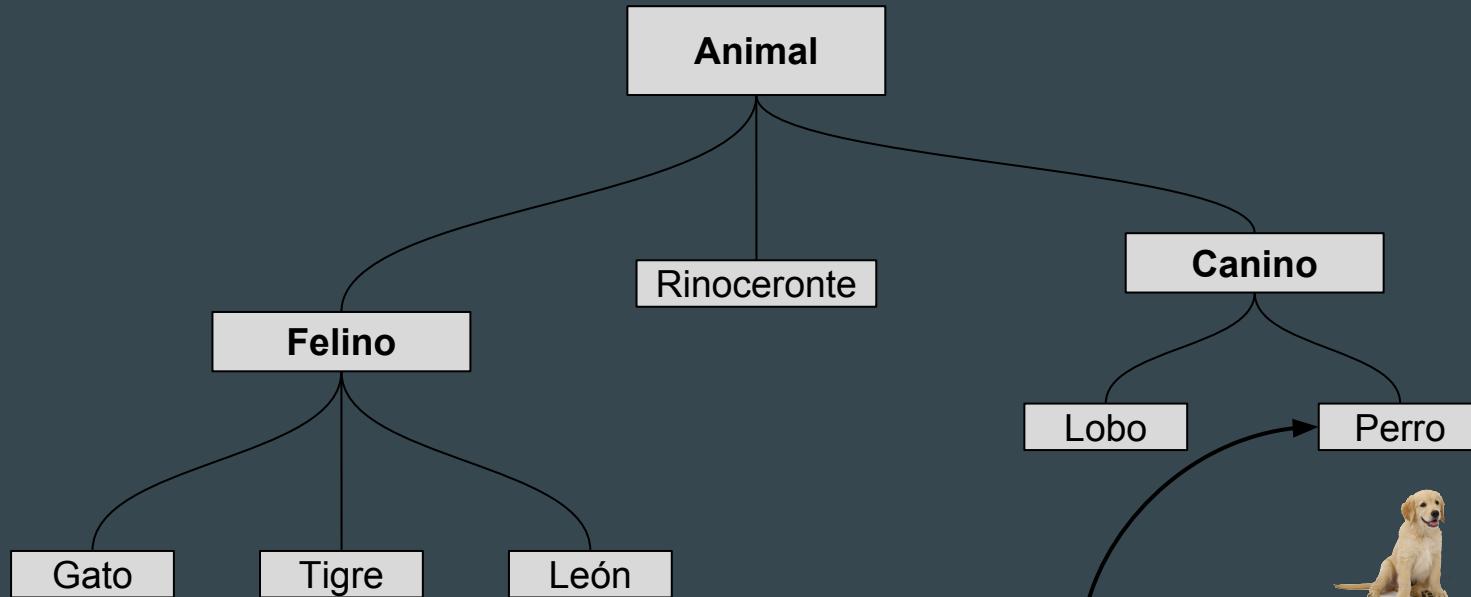
Interfaces

• • •

UML Zoológico



UML Veterinaria

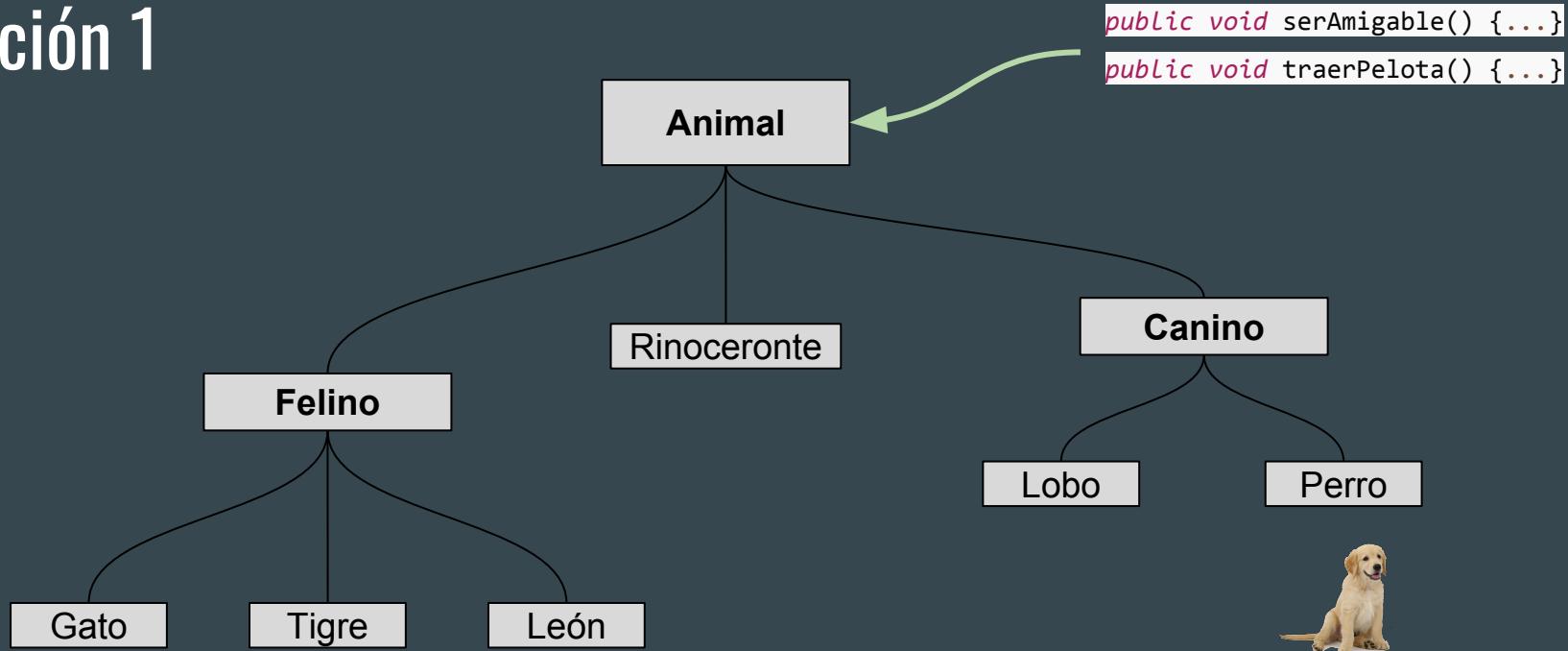


¿Comportamiento de mascota?

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

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

Solución 1



Pros:

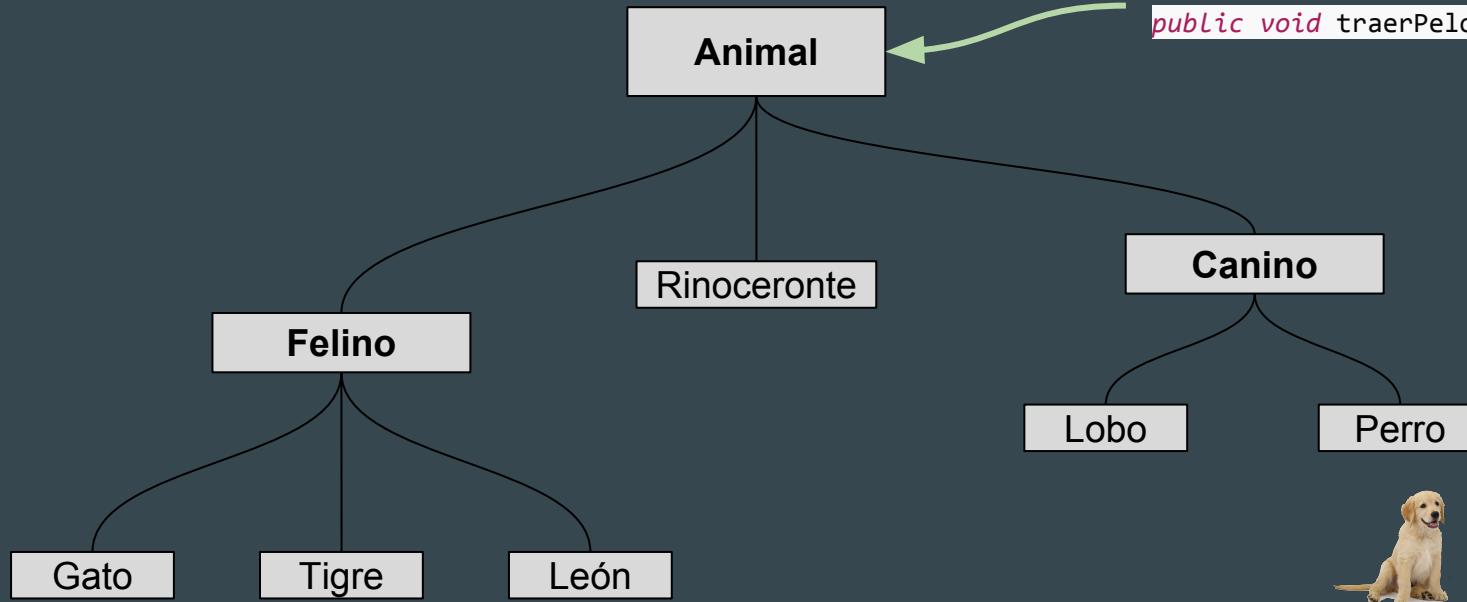
- Todos los animales heredan el comportamiento de mascota.
- No hay que tocar el comportamiento de las subclases existentes.
- Las futuras clases implementadas podrán hacer uso de los nuevos métodos.
- **Animal** puede ser utilizado como tipo polimórfico.





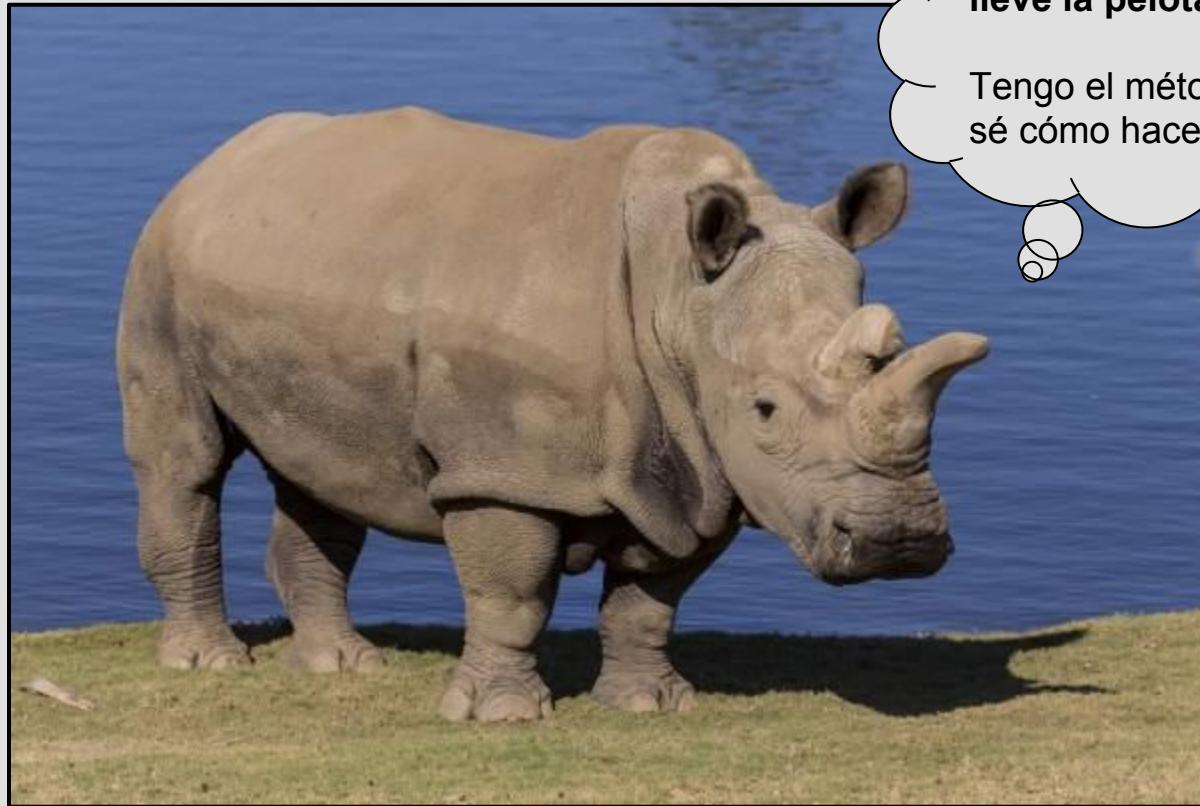
Solución 2

¿Métodos abstractos en la superclase?



Pros:

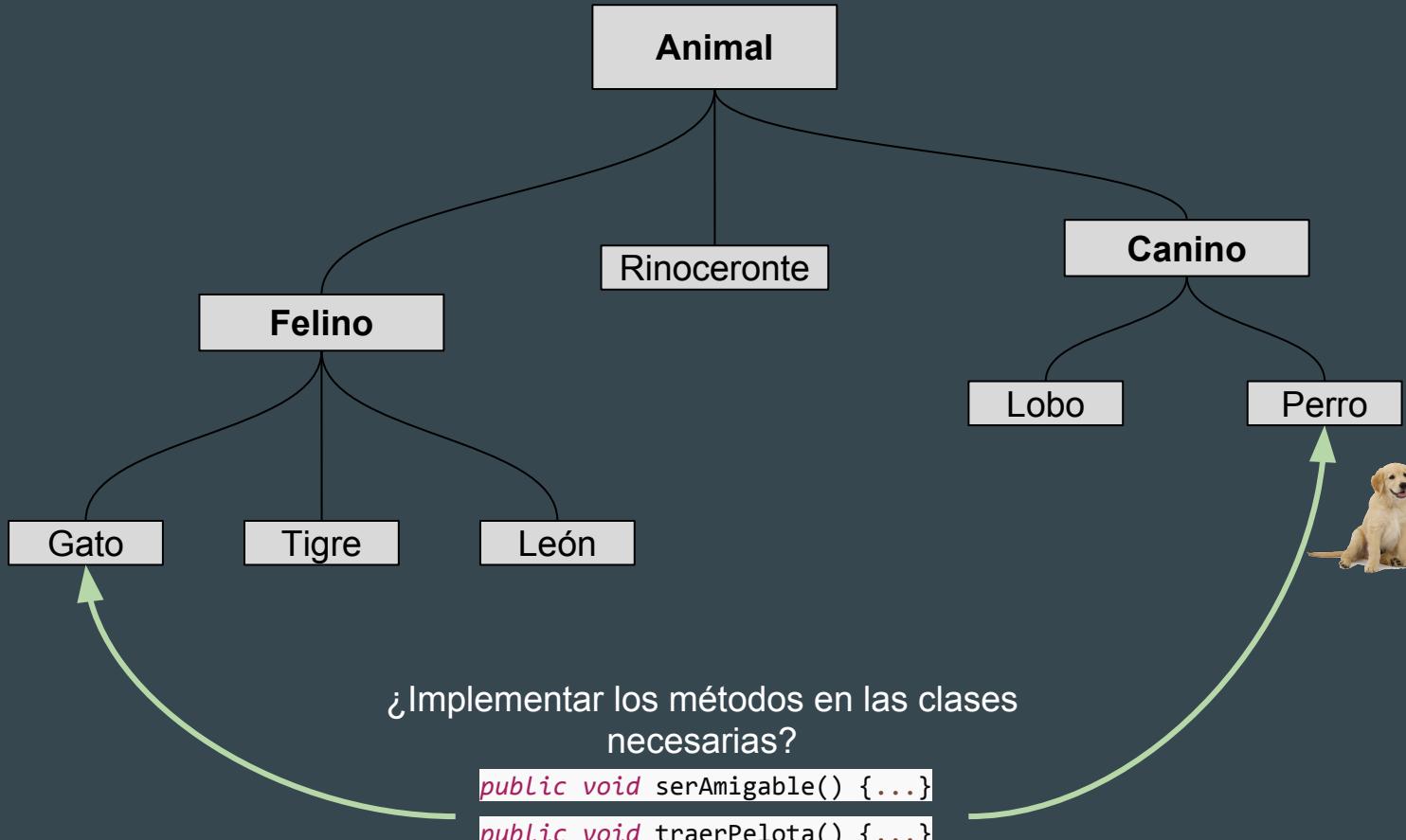
- Todos los beneficios de la solución anterior.
- Se pueden implementar métodos que no hagan nada para los animales que no se consideran mascotas.



.- ¿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.

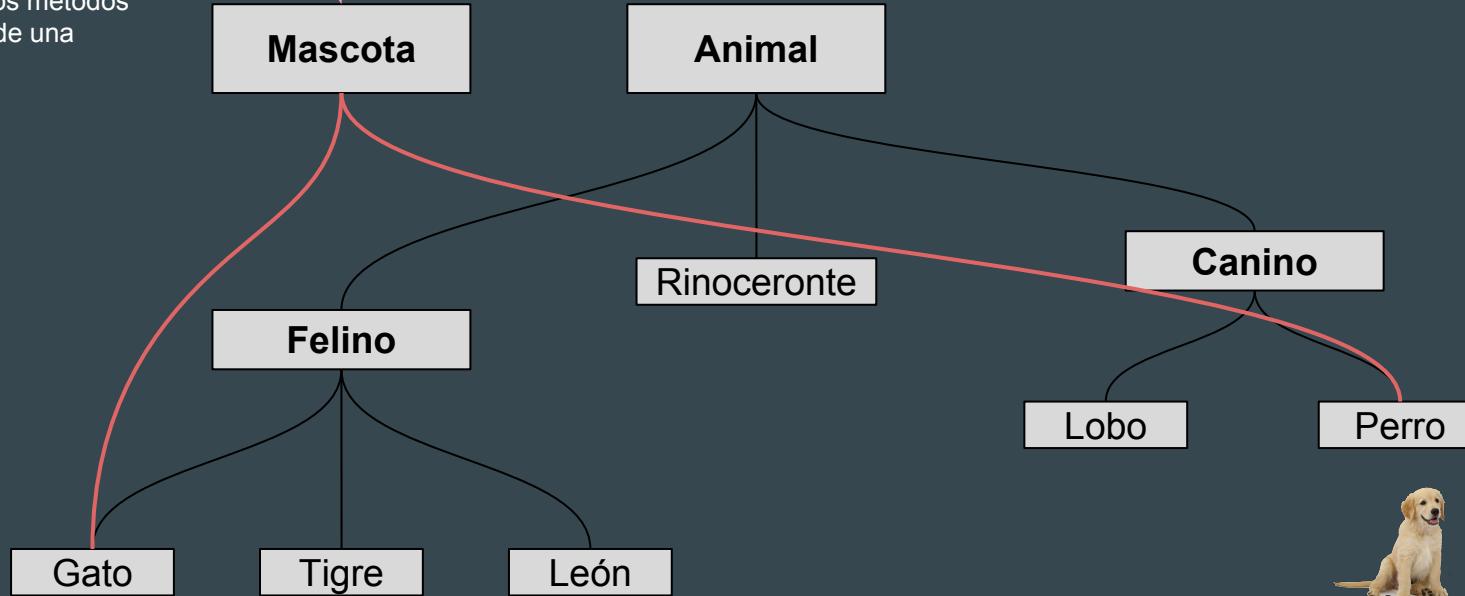
Contras:

- 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?

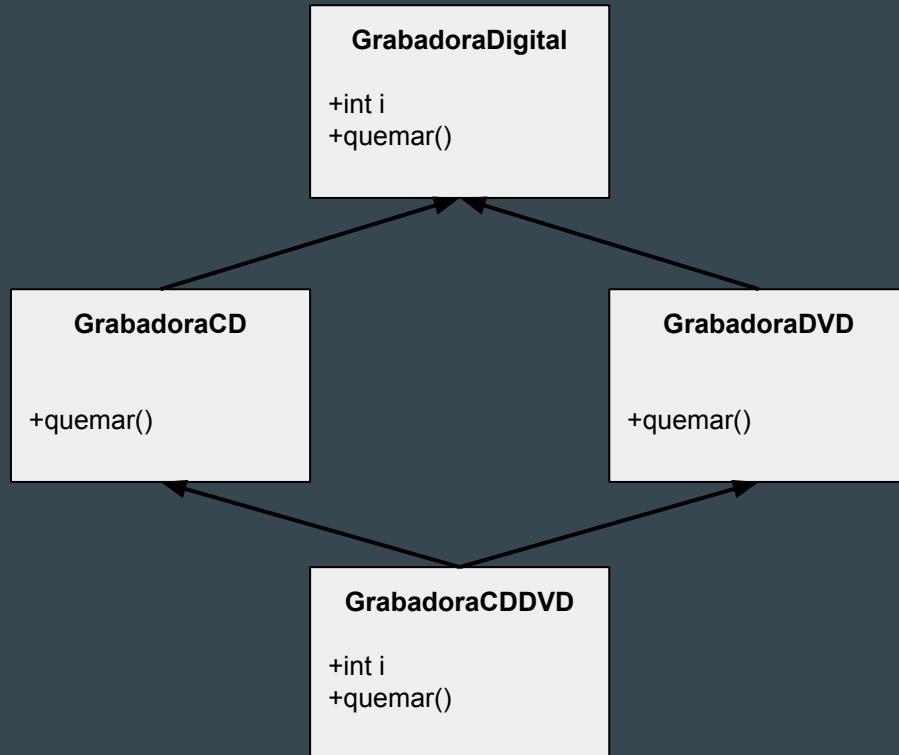
- Una forma de tener el comportamiento de mascotas solo en las clases necesarias.
- Una forma de garantizar que todos los métodos definidos poseen la misma signatura, un “contrato”.
- Una forma de tomar ventaja del polimorfismo.

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

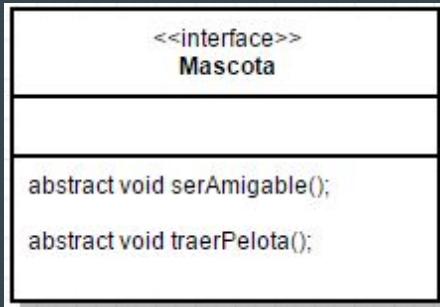


¿Herencia múltiple?

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.

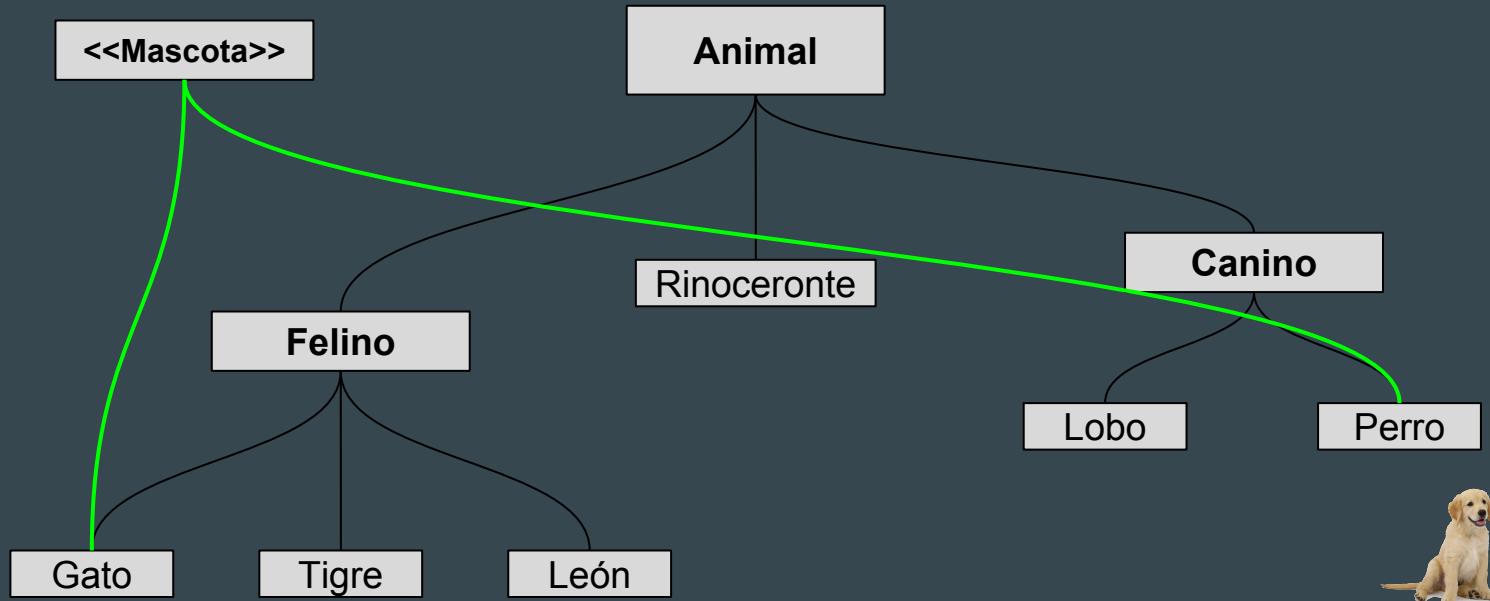
```
package com.utn.learning.interfaces;

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(Comparable 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 —

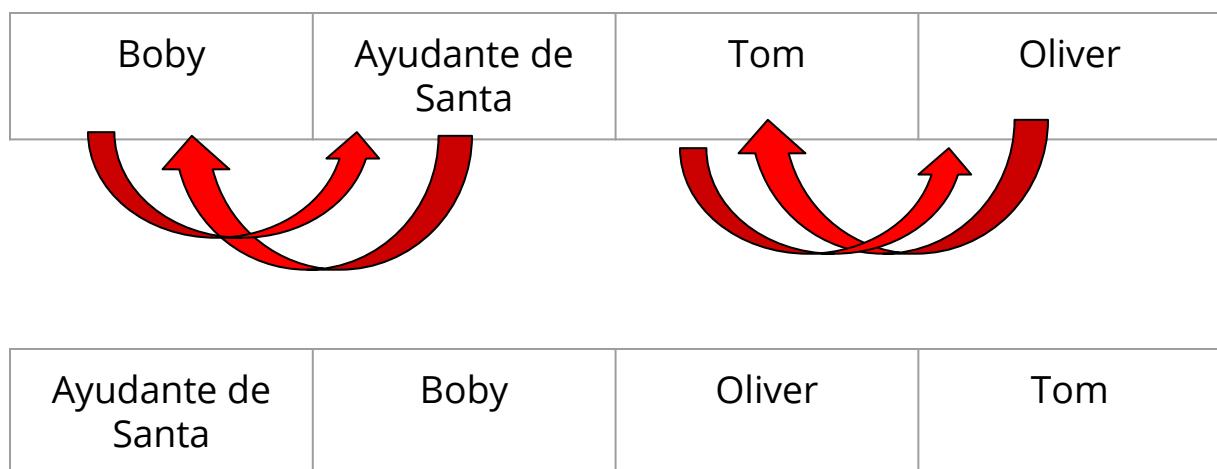
Ejemplo

```
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;  
  
}
```

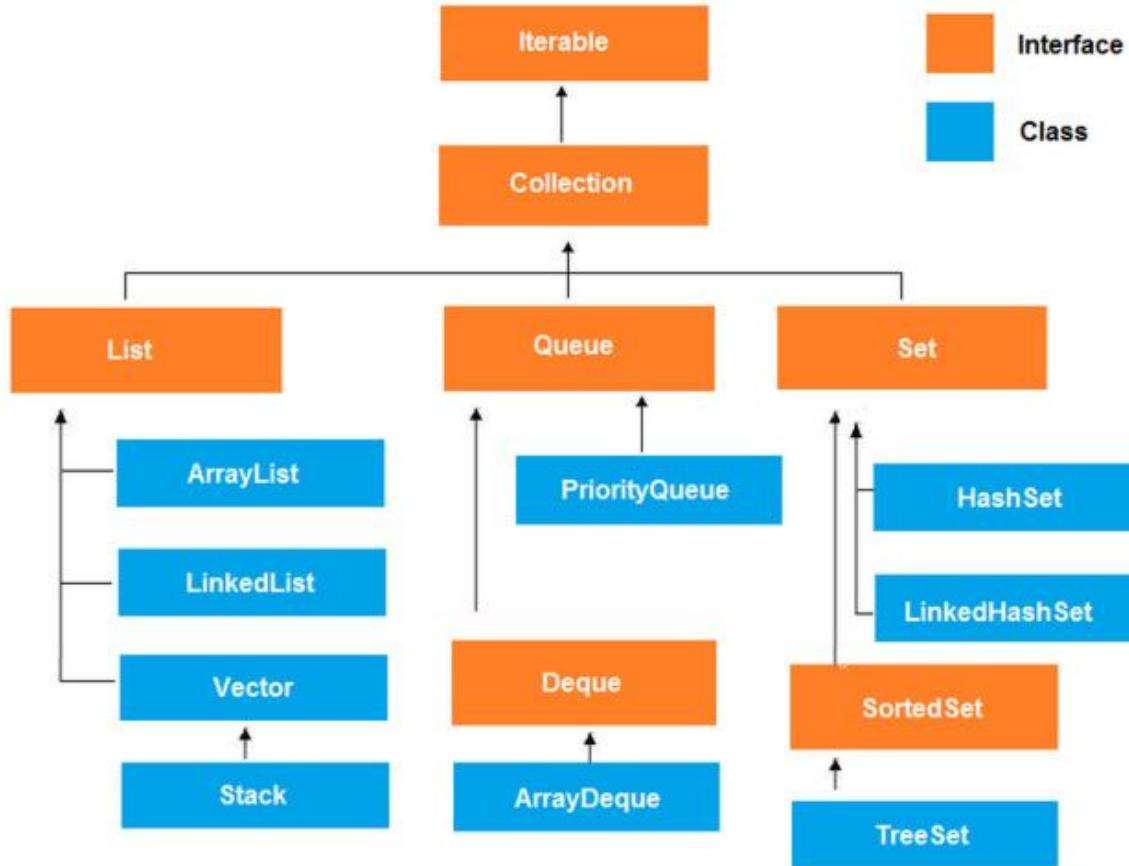
PREVIOUSLY ON...

Problema: Ordenar el arreglo?

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

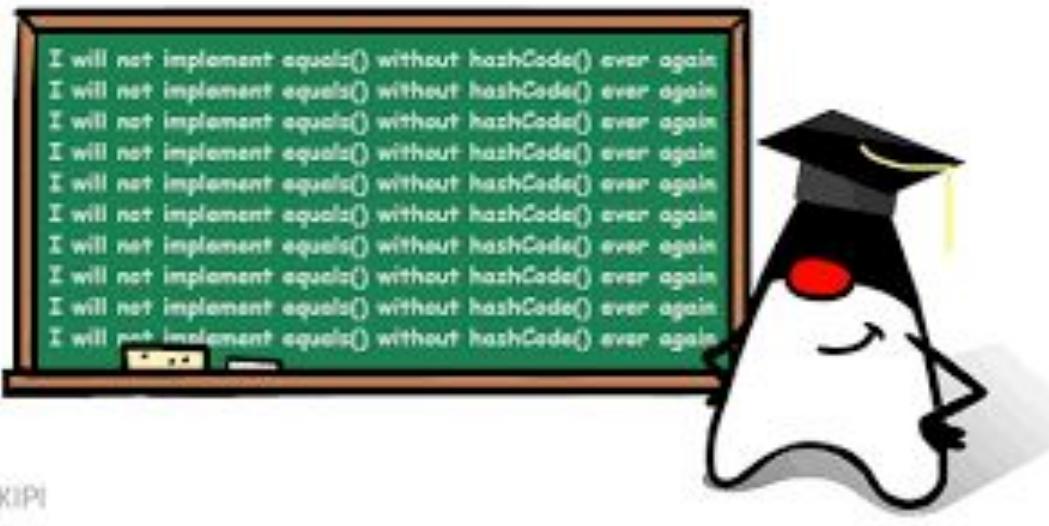


Solución: Collection API



Agenda

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



TAKIPI
TAKIPI

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.

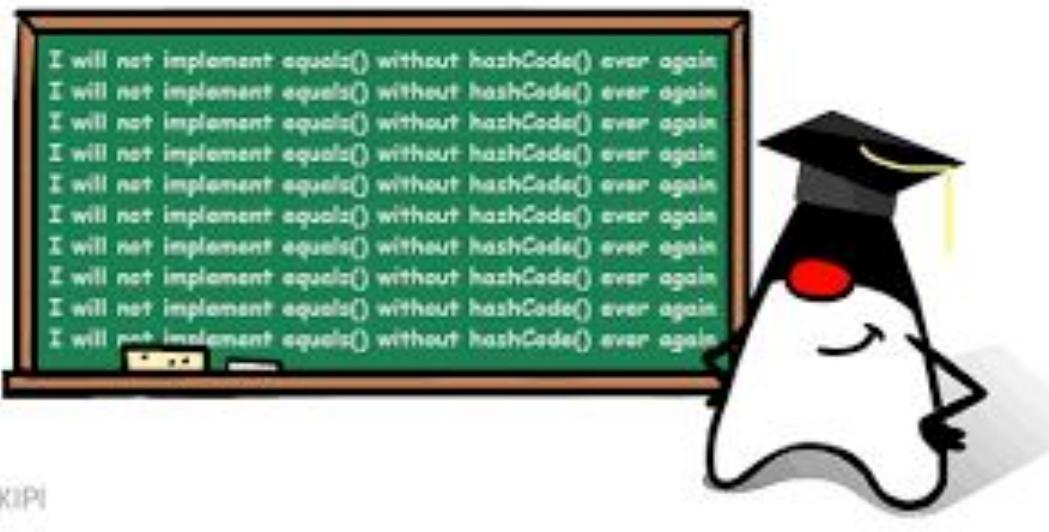


- Reduce los esfuerzos de programación al proveer estructuras de datos y algoritmos que no tenemos que escribir.
- Provee implementaciones de alta performance.
- Fomenta la reutilización del software al proporcionar una interfaz para colecciones y algoritmos para manipularlas.

Agenda

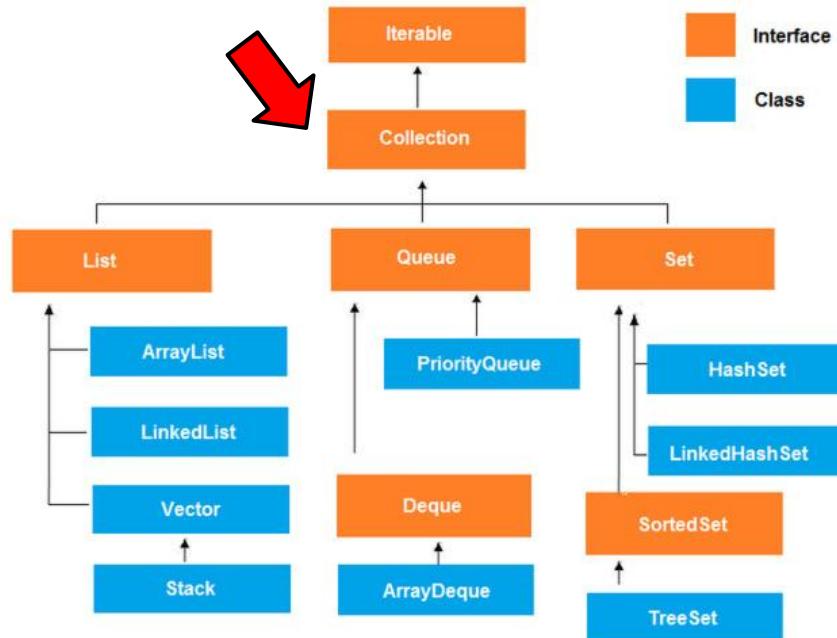
— Collection API

- **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**

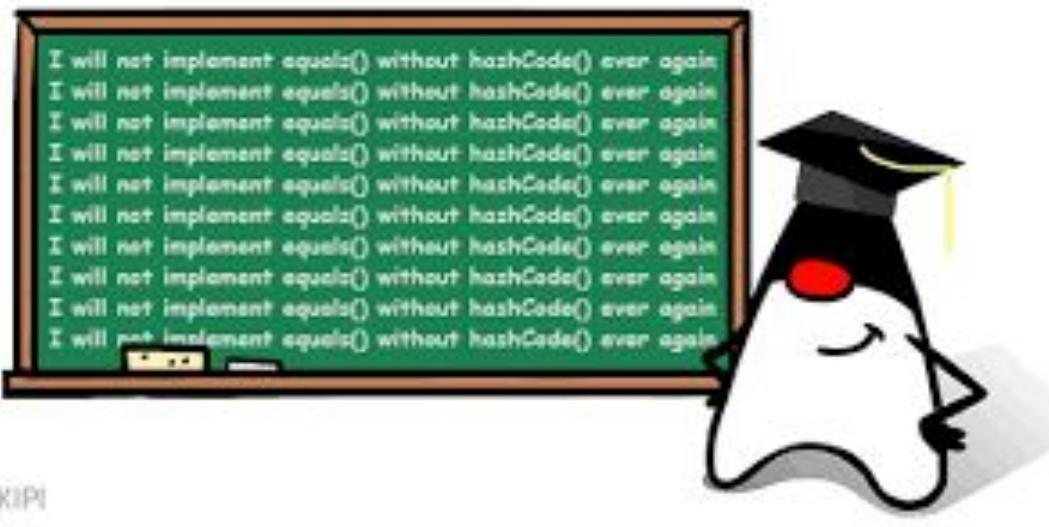
- **ArrayList**

- Insertar elemento
 - Lectura de un elemento
 - Eliminar elemento

- Método **equals()**

- Método **hashCode()**

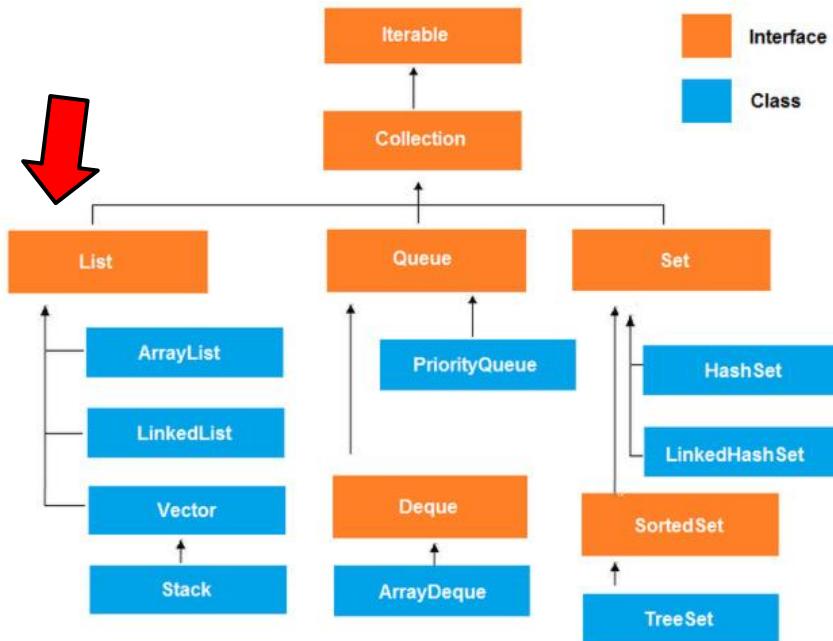
- Método **equals()** - Ejemplo



TAKIPI
TAKIPI

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

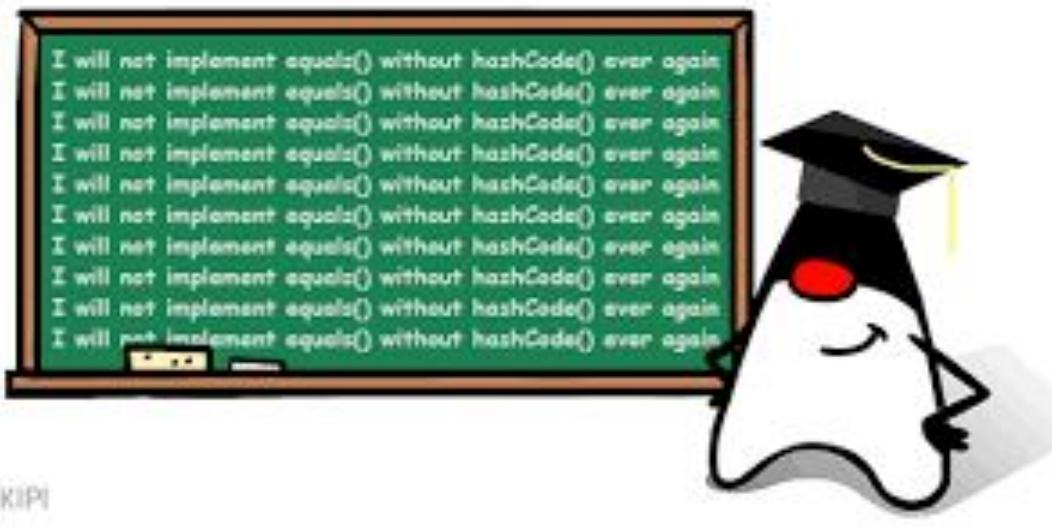
- **ArrayList**

- Insertar elemento
- Lectura de un elemento
- Eliminar elemento

- Método equals()

- Método hashCode()

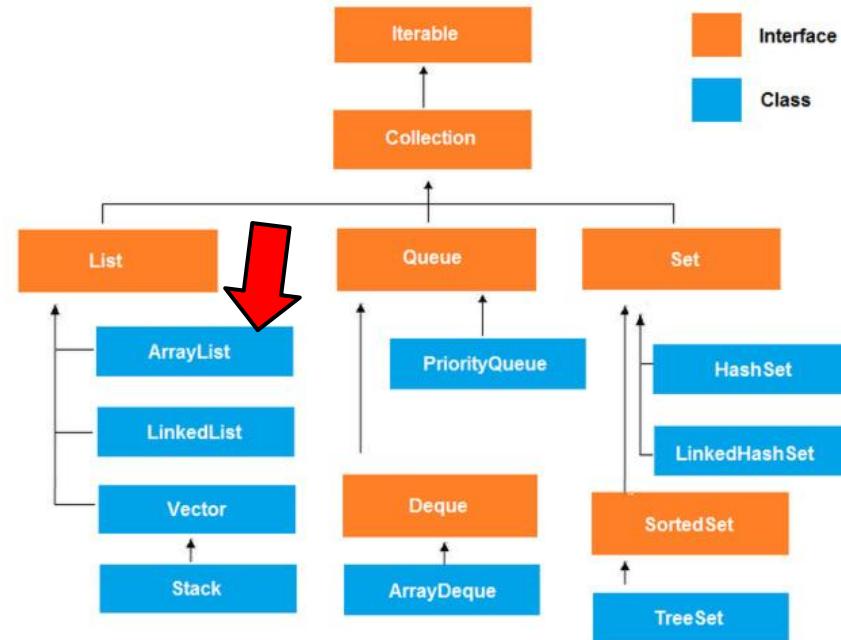
- Método equals() - Ejemplo



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

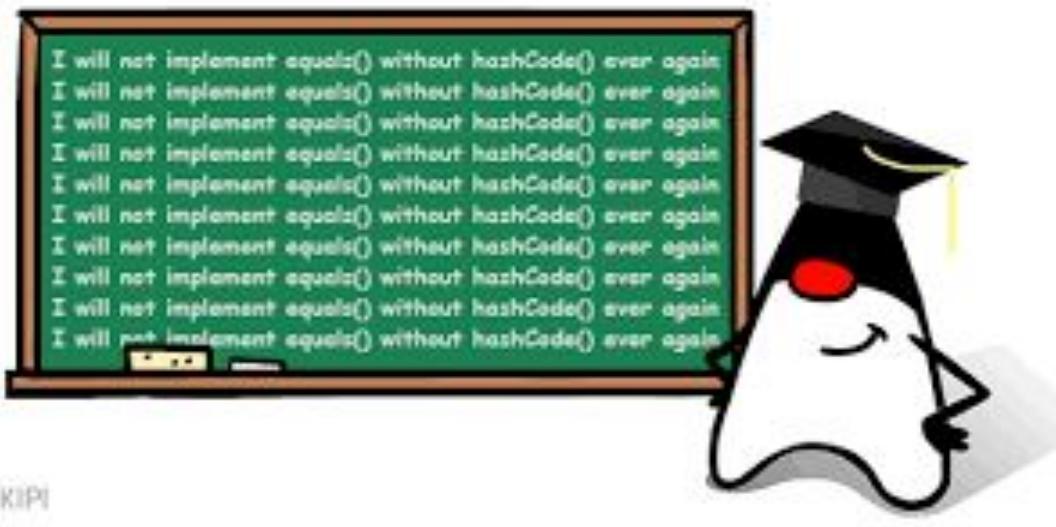
- **ArrayList**

- **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);  
}  
C  
Antes de agregar un elemento  
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.

perro1 perro2 null null null null null null null null null

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.

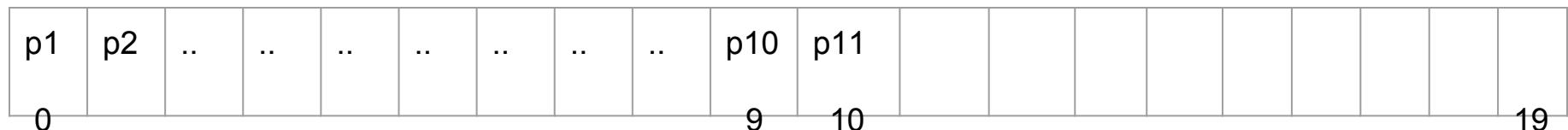
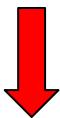


0

9

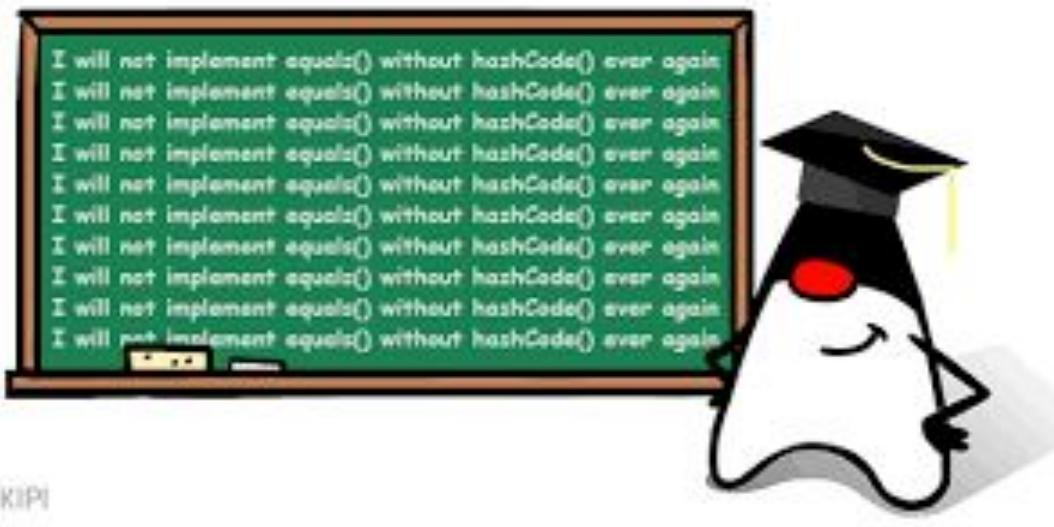
p11

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.

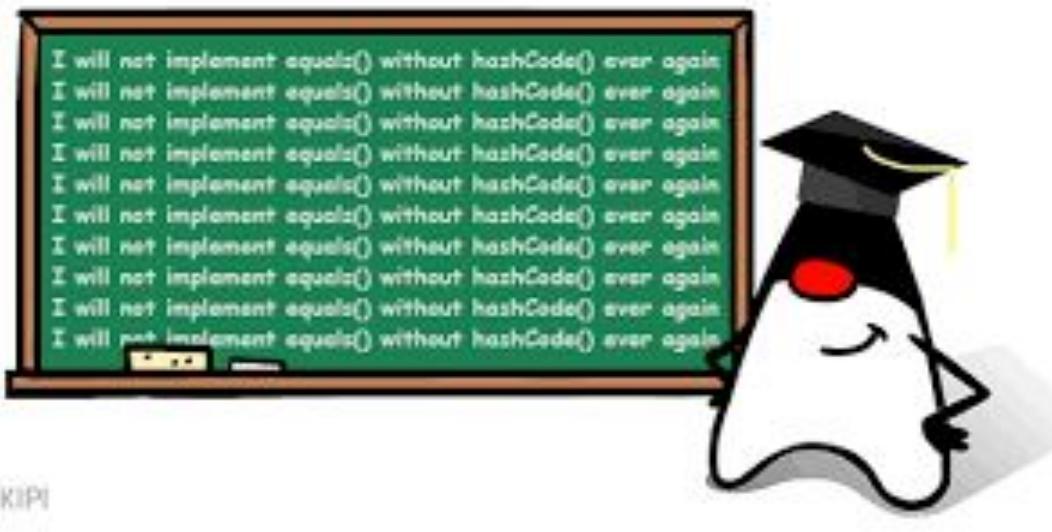
```
public static void main(String [] args) {  
  
    List<Perro> perros = new ArrayList<>();  
  
    Perro p1 = new Perro("Boby");  
    Perro p2 = new Perro("Ayudante de Santa");  
  
    perros.add(p1);  
  
    System.out.println(perros.get(0).getNombre());  
}
```



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

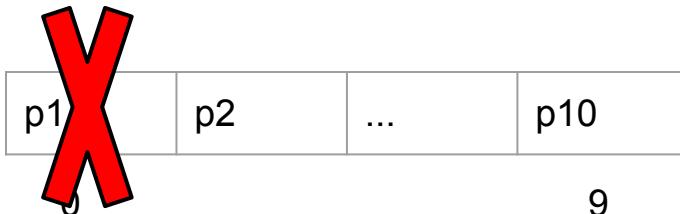


ArrayList - Eliminar elemento

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))`



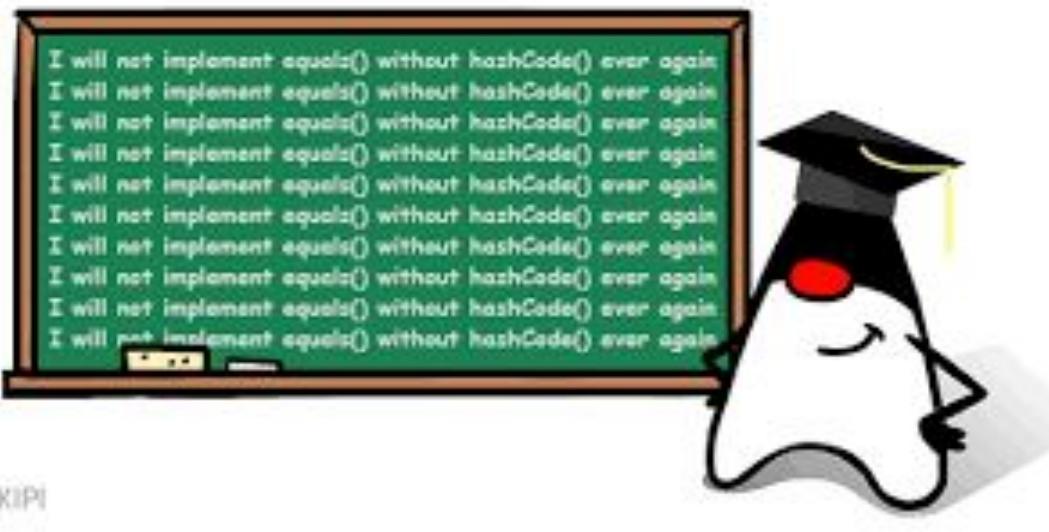
Se deben correr los elementos



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**

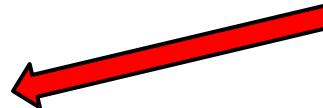


TAKIPI
2010

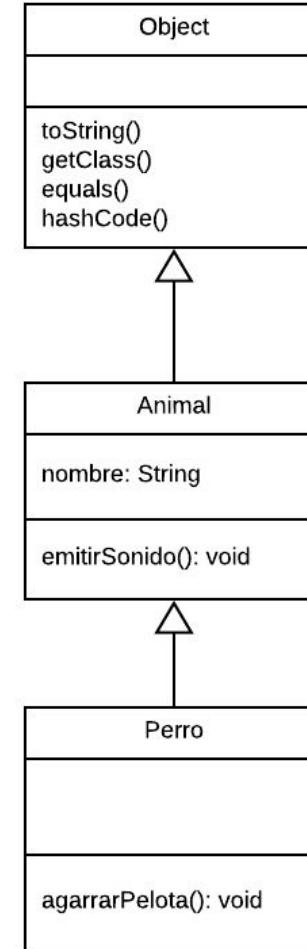
Método equals()

```
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**



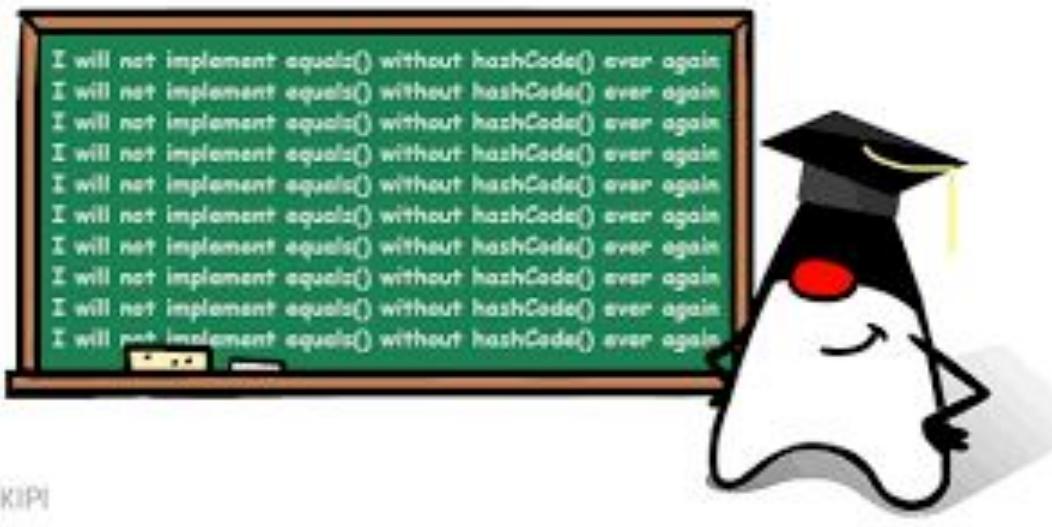
Método equals() (2)

- 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()

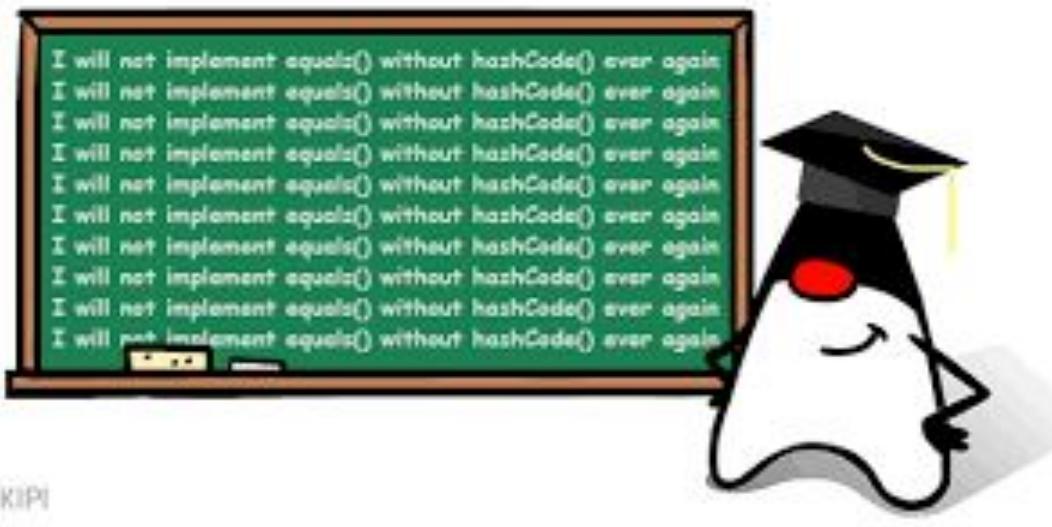
```
public int hashCode()
```

- 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>

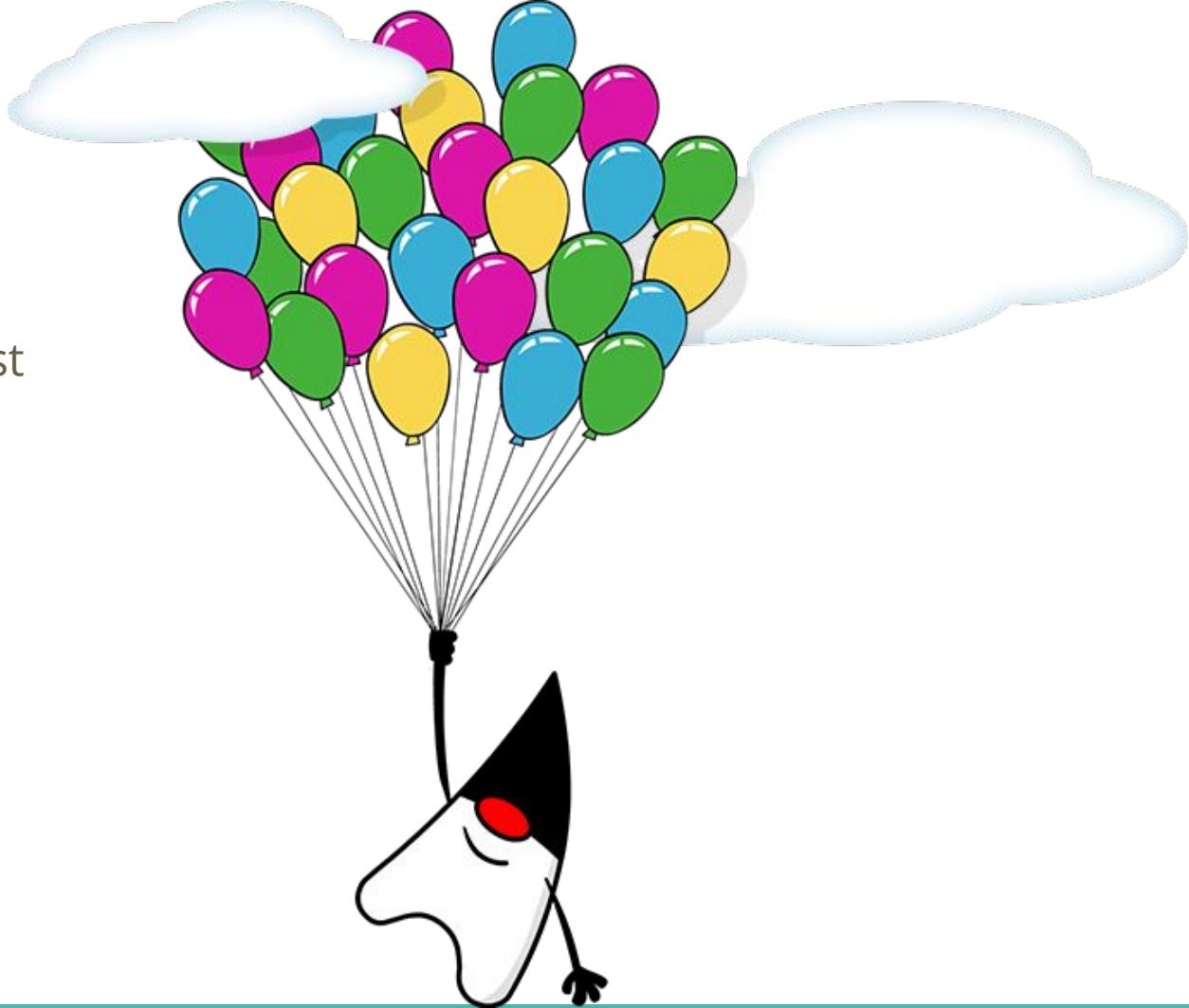
Clase 11: Collection - Listas

Parte II

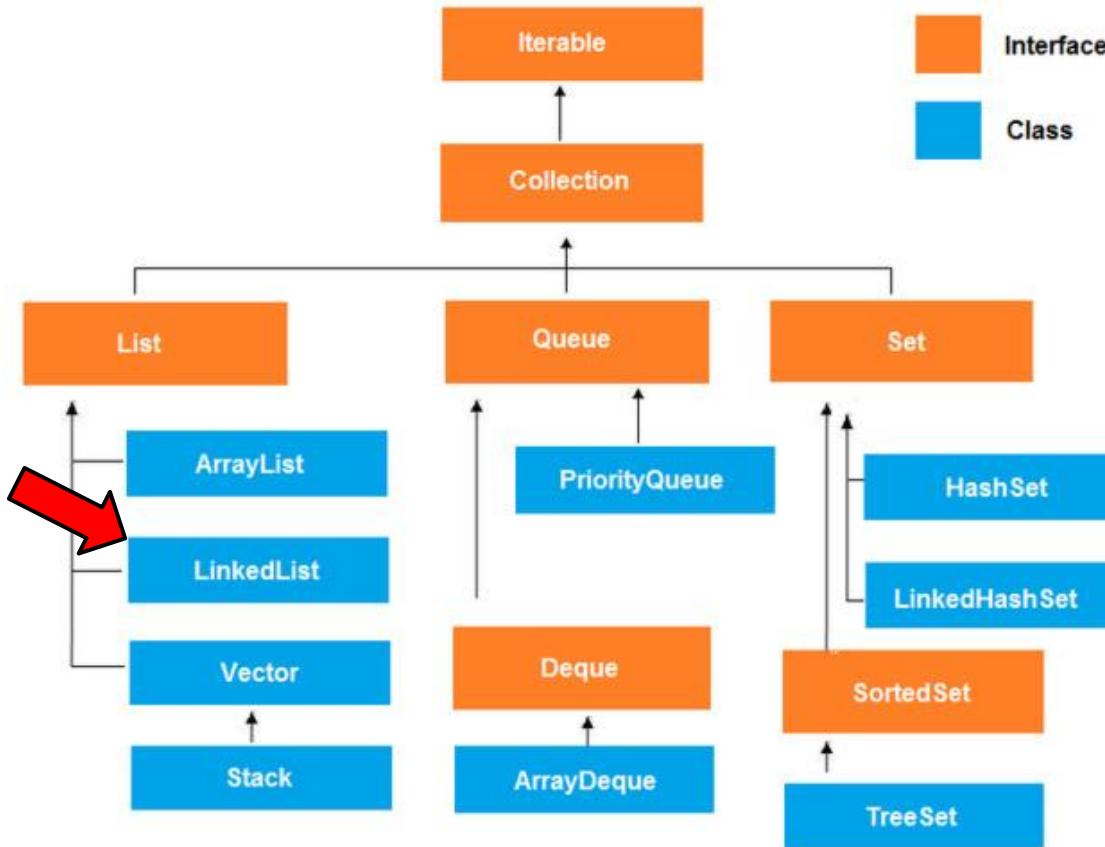
Programación III

Agenda

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

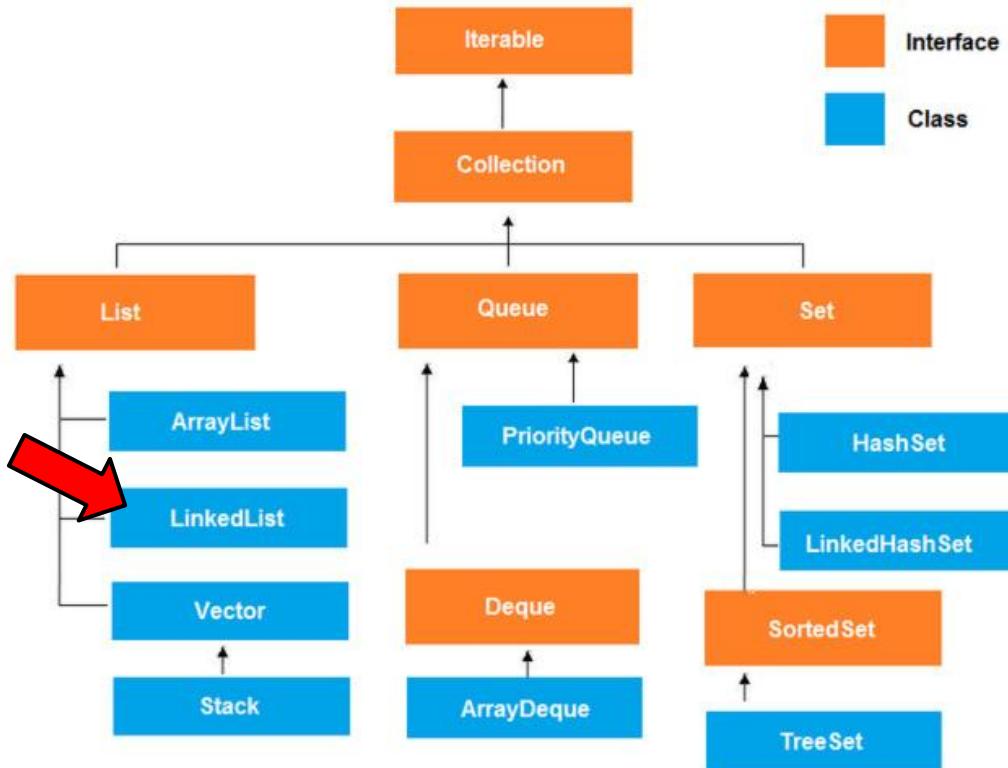


Collection API



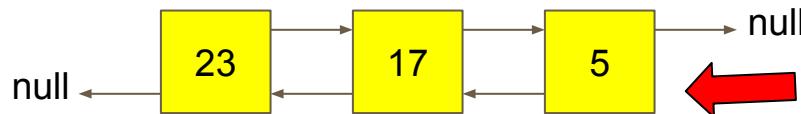
LinkedList

- 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.

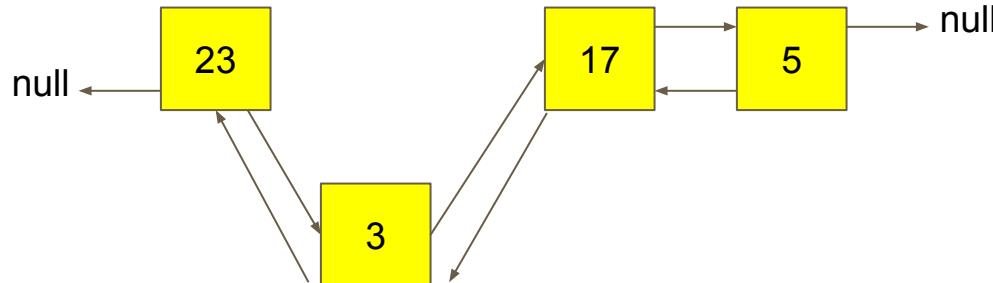


LinkedList - Insertar elemento

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

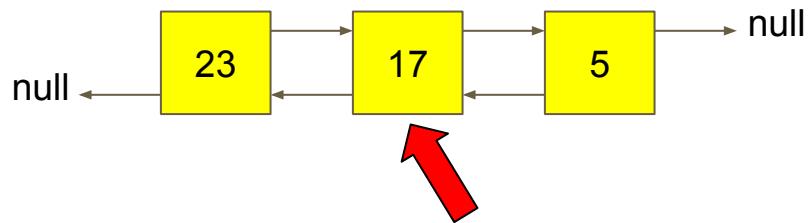


- 2) Agregar con índice → void add(int index, E element)



LinkedList - Leer elemento

- E get(int index)

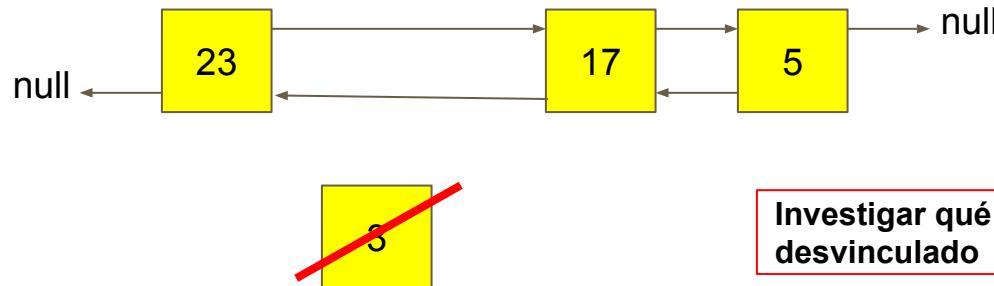


Investigar implementación
(link)



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

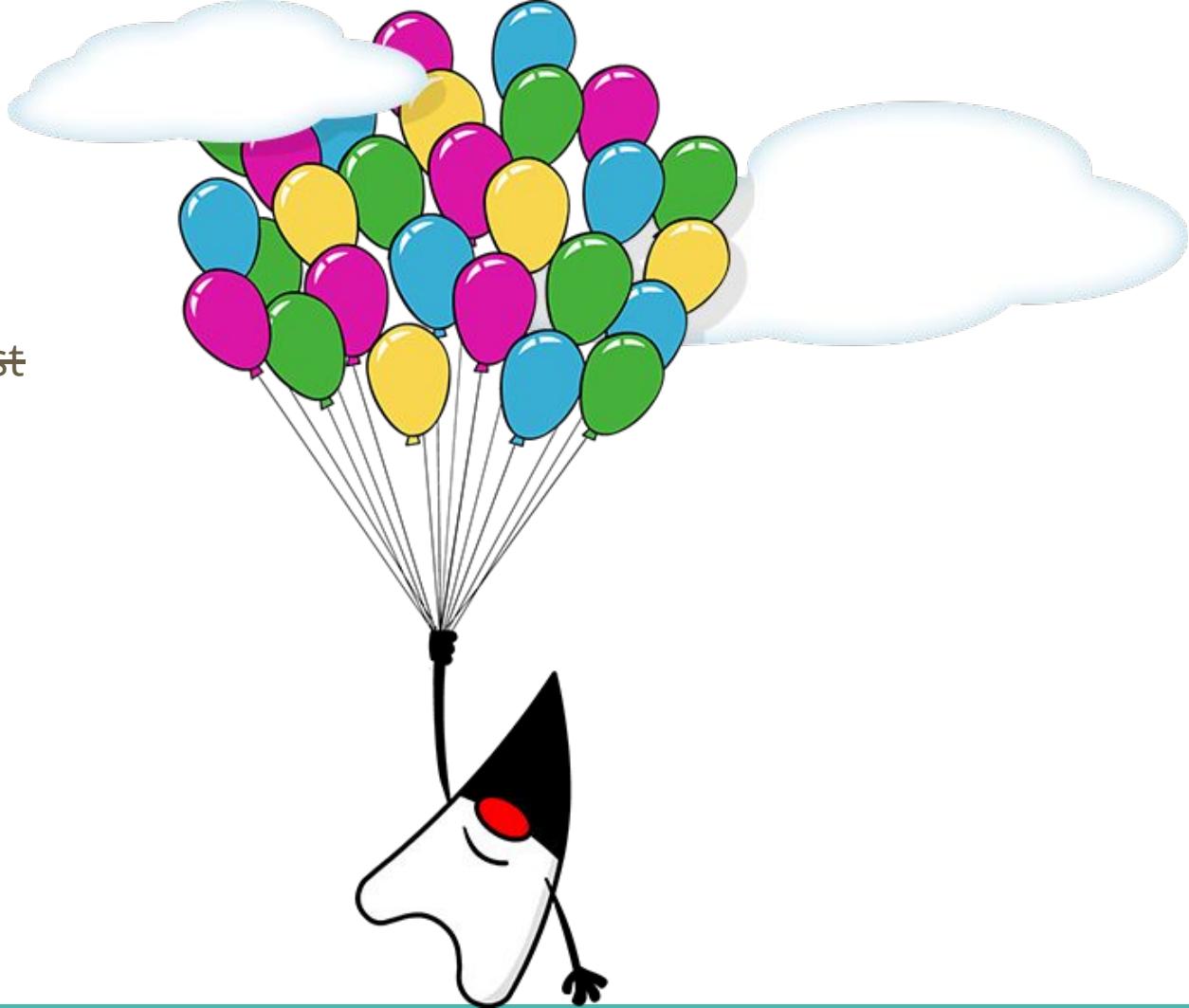


ArrayList vs LinkedList(2)

- 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).

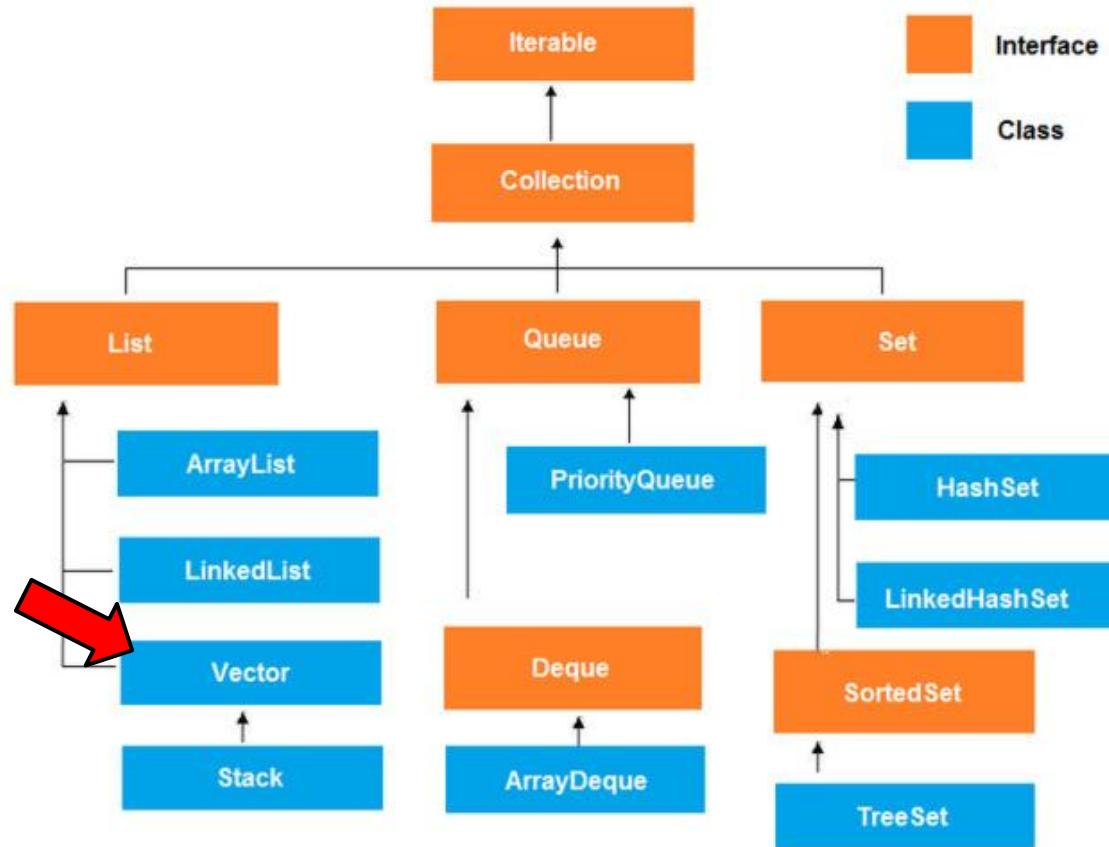
Agenda

- ~~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);
```

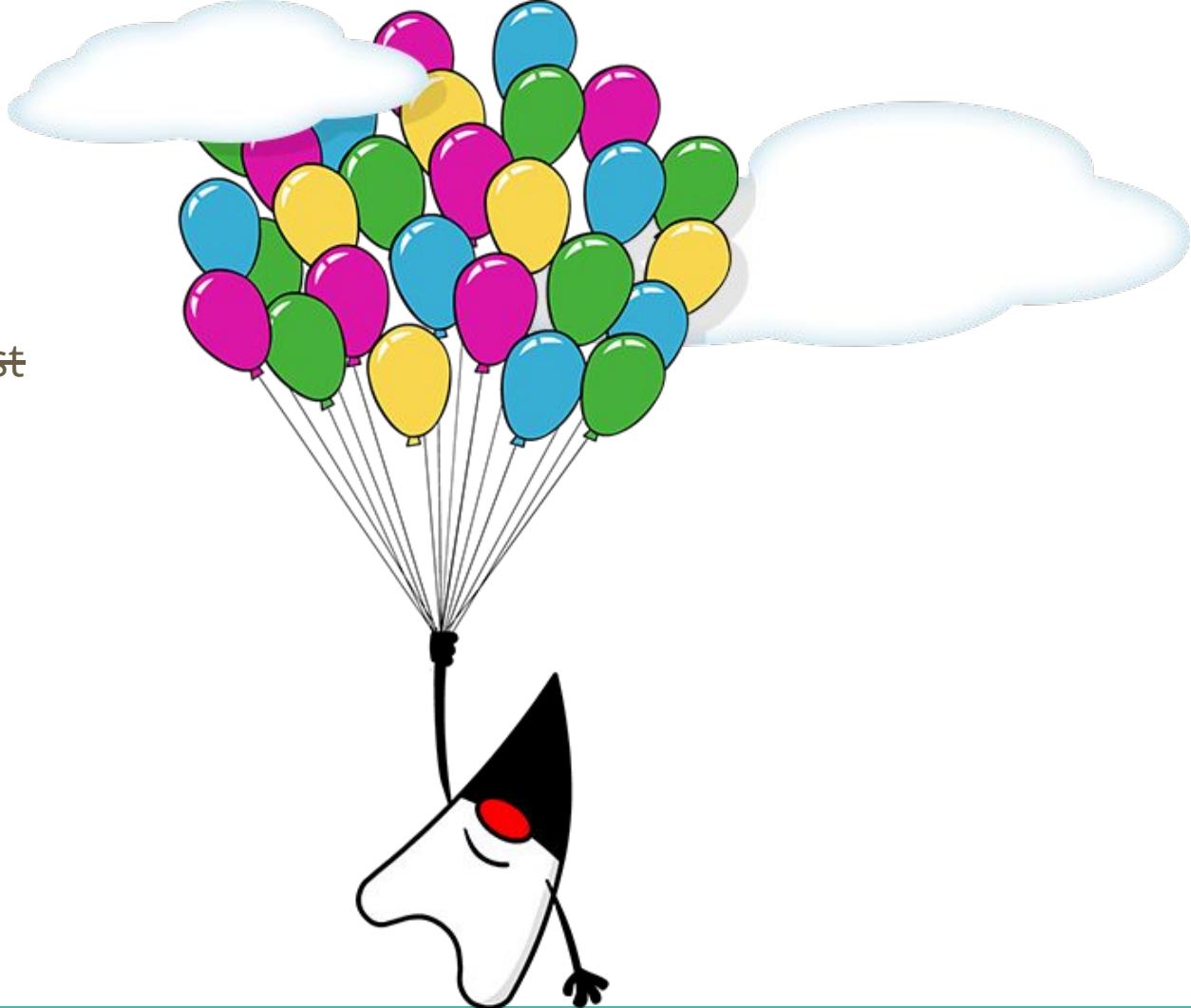
- Si se rebasa la dimensión inicial guardando 21 elementos, la dimensión del vector se duplica.

```
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

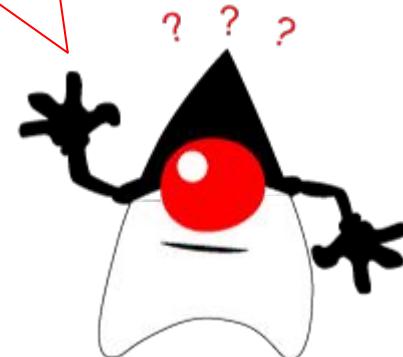
- ~~LinkedList (Repaso)~~
- ~~ArrayList vs. LinkedList~~
- ~~Vector~~
- **Vector vs. ArrayList**
- Stack



Vector vs. ArrayList

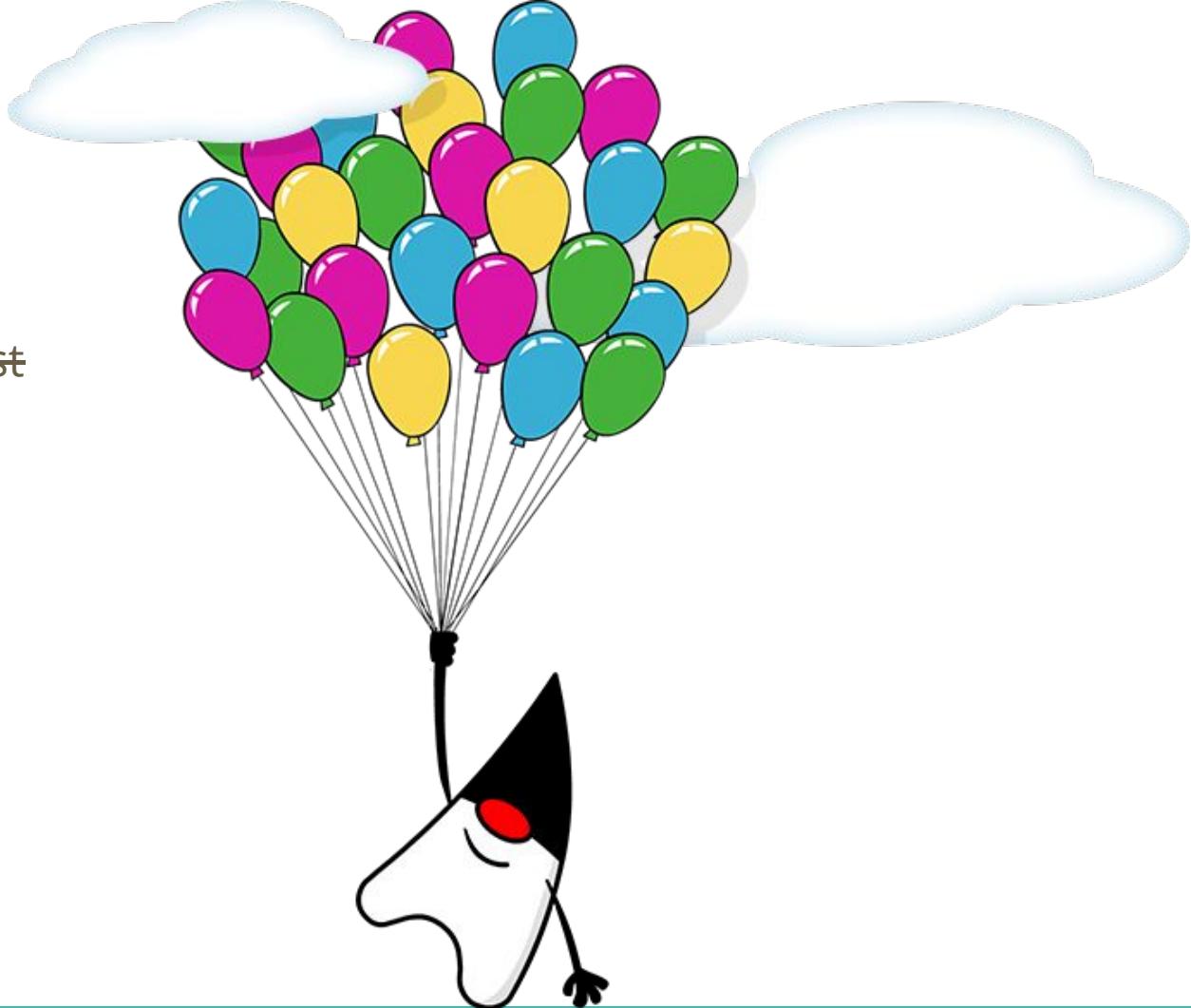
- Vector es sincronizada mientras que ArrayList no lo es. Si no trabajamos en un entorno multihilos utilizar ArrayList.
- La estructura de ambos está basada en array.
- Ambos pueden crecer y reducirse en forma dinámica, sin embargo la forma en la que se redimensionan es diferente.

Investigar
Collections.synchronizedList



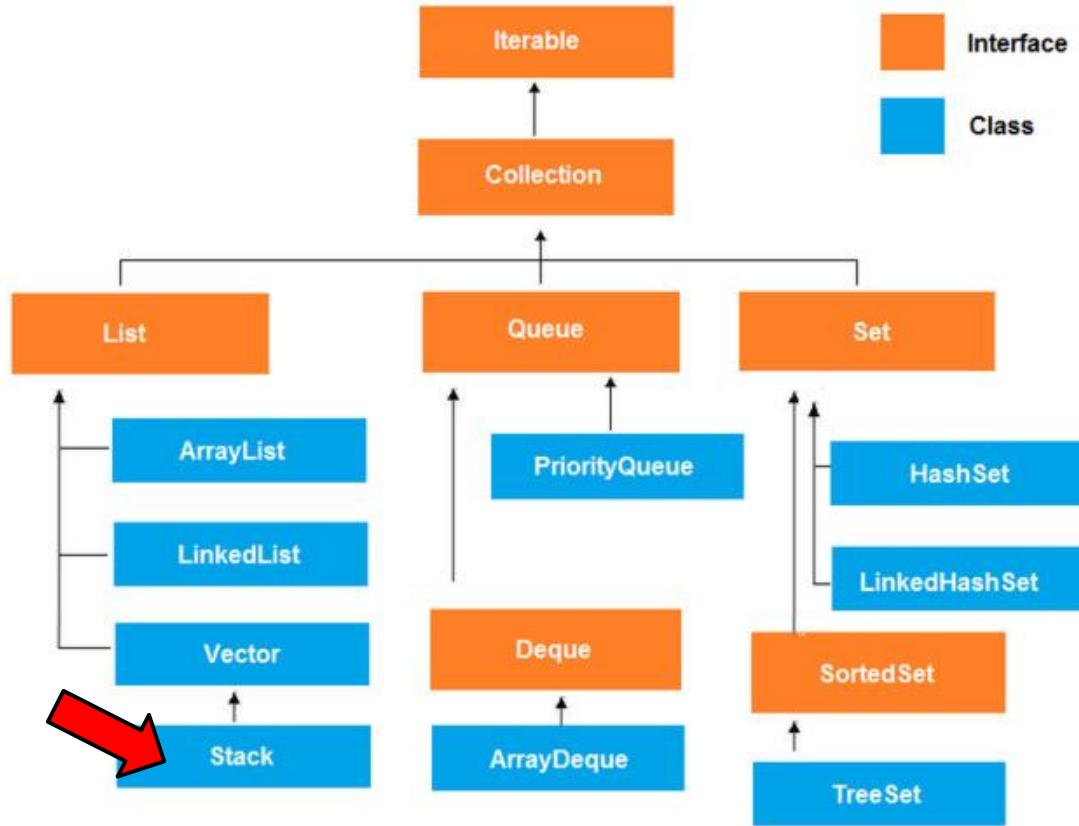
Agenda

- ~~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>

Métodos comunes de todos los objetos

equals hashCode comparable

Método equals, cuando NO:

- Cada instancia de la clase es propiamente única.
- No hay necesidad para la clase de proveer una igualdad lógica.
- Una superclase ya implementó este método y el mismo es apropiado para la clase.
- La clase es privada y estamos seguros que el método equals nunca será invocado.

Método equals, cuando SI:

- Cuando una clase tiene una igualdad lógica que va más allá de la identidad del objeto.
Y una superclase no haya implementado este método.

Que establece su contrato?

El método equals implementa una relación de equivalencia. Con las siguientes propiedades:

- Reflexivo: Para cada valor de X no nulo.
X.equals(X) es **true**.
- Simétrico: Para cada valor de X e Y no nulo.
X.equals(Y) es **true** si y sólo si Y.equals(X) es **true**.
- Transitivo: Para cada valor de X, Y, Z no nulo.
Si X.equals(Y) es **true**
Y Y.equals(Z) es **true**
Entonces X.equals(Z) debe ser **true**.
- Consistente: Para cada valor de X e Y. Múltiples invocaciones de X.equals(Y) debe retornar consistentemente **true** o consistentemente **false**. Siempre que no se modifique la información utilizada en ambos.
Para cada valor de X no nulo. X.equals(null) debe retornar **false**.

Receta para un método equals de calidad:

- 1) Usar el operador == para chequear si el argumento es una referencia a este objeto. Si lo es retorna true.
- 2) Usar el operador instanceof para chequear si el argumento posee el tipo correcto. Si no, retorna false.
- 3) Castear el argumento al tipo correcto. Como el casteo se realiza después de un instanceof está garantizado que será satisfactorio.
- 4) Por cada atributo "significante" en la clase chequear si ese atributo es igual al atributo correspondiente de este objeto. Si todos estos son satisfactorios, retorna true. Caso contrario false.

Para los atributos primitivos cuyo tipo no es float ni double usar el operador == para comparar.

Para objetos, llamar al método equals recursivamente.

Para atributos float usar el método estático compare de la clase Float. Float.compare(float1, float2)

Para atributos double usar el método estático compare de la clase Double. Double.compare(double1, double2)

Ejemplo método equals

```
public class Telephone {  
    private short areaCode;  
    private short prefix;  
    private short lineNum;  
    ...  
    @Override  
    public boolean equals(Object o) {  
        if (o == this) return true;  
        if (!(o instanceof Telephone)) return false;  
        Telephone tp = (Telephone)o;  
        return tp.lineNum == lineNum && tp.prefix == prefix && tp.areaCode ==  
areaCode;  
    }  
    ...
```

Método hashCode:

**Siempre se debe hacer Override de hashCode cuando se hace
Override de equals.**

Que establece su contrato?

- Cuando el método hashCode es invocado en un objeto repetidas veces debe retornar el mismo valor consistentemente.
- Si dos objetos son iguales de acuerdo al método equals, entonces hashCode debe retornar el mismo valor para ambos.
- Si dos objetos son distintos de acuerdo al método equals, no es necesario que hashCode produzca un valor distinto. Sin embargo esto puede mejorar la performance de una hash table.

Cómo NO escribir un método hashCode

```
@Override  
public int hashCode() {  
    return 42;  
}
```

Receta para un buen método hashCode:

1. Declarar una variable entera llamada resultado e inicializarla con el valor de hash code del primer atributo significante en nuestro objeto.
2. Para cada atributo remanente en nuestro objeto hacer lo siguiente:
 - a. Si el atributo es de tipo primitivo computar Type.hashCode(atributo). Donde Type es el Tipo correspondiente a la primitiva.
 - b. Si el atributo es un objeto y esa clase hace uso de equals de manera recursiva para sus atributos. Usar hashCode recursivamente en el atributo. Si el valor del atributo es null usar 0.
 - c. Si el atributo es un array, tratarlo como si cada elemento fuera un atributo separado. Esto significa calcular un hashCode para cada elemento en el array y combinar los valores. Si el array no tiene elementos significantes usar una constante diferente de 0. Si TODOS los elementos son importantes usar Arrays.hashCode.
3. Combinar el resultado de esta forma: $\text{resultado} = 31 * \text{resultado} + \text{atributo};$

Ejemplo método hashCode

```
public class Telephone {  
    private short areaCode;  
    private short prefix;  
    private short lineNum;  
    ...  
    public int hashCode() {  
        int result = Short.hashCode(areaCode);  
        result = 31 * result + Short.hashCode(prefix);  
        result = 31 * result + Short.hashCode(lineNum);  
        return result;  
    }  
}
```

Interfaz Comparable

```
public interface Comparable<T> {  
    int compareTo(T t);  
}
```

Al implementar esta interfaz en una clase estamos indicando que las instancias de dicha clase poseen un **orden natural**.

El contrato de este método especifica que debe devolver lo siguiente:

- Entero negativo si el objeto es menor al especificado por parámetro.
- Cero si el objeto es igual al especificado por parámetro.
- Entero positivo si el objeto es mayor al especificado por parámetro.

Ejemplo método compareTo

```
public class Telephone {  
    private short areaCode;  
    private short prefix;  
    private short lineNum;  
    ...  
    public int compareTo(Telephone tp) {  
        int result = Short.compare(areaCode, tp.areaCode);  
        if (result == 0) {  
            result = Short.compare(prefix, tp.prefix);  
            if (result == 0) result = Short.compare(lineNum, tp.lineNum);  
        }  
        return result;  
    }  
    ...
```

Clase 13: Map

— Programación y Laboratorio III —

Agenda

- **Interfaz Map**
- **HashMap**
- Principio de Hashing
- equals() & hashCode()
- **HashMap vs. Hashtable**
- **LinkedHashMap**
- **TreeMap**

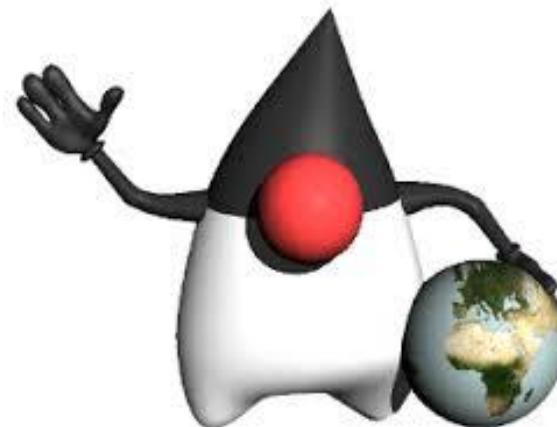


Interfaz Map

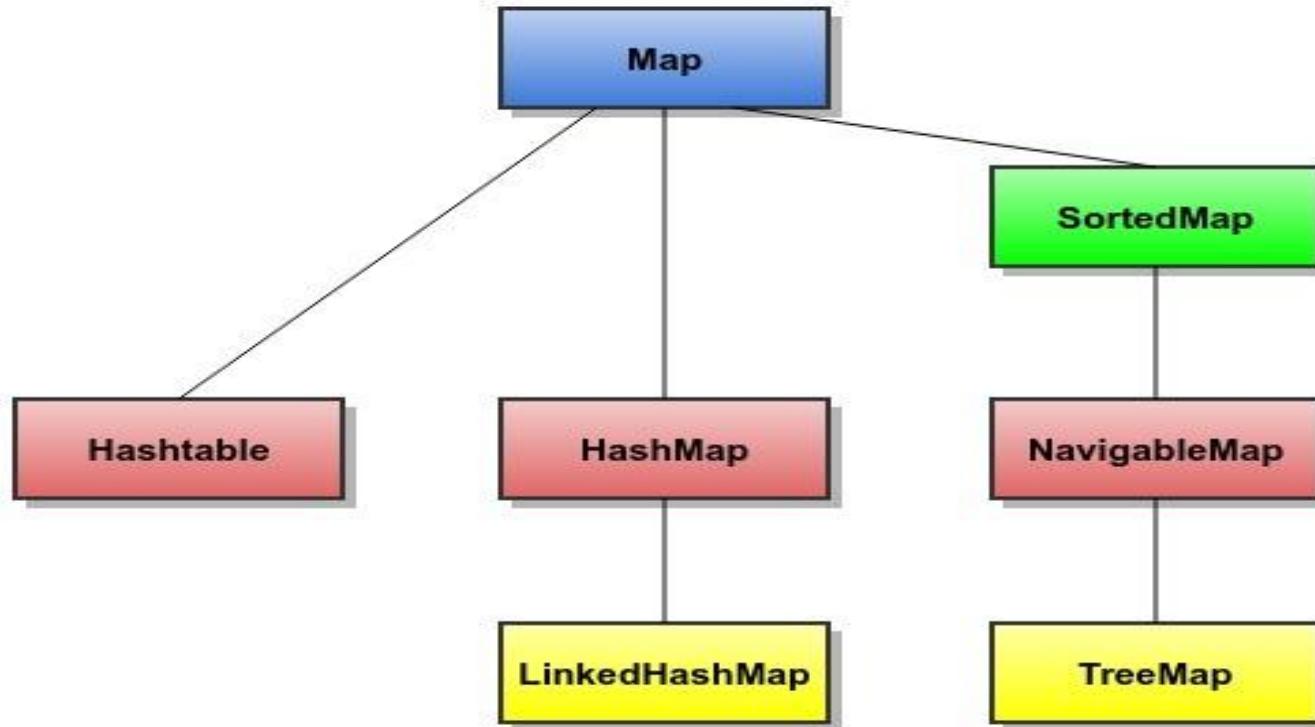
- Nos permite representar una estructura de datos para almacenar pares “clave-valor”.
- Para una clave sólo tenemos un valor.
- Map tienen implementada por debajo toda la teoría de las estructuras de datos de árboles, por lo tanto permiten añadir, eliminar y modificar elementos de forma transparente.
- La clave funciona como un identificador único y no se admiten claves duplicadas.

Map - Operaciones

- boolean containsKey(Object key)
- boolean containsValue(Object var1);
- V get(Object var1);
- V put(K var1, V var2);
- V remove(Object var1);
- int size();
- boolean isEmpty();



Interfaz Map



Agenda

~~Interfaz Map~~

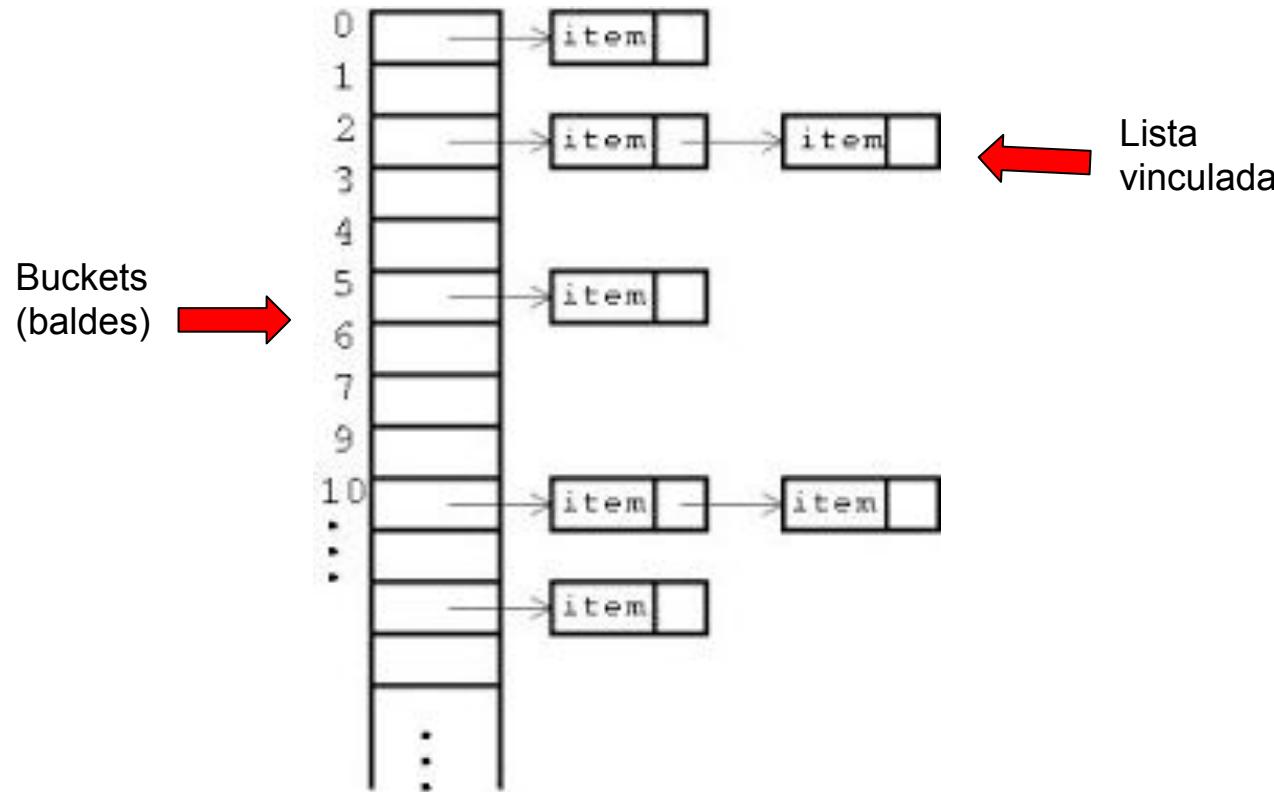
- **HashMap**
 - Principio de Hashing
 - equals() & hashCode()
 - HashMap vs. Hashtable
 - LinkedHashMap
 - TreeMap



HashMap

- Un HashMap es una colección de objetos, como los arrays, pero estos no tienen orden.
- Cada objeto se identifica mediante algún identificador y conviene que sea inmutable (`final`), de modo que no cambie en tiempo de ejecución.
- El nombre Hash hace referencia a una técnica de organización de archivos llamada hashing o “dispersión” en el cual se almacenan registros en una dirección del archivo que es generada por una función que se aplica a la clave del mismo.
- Los elementos que se insertan en un HashMap no tendrán un orden específico.
- Permite una clave null.

HashMap - Estructura

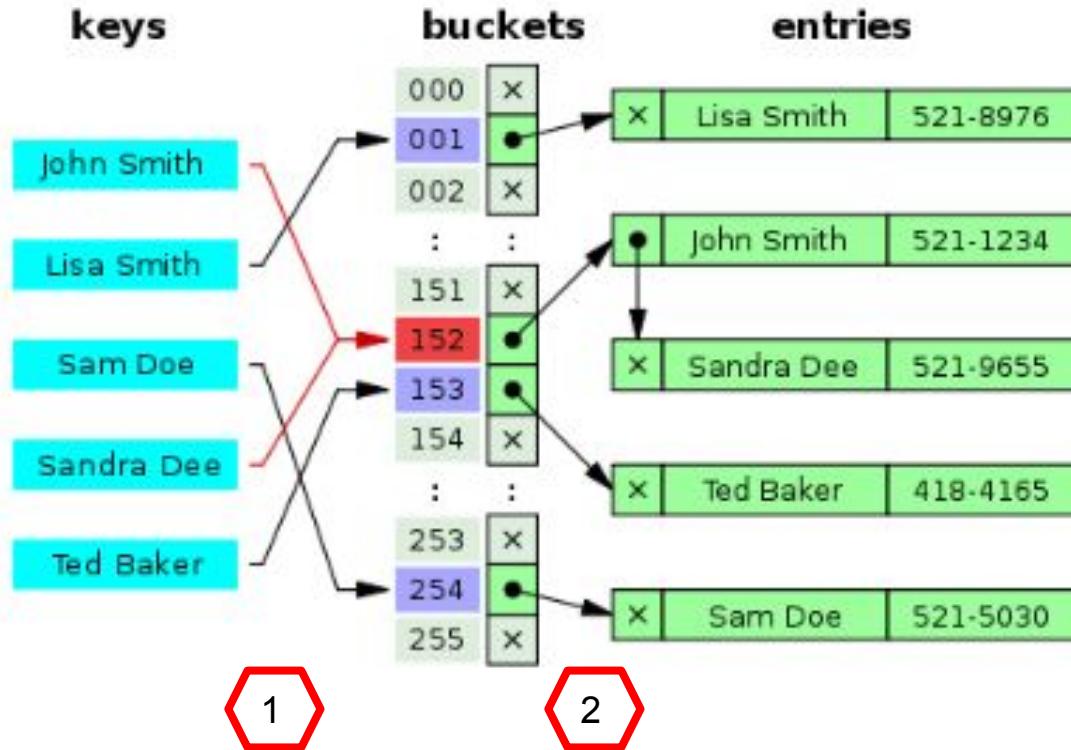


Agenda

- ~~Interfaz Map~~
- ~~HashMap~~
- **Principio de Hashing**
- equals() & hashCode()
- HashMap vs. Hashtable
- LinkedHashMap
- TreeMap



Principio de Hashing



Principio de Hashing (2)

- 1 Se utiliza el método hashCode() para encontrar el bucket correspondiente.
- 2 Se utiliza el método equals() para buscar el valor correspondiente a la clave dada.
 - Dos objetos idénticos tendrán el mismo identificador returnedo por el método hashCode().

Agenda

- ~~Interfaz Map~~
- ~~HashMap~~
- ~~Principio de Hashing~~
- **equals() & hashCode()**
- HashMap vs. Hashtable
- LinkedHashMap
- TreeMap



equals() & hashCode()

NO OLVIDAR:



“Si dos objetos son iguales usando equals(), entonces la invocación a hashCode() de ambos objetos debe retornar el mismo valor.”

equals() & hashCode() (2)

1) Objetos que son iguales pero retornan diferentes hashCode.

Cada uno de estos baldes tiene asignado un número que lo identifica. Cuando agregamos un valor al HashMap, almacena el dato en uno de esos baldes. El balde que se usa depende del hashCode que devuelva el objeto a ser almacenado. Por ejemplo, si el método hashCode() del objeto retorna 49, entonces se almacena en el balde 49 dentro del HashMap.

Más tarde, cuando verifiquemos si la colección contiene al elemento invocando el método contains(elemento), el HashMap primero obtiene el hashCode de ese "elemento". Luego buscará el balde que corresponde a ese hashCode. Si el balde está vacío, significa que el HashMap no contiene al elemento y devuelve false.

equals() & hashCode() (3)

2) Objetos que no son iguales pero retornan el mismo hashCode.

Se utiliza este mecanismo de "baldes" por un tema de eficiencia. Si todos los objetos que se agregan a un HashMap se almacenaran en una única lista grande, entonces tendríamos que comparar la entrada con todos los objetos de la lista para determinar si un elemento en particular está contenido en el Map. Como se usan baldes, sólo se comparan los elementos del balde específico, y en general cada balde sólo almacena una pequeña cantidad de elementos en el HashMap.

Agenda

- ~~Interfaz Map~~
- ~~HashMap~~
- ~~Principio de Hashing~~
- ~~equals() & hashCode()~~
- **HashMap vs. Hashtable**
- **LinkedHashMap**
- **TreeMap**



HashMap vs. Hashtable

- La principal diferencia entre uno y otro es la sincronización interna del objeto. Para aplicaciones multihilos es preferible elegir Hashtable sobre HashMap, que no tiene sincronización.
- HashMap es mejor en cuanto a performance.
- Hashtable no admite valores nulos en ninguna de sus partes, mientras que HashMap sí.

Agenda

- ~~Interfaz Map~~
- ~~HashMap~~
- ~~Principio de Hashing~~
- ~~equals() & hashCode()~~
- ~~HashMap vs. Hashtable~~
- **LinkedHashMap**
- **TreeMap**



LinkedHashMap

- Similar a HashMap pero con la diferencia que mantiene una lista doblemente vinculada, además del array de baldes.
- La lista doblemente vinculada define el orden de iteración de los elementos

Agenda

- ~~Interfaz Map~~
- ~~HashMap~~
- ~~Principio de Hashing~~
- ~~equals() & hashCode()~~
- ~~HashMap vs. Hashtable~~
- ~~LinkedHashMap~~
- **TreeMap**



TreeMap

- Implementado mediante un árbol binario y permite tener un mapa ordenado.
- Cuando iteramos un objeto TreeMap los objetos son extraídos en forma ascendente según sus keys.
- TreeMap no sabe cómo ordenar la colección cuando se utiliza una key creada por el programador, sólo lo hace si trabajamos con objetos tipo String, Integer, etc.



Bibliografía oficial

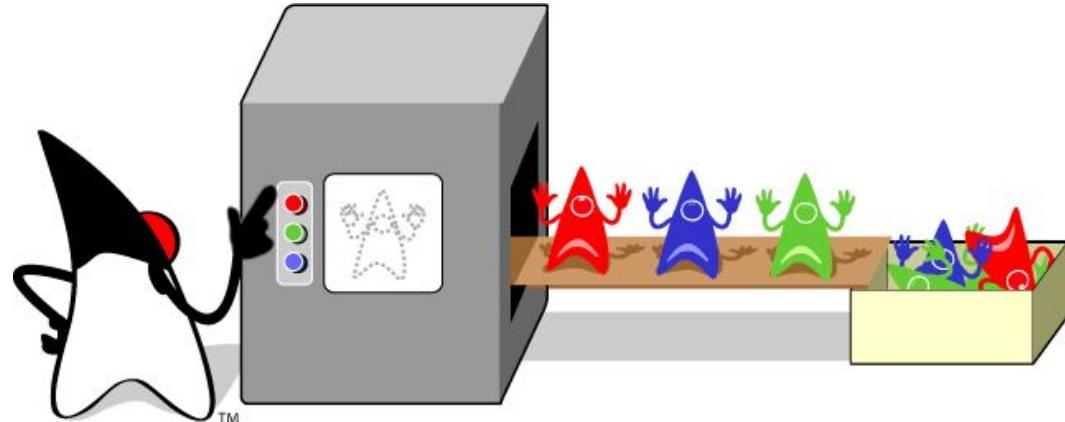
- <https://docs.oracle.com/javase/8/docs/api/java/util/Map.html>
- <https://docs.oracle.com/javase/8/docs/api/java/util/Hashtable.html>
-

Clase 14: Collection - Set

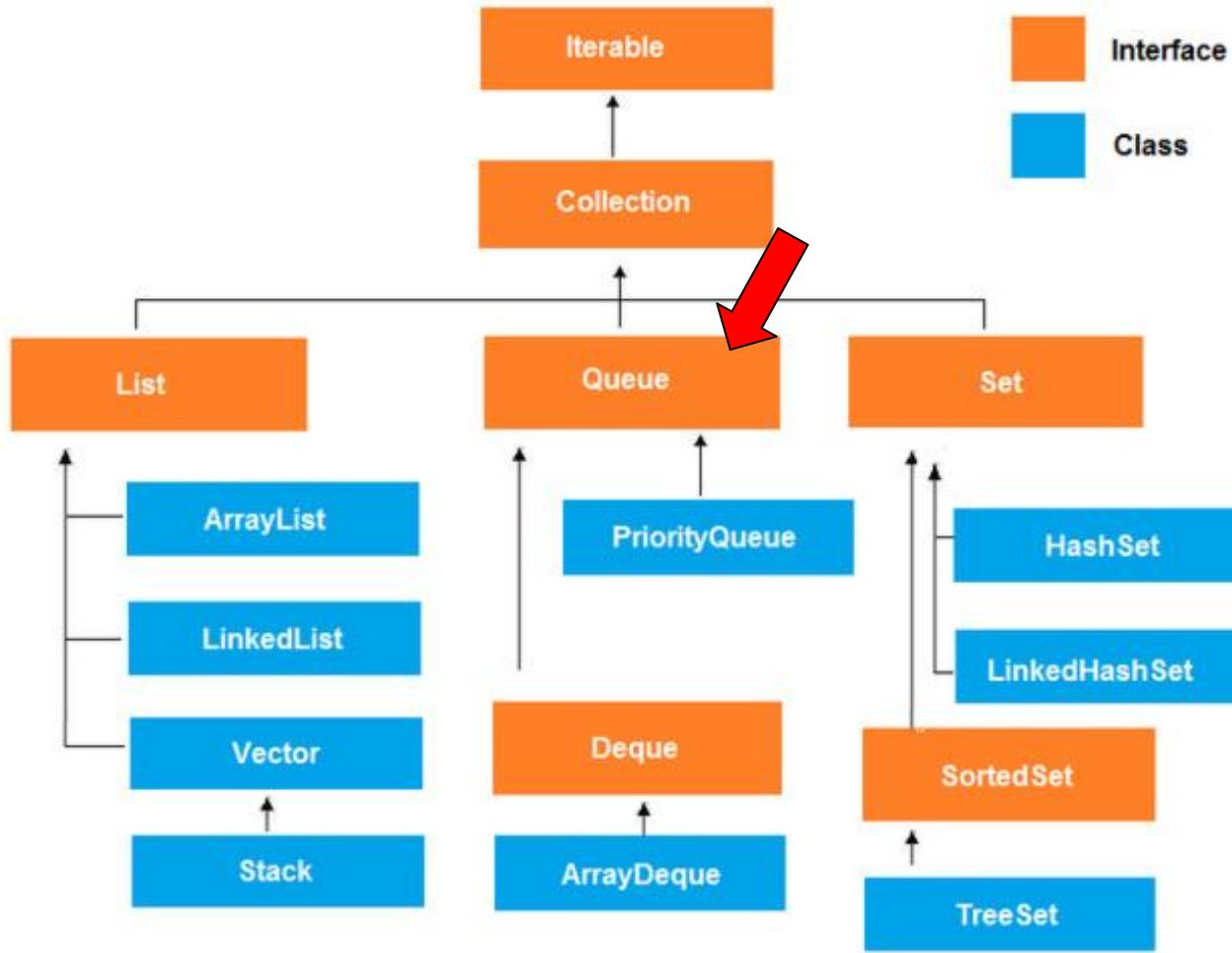
— Programación y Laboratorio III —

Agenda

- Queue
- Set
- HashSet
- LinkedHashSet
- TreeSet
- Diagrama de decisiones para uso de colecciones

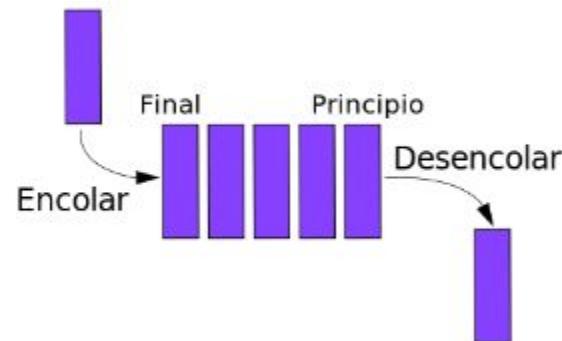


Collection Queue



Queue

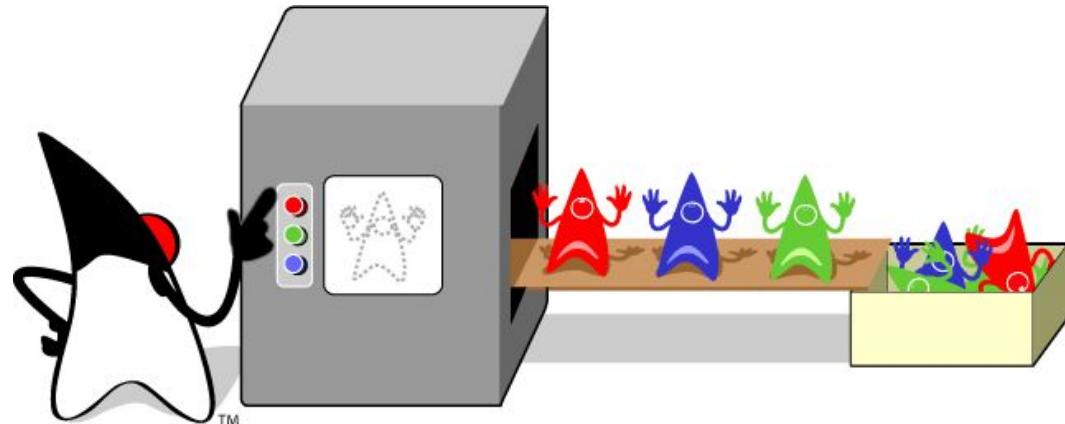
- Representa al tipo Cola, que es una lista en la que sus elementos se introducen únicamente por un extremo (fin de la cola) y se remueven por el extremo contrario (principio de la cola).



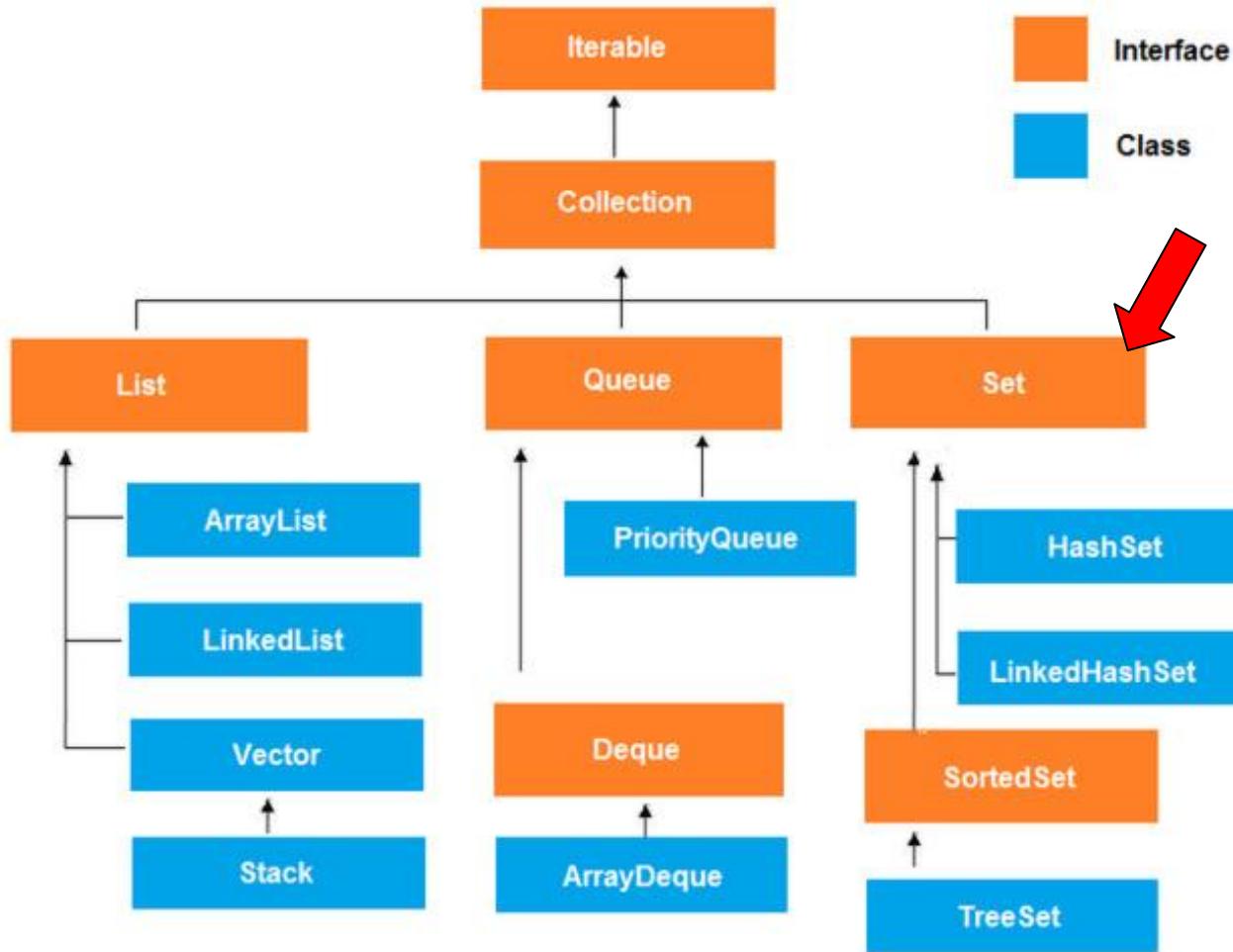
Agenda

~~Queue~~

- **Set**
 - HashSet
 - LinkedHashSet
 - TreeSet
 - Diagrama de decisiones para uso de colecciones



Collection Set



Interfaz Set

- Modela los conjuntos de la matemática y sus propiedades.
- Set hereda los métodos de Collection y agrega sus propias restricciones para prohibir el duplicado de elementos.

Si tenemos un objeto que tienen las mismas características (equals) y el mismo hashCode que los objetos que ya se encuentran en el Set → **No se agrega a la colección**

Interfaz Set - Algunas operaciones

- int size();
- boolean isEmpty();
- boolean contains(Object var1);
- Iterator<E> iterator();
- boolean add(E var1);
- boolean remove(Object var1);
- **boolean addAll(Collection<? extends E> var1);**
- **boolean retainAll(Collection<?> var1);**
- **boolean removeAll(Collection<?> var1);**

Interfaz Set - Operaciones

- **boolean addAll(Collection<? extends E> var1);**

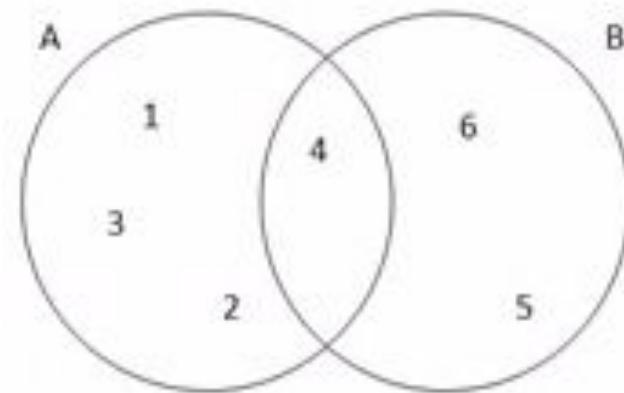
$$A \cup B = \{1,2,3,4,5,6\}$$

- **boolean retainAll(Collection<?> var1);**

$$A \cap B = \{4\}$$

- **boolean removeAll(Collection<?> var1);**

$$A - B = \{1,2,3\}$$

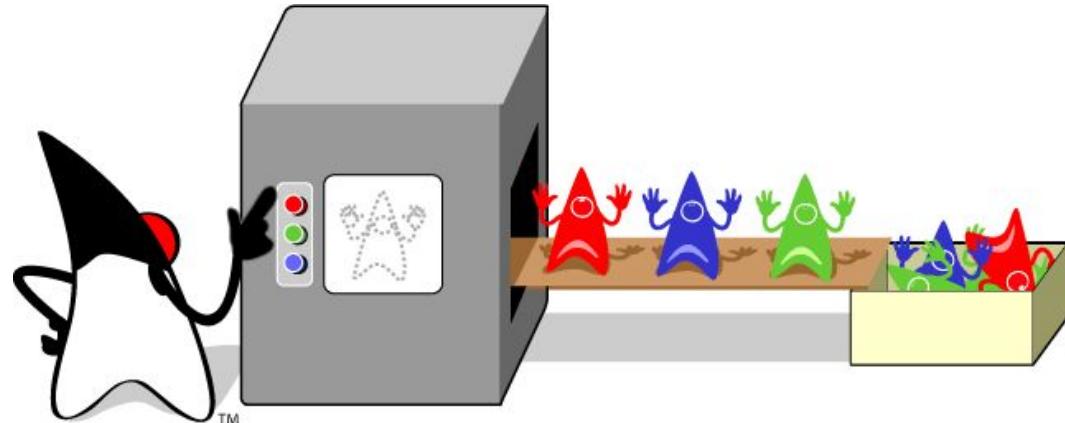


Agenda

~~— Queue~~

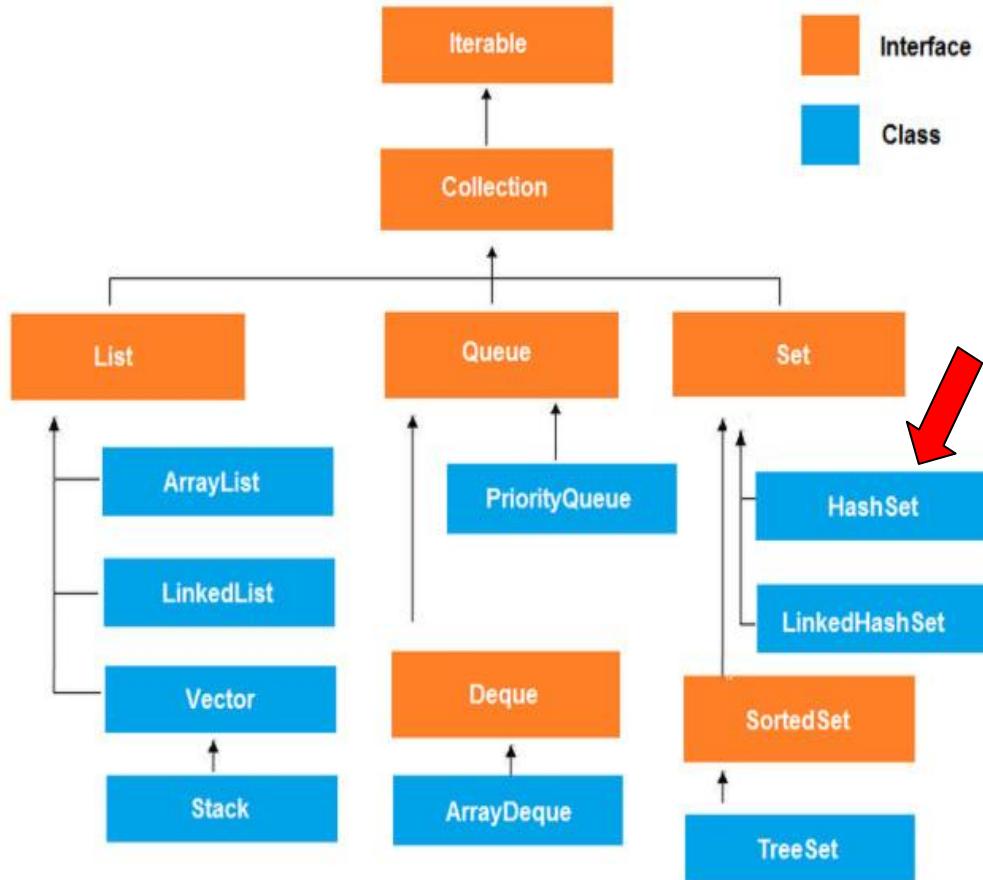
~~— Set~~

- **HashSet**
- **LinkedHashSet**
- **TreeSet**
- Diagrama de decisiones para uso de colecciones



HashSet

- Esta implementación de Set almacena los elementos en una **tabla hash**.
- Esta implementación proporciona tiempos constantes en las operaciones básicas siempre y cuando la función hash disperse de forma correcta los elementos dentro de la tabla hash.



HashSet (2)

- Esta clase delega casi todas sus funcionalidades a un **HashMap<T, Object>**, donde el objeto que se inserta en el HashSet será la clave del mapa interno, y se registrará con el valor de un objeto por defecto dentro del mapa.
- Toda la lógica de verificación que el elemento no esté duplicado la maneja el mapa interno utilizado en la instancia del set → Las claves en un HashMap deben ser únicas.

HashSet (3)

- 1) Creación de HashSet:

```
HashSet<String> hashSet = new HashSet<String>();
```

- 2) Al llamar al constructor de HashSet, se ejecuta lo siguiente:

```
public HashSet() {  
    map = new HashMap<>();  
}
```

“map” es una variable de instancia de
HashSet declarada como:
HashMap<E, Object> map;

HashSet (4)

3) Una vez creado el HashSet, agregamos un conjunto de nombres:

```
hashSet.add("Pedro");  
hashSet.add("Pepe");  
hashSet.add("Luis");
```

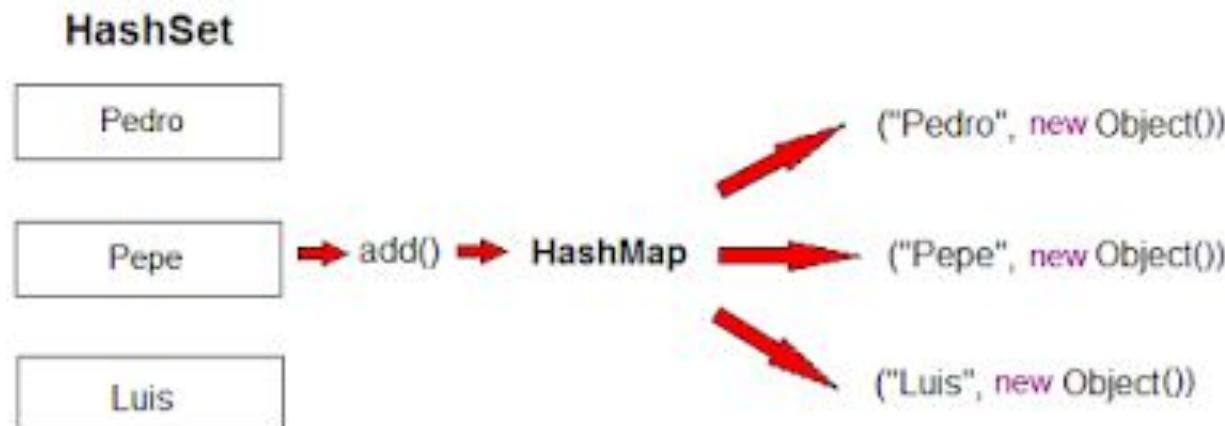
4) Implementación del método add(E e):

```
public boolean add(E e) {  
    return map.put(e, PRESENT)==null;  
}
```

```
Object PRESENT = new Object();
```

HashSet (4)

- Si ese objeto ya está agregado devuelve el valor anterior de esa key pero si el objeto no está y se agrega al HashMap devuelve null, entonces lo que hace el método add() es evaluar que devuelve y así se sabe si agrega o no el objeto.



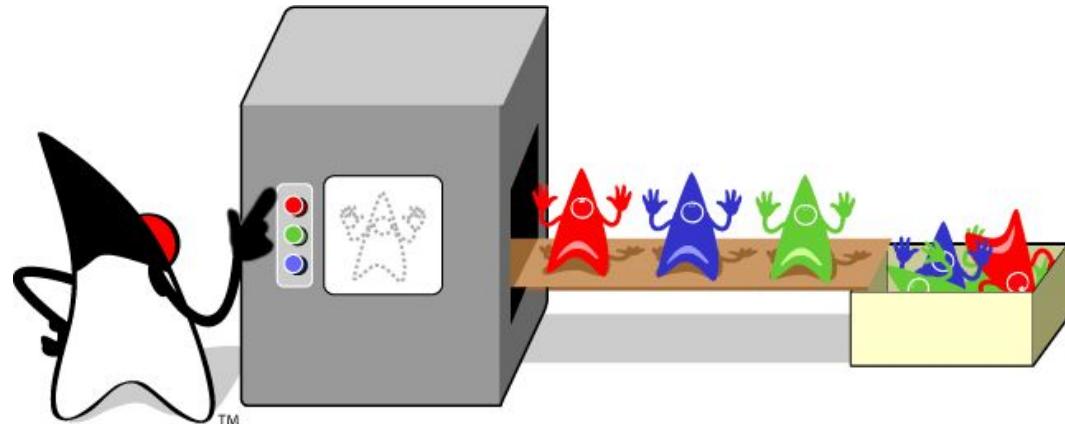
Agenda

~~Queue~~

~~Set~~

~~HashSet~~

- **LinkedHashSet**
- TreeSet
- Diagrama de decisiones para uso de colecciones

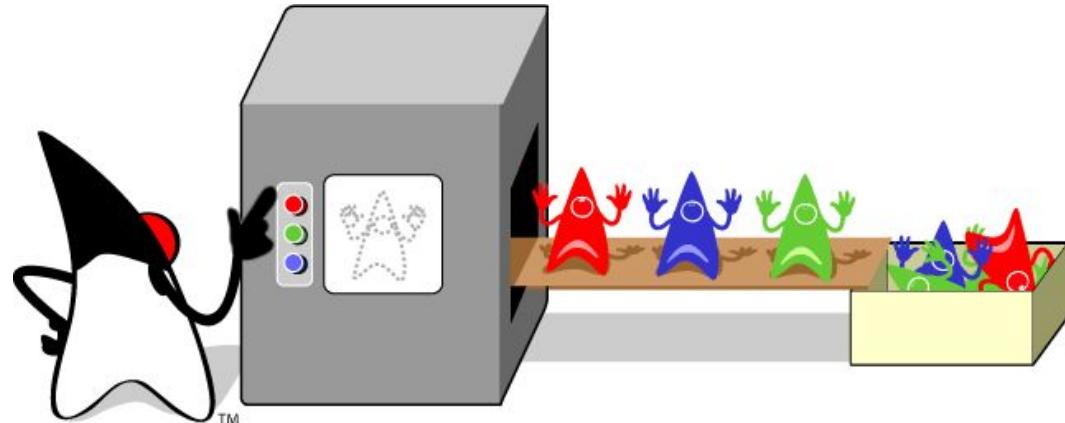


LinkedHashSet

- Es similar a HashSet pero la tabla de dispersión es doblemente enlazada.
- Los elementos que se inserten tendrán enlaces entre ellos. Por lo tanto, las operaciones básicas seguirán teniendo coste constante, con la carga adicional que supone tener que gestionar los enlaces. Sin embargo habrá una mejora en la iteración, ya que al establecerse enlaces entre los elementos no tendremos que recorrer todas las entradas de la tabla, el coste sólo estará en función del número de elementos insertados.
- En este caso, al haber enlaces entre los elementos, estos enlaces definirán el **orden** en el que se insertaron en el conjunto, por lo que el orden de iteración será el mismo orden en el que se insertaron.

Agenda

- ~~Queue~~
- ~~Set~~
- ~~HashSet~~
- ~~LinkedHashSet~~
- **TreeSet**
- Diagrama de decisiones para uso de colecciones



TreeSet

- Esta implementación almacena los elementos ordenándolos en función de sus valores.
- Es bastante más lento que HashSet.
- Los elementos almacenados deben implementar la **interfaz Comparable**.
- Garantiza, siempre, un rendimiento de $\log(N)$ en las operaciones básicas, debido a la estructura de árbol empleada para almacenar los elementos.

Agenda

- Queue
- Set
- HashSet
- LinkedHashSet
- TreeSet
- Diagrama de decisiones para uso de colecciones

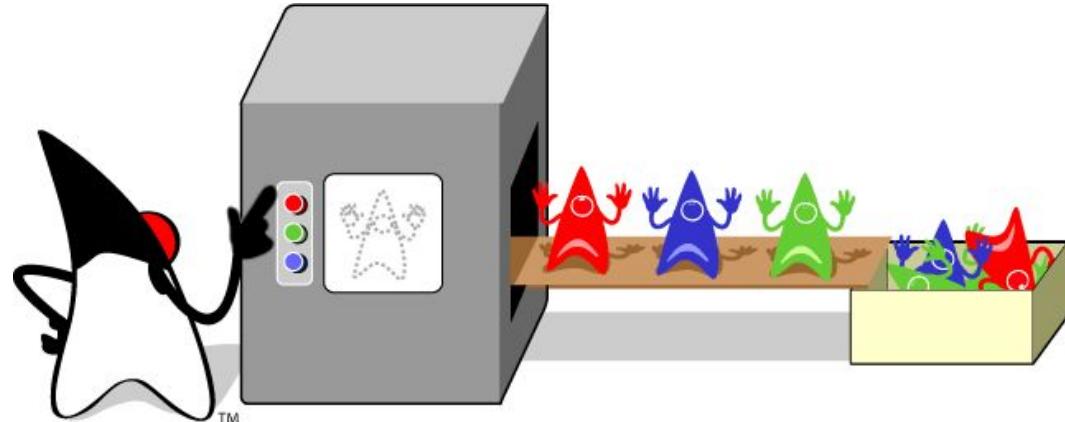
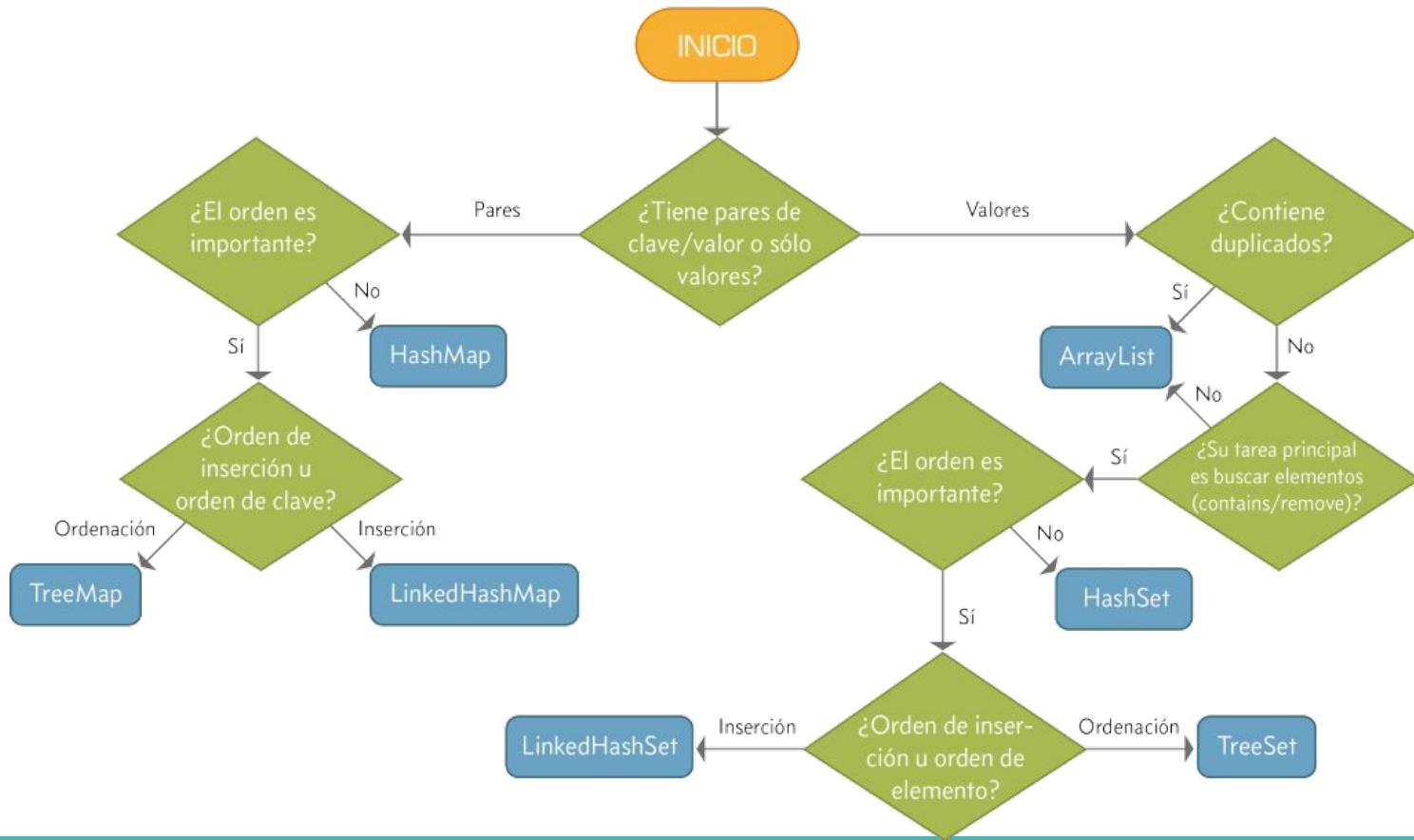


Diagrama de decisión para uso de colecciones Java



Bibliografía oficial

- <https://docs.oracle.com/javase/tutorial/collections/implementations/set.html>
- <https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html>
- <https://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html>
- <https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashSet.html>
- <https://docs.oracle.com/javase/tutorial/collections/implementations/summary.html>

Clase 15: Genéricos

— Programación & Laboratorio III —

Agenda

- Ejemplo
- Genericidad
- Declaración de tipos genéricos
- Declaración de una clase genérica
- Tipos genéricos - Convención
- Ejemplos



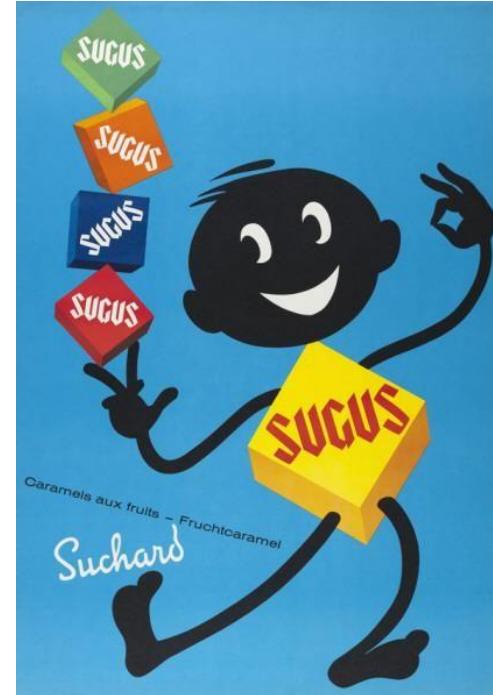
Ejemplo

```
public class Caja {  
  
    private List<Object> elementos = new ArrayList<>();  
    private int tope;  
  
    private Caja(int tope) {  
        this.tope = tope;  
    }  
  
    public boolean agregarElemento(Object o) {  
        if (tope < elementos.size()) {  
            elementos.add(o);  
            return true;  
        }  
        return false;  
    }  
}
```



Ejemplo (2)

```
public class Caramelo {  
  
    private String marca;  
    private String sabor;  
  
    private Caramelo(String marca, String sabor) {  
        this.marca = marca;  
        This.sabor = sabor;  
    }  
  
    @Override  
    public String toString() {  
        return "Caramelo= [marca= " + marca +  
               ", sabor= " + sabor + "]");  
    }  
}
```



Ejemplo (3)

```
public class Main {  
  
    public static void main(String[] args) {  
        Caja miCaja = new Caja(10);  
  
        Caramelo c1 = new Caramelo("Sugus", "menta");  
  
        miCaja.agregarElemento(c1);  
  
        Perro p1 = new Perro();  
        Gato g1 = new Gato();  
  
        miCaja.agregarElemento(p1);  
        miCaja.agregarElemento(g1);  
  
        ...  
    }  
}
```



Ejemplo (4)

...

```
//Imprimir los caramelos de la caja
for (Object o : miCaja.getElementos()) {
    if (o instanceof Caramelo) {
        System.out.println(((Caramelo) o).toString());
    }
}
```

Conclusión

- El uso de Object como una referencia genérica es potencialmente inseguro y no se puede hacer nada para que el programador no cometa un error equivocado.
- El error es descubierto en tiempo de ejecución, al momento de realizar el casteo cuando se lanza una excepción del tipo `ClassCastException`.

Solución

```
public class Caja<T> {  
  
    private List<T> elementos = new ArrayList<>();  
    private int tope;  
  
    private Caja(int tope) {  
        this.tope = tope;  
    }  
  
    public boolean agregarElemento(T t) {  
        if (tope < elementos.size()) {  
            elementos.add(t);  
            return true;  
        }  
        return false;  
    }  
}
```

Solución (2)

```
public class Main {  
  
    public static void main(String[] args) {  
        Caja<Caramelo> miCaja = new Caja(10);  
  
        Caramelo c1 = new Caramelo("Sugus", "menta");  
        miCaja.agregarElemento(c1);  
  
        Perro p1 = new Perro();  
        miCaja.agregarElemento(p1);          Error en tiempo de compilación  
  
        //Imprimir los caramelos de la caja  
        for (Caramelo c : miCaja.getElementos()) {  
            System.out.println(c.toString());  
        }  
    }  
}
```

Agenda

— Ejemplo

- **Genericidad**
- Declaración de tipos genéricos
- Declaración de una clase genérica
- Tipos genéricos - Convención
- Ejemplos



Genericidad

- El término “genericidad” se refiere a una serie de técnicas que permiten escribir algoritmos o definir contenedores de forma que puedan aplicarse un amplio rango de tipos de datos.
- Es una construcción importante en los lenguajes de programación orientada a objetos, que si bien no es exclusiva de este tipo de lenguajes, ha adquirido verdadera importancia y uso.
- Permite definir una clase o un método sin especificar el tipo de dato o parámetros, de esta forma se puede cambiar la clase para adaptarla a diferentes usos sin tener que reescribirla.

Genericidad (2)

- La razón de la genericidad se basa principalmente en el hecho de que los algoritmos de resolución de numerosos problemas no dependen del tipo de dato que procesan. Por ejemplo: un algoritmo que implementa una pila de caracteres es igual al que se usa para implementar una pila de números enteros.
- La programación genérica significa escribir un código que puedan reutilizar muchos tipos diferentes de objetos

Agenda

— Ejemplo

— Genericidad

- **Declaración de tipos genéricos**
- Declaración de una clase genérica
- Tipos genéricos - Convención
- Ejemplos



Declaración de tipos genéricos

- Una declaración de tipos genéricos puede tener múltiples tipos parametrizados, separados por comas.
- Todas las invocaciones de clases genéricas son expresiones de una clase. Al instanciar una clase genérica no se crea una nueva clase.
- No se puede usar el tipo genérico <T> como tipo de un campo estático o en cualquier lugar dentro de un método estático o inicializador estático. Por ejemplo:

```
private static final T MI_CONSTANTE = new T();
```

Declaración de tipos genéricos (2)

- No se puede usar un tipo de datos genérico en la creación de objetos y arreglos. Por ejemplo:

```
public class Main {  
  
    public static void main(String[] args) {  
        T[] miArreglo = new T[10];  
  
        T t = new T();  
        miArreglo[0] = t;  
  
        //Imprimir los elementos del arreglo  
        for (int i=0; i<10; i++) {  
            System.out.println(t.toString());  
        }  
    }  
}
```

Declaración de tipos genéricos (3)

- Dentro de una definición de clases, el tipo genérico puede aparecer en cualquier declaración no estática donde se podría utilizar cualquier tipo de datos concreto.

```
public class Caja<T> {  
  
    private List<T> elementos = new ArrayList<>();  
    ...  
  
    public boolean agregarElemento(T t) {  
        ...  
    }  
}
```

Agenda

- ~~- Ejemplo~~
- ~~- Genericidad~~
- ~~- Declaración de tipos genéricos~~
- **Declaración de una clase genérica**
- Tipos genéricos - Convención
- Ejemplos



Declaración de una clase genérica

- Una clase genérica o parametrizable es una clase con una o más variables de tipo genérico.

```
public class MiClase<tipo_genérico> {  
    ...  
}
```

Donde “MiClase” es el nombre de la clase genérica y “tipo_genérico” es el tipo parametrizado genérico.

Agenda

- Ejemplo
- Genericidad
- Declaración de tipos genéricos
- Declaración de una clase genérica
- **Tipos genéricos - Convención**
- Ejemplos



Tipos genéricos - Convención

- E → elemento de una colección
- K → clave
- N → número
- T → tipo
- V → valor
- S, U, V, etc. → para segundos, terceros y cuartos tipos.

Agenda

- Ejemplo
- Genericidad
- Declaración de tipos genéricos
- Declaración de una clase genérica
- Tipos genéricos Convención
- Ejemplos



Ejemplos

- Ejemplo 1:

```
private Box<Integer> numeros = new Box<>();
```

- Ejemplo 2:

```
public interface Par<K, V> {  
    ...  
}  
  
public class ParOrdenado<K, V> implements Pair<K,V> {  
}
```

Ejemplos (2)

- Ejemplo 3:

```
public class NumeroNatural<T extends Integer> {  
    ...  
}
```

- Ejemplo 4:

```
public class A {...}  
  
public interface B {...}  
  
public class C <T extends A & B> {...}
```

Bibliografía oficial

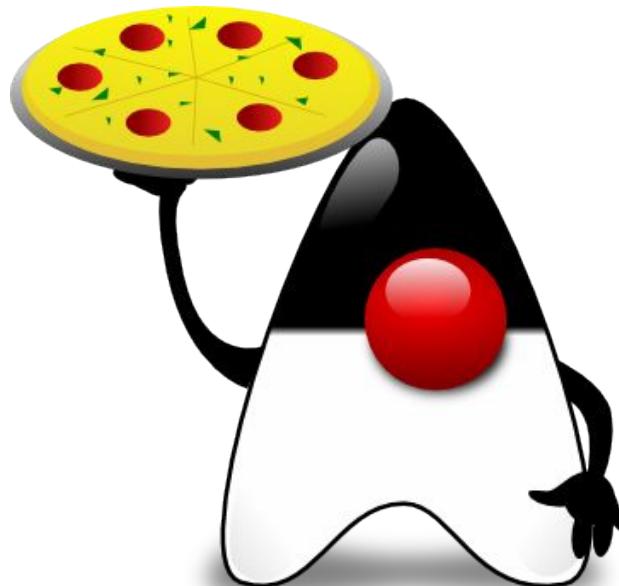
- <https://docs.oracle.com/javase/tutorial/java/generics/types.html>

Clase 16: Enum

— Programación & Laboratorio III —

Agenda

- **Enum - Características**
- Ejemplo 1
- Ejemplo 2
- Más características
- Ejemplo 3
- Operador “==”



Enum - Características

- La palabra reservada “enum” fue introducida en Java 5 para representar el tipo especial de clase que siempre extiende de java.lang.Enum.
- Constantes definidas de esta manera hacen el código más legible, permiten verificación en tiempo de compilación, documentan por adelantado la lista de valores aceptados y evitan el comportamiento inesperado si es que se reciben valores no válidos.
- Ejemplo:

```
public enum PizzaStatus {  
    ORDERED,  
    READY,  
    DELIVERED;  
}
```

Enum - Características (2)

- Un tipo enumerado restringe los posibles valores que puede tomar una variable. Esto ayuda a reducir los errores en el código y permite algunos usos especiales interesantes.
- Por convención, los nombres de los valores que puede tomar **se escriben en letras mayúsculas** para recordarnos que son valores fijos (que en cierto modo podemos ver como constantes).
- Una vez declarado el tipo enumerado, todavía no existen variables hasta que no las creamos explícitamente:

```
TipoEnumerado nombreVariable; → PizzaStatus status;
```

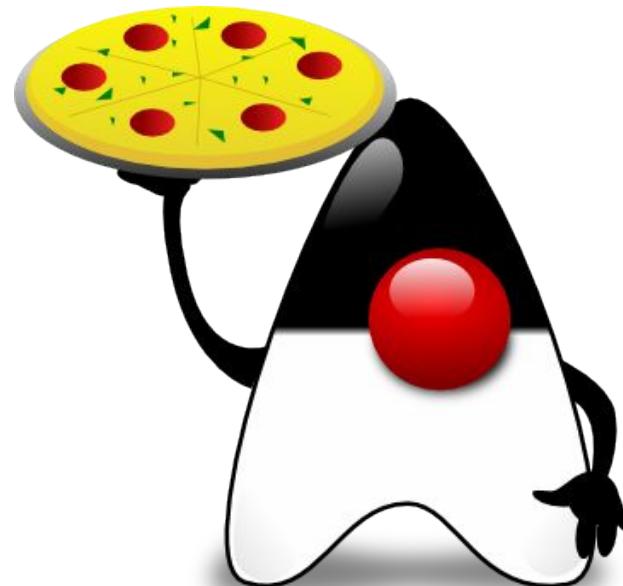
Enum - Características (3)

- Un tipo enumerado puede ser declarado dentro o fuera de una clase, pero no dentro de un método. Por tanto no podemos declarar un enum dentro de un método main; si lo hacemos nos saltará el error de compilación “enum types must not be local” (los tipos enumerados no deben ser locales a un método).
- Declararemos un tipo enumerado como una clase aparte o dentro de otra, pero fuera de cualquier método o constructor.

Agenda

~~Enum Características~~

- Ejemplo 1
- Ejemplo 2
- Más características
- Ejemplo 3
- Operador “==”



Ejemplo 1: Enum declarado dentro de clase

```
public class Pizza {  
  
    private PizzaStatus status;  
  
    public enum PizzaStatus {  
        ORDERED,  
        READY,  
        DELIVERED;  
    }  
  
    public boolean isDeliverable() {  
        if (status == PizzaStatus.READY) {  
            return true;  
        }  
        return false;  
    }  
}
```

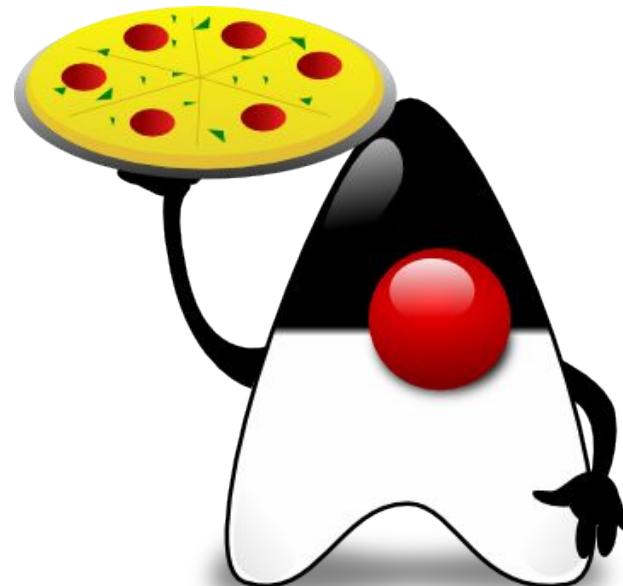
Agenda

~~Enum Características~~

~~Ejemplo 1~~

- **Ejemplo 2**

- Más características
- Ejemplo 3
- Operador “==”



Ejemplo 2: Enum como clase

```
public enum PizzaStatus {  
    ORDERED,  
    READY,  
    DELIVERED;  
  
    public boolean isDeliverable() {  
        if (status == PizzaStatus.READY) {  
            return true;  
        }  
        return false;  
    }  
}
```

Agenda

~~— Enum Características~~

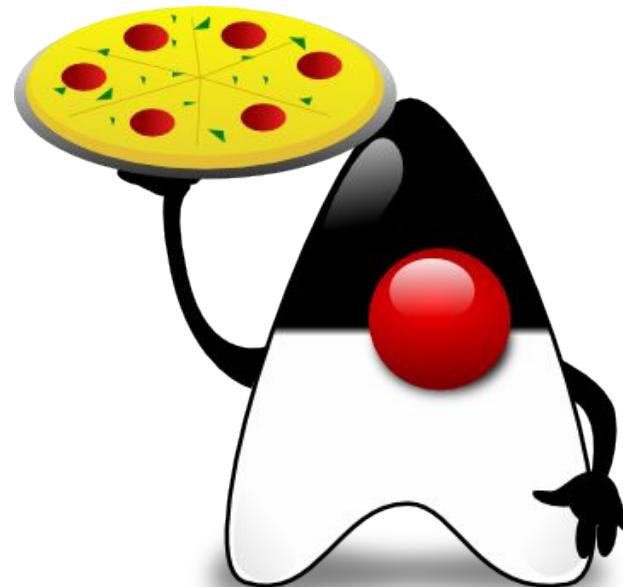
~~— Ejemplo 1~~

~~— Ejemplo 2~~

- **Más características**

- Ejemplo 3

- Operador “==”

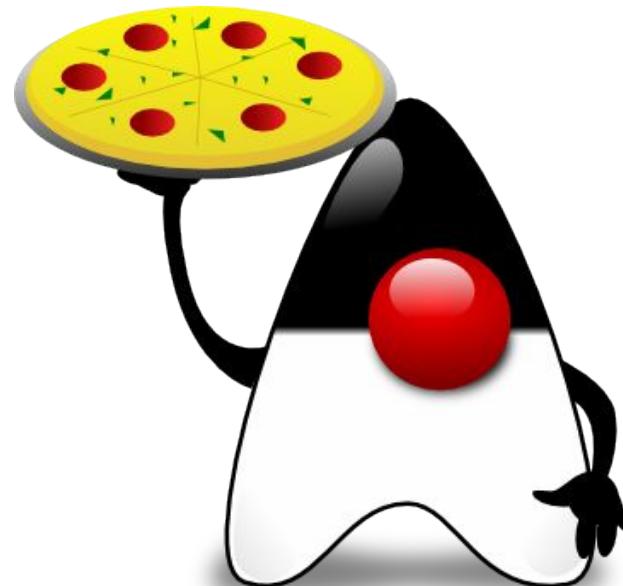


Más características

- Se dispone automáticamente de métodos especiales como por ejemplo el método `values()`, que el compilador agrega automáticamente cuando se crea un enum. Este método devuelve un array conteniendo todos los valores del enumerado en el orden en que son declarados
- Un tipo enumerado puede añadir campos constantes al objeto enumerado y recuperar esos campos. Para ello se usa un constructor especial para tipos enumerados.

Agenda

- ~~Enum Características~~
- ~~Ejemplo 1~~
- ~~Ejemplo 2~~
- ~~Más características~~
- **Ejemplo 3**
- Operador “==”



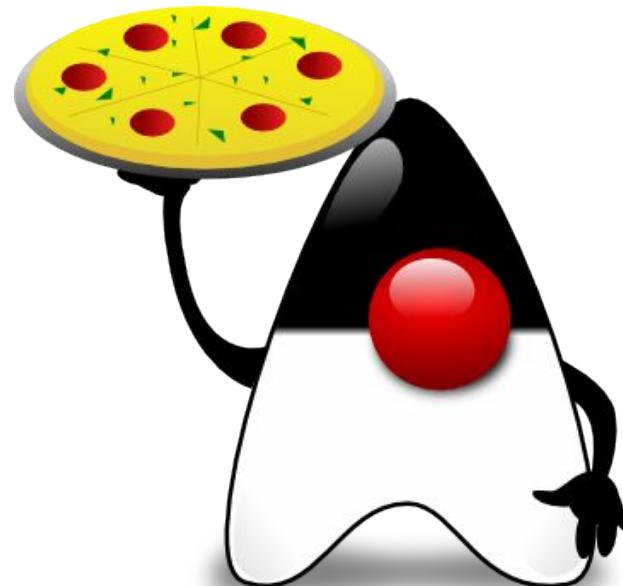
Ejemplo 3: Enum con constructor

```
public enum TipoDeMadera {  
    ROBLE ("Castaño verdoso"),  
    CAOBA ("Marrón oscuro"),  
    NOGAL("Marrón rojizo");  
  
    private String color;  
  
    TipoDeMadera (String color) {  
        this.color = color;  
    }  
  
    public String getColor() {  
        return color;  
    }  
}
```

Por defecto el constructor es “private”, es redundante escribirlo.

Agenda

- ~~Enum Características~~
- ~~Ejemplo 1~~
- ~~Ejemplo 2~~
- ~~Más características~~
- ~~Ejemplo 3~~
- **Operador “==”**



Operador “==”

- Usando el tipo enum nos aseguramos que sólo existe una instancia de la constante en la JVM. por lo tanto se puede usar el operador “==” para comparar dos variables del mismo tipo enum.
- Seguridad en tiempo de ejecución:

```
1) if(pizza.getStatus().equals(Pizza.PizzaStatus.DELIVERED));  
2) if(pizza.getStatus() == Pizza.PizzaStatus.DELIVERED);
```

En la opción 1) si el status de pizza es null, obtendremos una excepción del tipo NullPointerException. No así en la opción 2)

Bibliografía oficial

- <https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html>
- <https://docs.oracle.com/javase/8/docs/technotes/guides/language/enums.html>

Clase 17: Excepciones

— Programación & Laboratorio III —

Agenda

- Concepto de excepción
- Tipos de excepciones
- Excepciones checked o comprobadas
- Excepciones unchecked o no comprobadas
- Lanzamiento de excepciones
- Transferencia de control
- Cláusula throws
- Creando nuestras propias excepciones
- Malas prácticas
- Buenas prácticas



Concepto de excepción

- Una excepción en Java es un error o situación “excepcional” que se produce durante la ejecución de un programa.
- Ejemplos:
 - Leer un archivo que no existe
 - Acceder al valor N de una colección que tiene menos de N elementos.
 - Enviar o recibir información por red mientras se produce una pérdida de conectividad.
- Todas las excepciones en Java se representan a través de objetos que heredan de `java.lang.Throwable`.

Concepto de excepción (2)

- Las excepciones proporcionan una forma clara de comprobar posibles errores sin oscurecer el código.
- Convierten las condiciones de error que un método puede señalar en una parte explícita del contrato del método.
- La lista de excepciones pueden ser vistas por el programador, comprobada por el compilador y preservada por las clases extendidas que redefinen el método.

Agenda

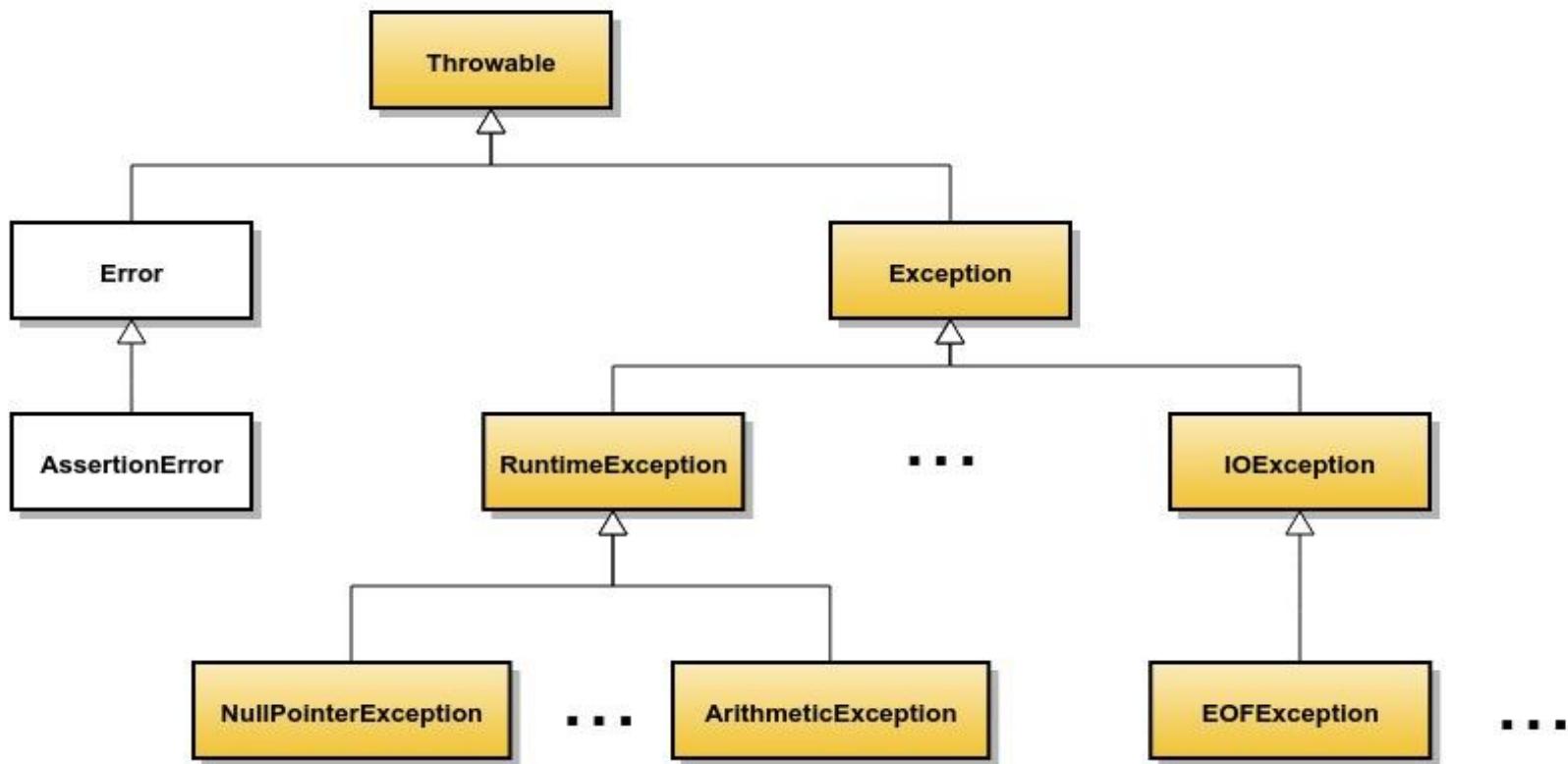
— ~~Concepto de excepción~~

- **Tipos de excepciones**

- Excepciones checked o comprobadas
- Excepciones unchecked o no comprobadas
- Lanzamiento de excepciones
- Transferencia de control
- Cláusula throws
- Creando nuestras propias excepciones
- Malas prácticas
- Buenas prácticas



Tipos de excepciones



Tipos de excepciones (2)

- Las excepciones también son objetos.
- Todos los tipos de excepción deben extender de la clase Throwable o de alguna de sus subclases.
- De Throwable nacen dos ramas: Error y Exception.
 - 1) Error representa errores de una magnitud tal que una aplicación nunca debería intentar realizar nada con ellos. Por ejemplo: errores de la JVM, desbordamiento de buffer.
 - 2) Exception representa aquellos errores que normalmente sí solemos gestionar.

Tipos de excepciones (3)

- Por convenio, los nuevos tipos de excepción extienden a Exception.
- De Exception nacen múltiples ramas: ClassNotFoundException, IOException, ParseException, SQLException → **Excepciones checked**.
- RuntimeException es la única **excepción unchecked**.

Agenda

~~Concepto de excepción~~

~~Tipos de excepciones~~

- **Excepciones checked o comprobadas**
- Excepciones unchecked o no comprobadas
- Lanzamiento de excepciones
- Transferencia de control
- Cláusula throws
- Creando nuestras propias excepciones
- Malas prácticas
- Buenas prácticas



Excepciones checked (comprobadas)

- Una excepción de tipo checked representa un error del cual técnicamente podemos recuperarnos.
- Son todas las situaciones que son totalmente ajenas al propio código, por ejemplo: fallo de una operación de lectura/escritura.
- Este tipo de excepciones deben ser **capturadas o relanzadas**.

Agenda

- Concepto de excepción
- Tipos de excepciones
- Excepciones checked o comprobadas
- **Excepciones unchecked o no comprobadas**
- Lanzamiento de excepciones
- Transferencia de control
- Cláusula throws
- Creando nuestras propias excepciones
- Malas prácticas
- Buenas prácticas



Excepciones unchecked (no comprobadas)

- Representan errores de programación. Por ejemplo: intentar leer de un array de N elementos un elemento que se encuentra en una posición mayor que N.

```
int[] numerosPrimos = {1, 3, 5, 7, 9, 11, 13, 17, 19, 23};  
System.out.println(numerosPrimos[10]); → ArrayIndexOutOfBoundsException
```

Agenda

- ~~Concepto de excepción~~
- ~~Tipos de excepciones~~
- ~~Excepciones checked o comprobadas~~
- ~~Excepciones unchecked o no comprobadas~~
- **Lanzamiento de excepciones**
- Transferencia de control
- Cláusula throws
- Creando nuestras propias excepciones
- Malas prácticas
- Buenas prácticas



Lanzamiento de excepciones

- Las excepciones se lanzan utilizando la palabra reservada throw:

```
throw AException
```

- AException debe ser un objeto Throwable.
- Si se lanza la excepción no se regresa al flujo normal del programa.
- Las excepciones son objetos, por lo tanto se debe crear una instancia antes de lanzarse.

Lanzamiento de excepciones (2)

- Ejemplo:

```
public void buscarNumeroPrimo(int n) {  
    for (int i=0; i<10; i++) {  
        if (numerosPrimos[i] == n) {  
            System.out.println("El número " + n + " existe.");  
            break;  
        }  
    }  
    throw new NumeroNoExisteException(n);  
}
```

Agenda

- ~~— Concepto de excepción~~
- ~~— Tipos de excepciones~~
- ~~— Excepciones checked o comprobadas~~
- ~~— Excepciones unchecked o no comprobadas~~
- ~~— Lanzamiento de excepciones~~
- **Transferencia de control**
 - Cláusula throws
 - Creando nuestras propias excepciones
 - Malas prácticas
 - Buenas prácticas



Transferencia de control

- Una vez que se produce una excepción, las acciones que hubiera detrás del punto donde se produjo la excepción no tienen lugar.
- Las acciones que sí se ejecutarán serán las contenidas dentro de los bloques `catch` y `finally`.

Transferencia de control (2)

- Ejemplo:

```
public static void main(String[] args) {  
    FileWriter fichero = null;  
    try {  
        fichero = new FileWriter(PATH_ARCHIVO);  
        fichero.write("Escribiendo...")  
    } catch(IOException e) {  
        e.printStackTrace();  
    } finally {  
        if (fichero != null) {  
            fichero.close();  
        }  
    }  
}
```

Transferencia de control (3)

- Las sentencias dentro de la cláusula `try` se ejecutan hasta que se lance una excepción o hasta que finalice con éxito.
- Si se lanza una excepción, se examinan sucesivamente las sentencias de la cláusula `catch`.
- La cláusula `finally` de una sentencia proporciona un mecanismo para ejecutar una sección de código, se lance o no una excepción.
- Generalmente la cláusula `finally` se utiliza para limpiar el estado interno o para liberar recursos, como archivos abiertos o cerrar conexiones a bases de datos.

Agenda

- ~~— Concepto de excepción~~
- ~~— Tipos de excepciones~~
- ~~— Excepciones checked o comprobadas~~
- ~~— Excepciones unchecked o no comprobadas~~
- ~~— Lanzamiento de excepciones~~
- ~~— Transferencia de control~~
- **Cláusula throws**
- Creando nuestras propias excepciones
- Malas prácticas
- Buenas prácticas



Cláusula throws

- Se utiliza para declarar las excepciones checked que puede lanzar un método.
- Se pueden declarar varias, separadas por comas.
- RuntimeException y Error son las únicas excepciones que no hace falta incluir en las cláusulas throws.

Cláusula throws (2)

- Si se invoca a un método que tiene una excepción checked en su cláusula throws, existen tres opciones:
 - 1) Capturar la excepción y gestionarla.

```
public void abrirArchivo(String path) throws IOException {  
    ...  
}  
  
public void leerArchivo(String path) {  
    try {  
        abrirArchivo(path);  
    } catch(IOException e) {  
        System.out.println("Se produjo un error al abrir el archivo").  
    }  
}
```

Cláusula throws (3)

- 2) Capturar la excepción y transformarla en una de nuestras excepciones.

```
public void abrirArchivo(String path) throws IOException {  
    ...  
}  
  
public void leerArchivo(String path) {  
    try {  
        abrirArchivo(path);  
    } catch(IOException e) {  
        throw new ArchivoNoExisteException(e);  
    }  
}
```

Cláusula throws (4)

- 3) Declarar la excepción en la cláusula throws y hacer que otro método la gestione.

```
public void abrirArchivo(String path) throws IOException {  
    ...  
}  
  
public void leerArchivo(String path) throws IOException {  
    abrirArchivo(path);  
    ...  
}
```

Cláusula throws (5)

- No se permite que los métodos redefinidos declaren más excepciones checked en la cláusula throws que las que declara el método.
- Se pueden lanzar subtipos de las excepciones declaradas ya que podrán ser capturadas en el bloque catch correspondiente a su supertipo.
- Si una declaración de un método se hereda en forma múltiple, la cláusula throws de ese método

Agenda

- ~~— Concepto de excepción~~
- ~~— Tipos de excepciones~~
- ~~— Excepciones checked o comprobadas~~
- ~~— Excepciones unchecked o no comprobadas~~
- ~~— Lanzamiento de excepciones~~
- ~~— Transferencia de control~~
- ~~— Cláusula throws~~
- **Creando nuestras propias excepciones**
- Malas prácticas
- Buenas prácticas



Creando nuestras excepciones

```
public class SaldoInsuficienteException extends Exception {  
    ...  
}  
  
public static void main(String[] args) {  
    Cliente cliente = new Cliente();  
    try {  
        cliente.pagar();  
    } catch(SaldoInsuficienteException e) {  
        e.printStackTrace();  
    }  
}
```

Agenda

- Concepto de excepción
- Tipos de excepciones
- Excepciones checked o comprobadas
- Excepciones unchecked o no comprobadas
- Lanzamiento de excepciones
- Transferencia de control
- Cláusula throws
- Creando nuestras propias excepciones
 - **Malas prácticas**
 - Buenas prácticas



Malas prácticas

1

```
FileWriter fichero = null;
try {
    fichero = new FileWriter("path");
    fichero.write("Writing...");
} catch (IOException e) {
}
```

2

```
int[] numerosPrimos = {1, 3, 5, 7, 9, 11, 13, 17, 19, 23};
int undecimoPrimo = numerosPrimos[10];
try {
    int i = 0;
    while(true) {
        System.out.println(numerosPrimos[i++]);
    }
} catch(ArrayIndexOutOfBoundsException e) {}
```

Malas prácticas

3

```
public void pagar() throws Exception{  
    //Do something  
}
```

4

```
public void pagar() {  
    try {  
    } catch (Exception e) {  
        throw new RuntimeException();  
    }  
}
```

Agenda

- ~~Concepto de excepción~~
- ~~Tipos de excepciones~~
- ~~Excepciones checked o comprobadas~~
- ~~Excepciones unchecked o no comprobadas~~
- ~~Lanzamiento de excepciones~~
- ~~Transferencia de control~~
- ~~Cláusula throws~~
- ~~Creando nuestras propias excepciones~~
- ~~Malas prácticas~~
- **Buenas prácticas**



Buenas prácticas

- Utilizar excepciones que ya existen.
- Lanzar excepciones de acuerdo al nivel de abstracción en el que nos encontramos.
- Documentar con Javadoc todas las excepciones que se lanzan en los métodos .

Bibliografía oficial

- <https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html>
- <https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html>
- <https://docs.oracle.com/javase/tutorial/essential/exceptions/handling.html>
- <https://docs.oracle.com/javase/tutorial/essential/exceptions/throwing.html>
- <https://docs.oracle.com/javase/tutorial/essential/exceptions/advantages.html>

Clase 18: Manejo de archivos

— Programación & Laboratorio III —

Agenda

- **Concepto de archivo**
- Archivos en Java
- Concepto de buffering
- Escribir un archivo
- Leer un archivo
- Cerrar un archivo



Concepto de archivo

- Desde el punto de vista de más bajo nivel, podemos definir un archivo (o fichero) como un conjunto de bits almacenados en un dispositivo, y accesible a través de un camino de acceso (pathname) que lo identifica

Por qué usar archivos?

- Con frecuencia tendremos que guardar los datos de nuestro programa para poderlos recuperar más adelante. Los archivos se usan cuando el volumen de datos no es relativamente muy elevado.

Concepto de archivo (2)

- Los archivos suelen organizarse en estructuras jerárquicas de directorios. Estos directorios son contenedores de ficheros (y de otros directorios) que permiten organizar los datos del disco.
- El nombre de un archivo está relacionado con su posición en el árbol de directorios que lo contiene, lo que permite no sólo identificar únicamente cada fichero, sino encontrarlo en el disco a partir de su nombre.
- Los archivos suelen tener una serie de metadatos asociados, como pueden ser su fecha de creación, la fecha de última modificación, su propietario o los permisos que tienen diferentes usuarios sobre ellos

Agenda

- Concepto de archivo
- **Archivos en Java**
 - Concepto de buffering
 - Escribir un archivo
 - Leer un archivo
 - Cerrar un archivo



Archivos en Java

- La clase `File` en java representa un fichero y nos permite obtener información sobre él, como por ejemplo atributos, directorios, etc.
- Tiene tres constructores:
 - `File(String path)`
 - `File(String path, String name)`
 - `File(File dir, String name)`

El parámetro “path” indica el camino hacia el directorio donde se encuentra el archivo, y “name” indica el nombre del archivo.

Archivos en Java (2)

- Algunos métodos importantes:

- String getName()
- String getPath()
- boolean exists()
- boolean canWrite()
- boolean canRead
- boolean isFile()
- boolean isDirectory()
- long lastModified()
- boolean renameTo(File dest);
- boolean delete()
- String[] list()



Archivos en Java (3)

- Al instanciar la clase `File` con un archivo, no se abre el fichero ni es necesario que este exista y precisamente esa es la primera información útil que podemos obtener del fichero: saber si existe o no.

```
import java.io.File;  
...  
File f = new File("path/mi_fichero.txt");  
if (f.exists())  
    // El fichero existe.  
else  
    // el fichero no existe.
```

Archivos en Java (4)

- Si el archivo existe, es decir, si la función exists() devuelve true, entonces podemos obtener información acerca del archivo. Por ejemplo:

```
if(f.exists()){\n    System.out.println("Nombre del archivo " + f.getName());\n    System.out.println("Camino           " + f.getPath());\n    System.out.println("Camino absoluto   " + f.getAbsolutePath());\n    System.out.println("Se puede escribir " + f.canRead());\n    System.out.println("Se puede leer     " + f.canWrite());\n    System.out.println("Tamaño           " + f.length());\n}
```

Archivos en Java (5)

- Si el archivo existe, podemos no tener permisos para leerlo, así que otra comprobación útil es si se puede hacer en él la operación deseada.

```
if (f.canRead())
    // El archivo existe y se puede leer.
```

```
if (f.canWrite())
    // El archivo existe y se puede escribir en él
```

```
if (f.canExecute())
    // El archivo existe y se puede ejecutar
```

Archivos en Java (6)

- Tenemos, además, los correspondientes métodos `setReadable()`, `setReadOnly()`, `setWritable()` y `setExcutable()` que nos permiten cambiar las propiedades del archivo siempre que seamos los propietarios del mismo o tengamos permisos para hacerlo.

Archivos en Java (7)

- La siguiente comprobación útil que se puede hacer con un `File` es determinar si es un archivo normal o es un directorio. Si es un directorio, podemos obtener un listado de los ficheros contenidos en él.

```
// Se crea el File del directorio
File directorio = new File("src/main/java/com/chuidiang/ejemplos/");

// Si es un directorio
if (directorio.isDirectory()) {
    // obtenemos su contenido
    File[] ficheros = directorio.listFiles();

    for (File fichero : ficheros) {
        System.out.println(fichero.getName());
    }
}
```

Archivos en Java (8)

- Son archivos que podremos crear desde un programa Java y leer con cualquier editor de texto, o bien crear con un editor de textos y leer desde un programa Java.
- Para manipular archivos, siempre tendremos los mismos pasos:
 - 1) Abrir el archivo → Si no abrimos el archivo, obtendremos un mensaje de error al intentar acceder a su contenido.
 - 2) Escribir datos o leerlos.
 - 3) Cerrar el archivo → Si no cerramos el archivo, puede que realmente no se llegue a guardar ningún dato.

Agenda

- ~~- Concepto de archivo~~
- ~~- Archivos en Java~~
- **Concepto de buffering**
- Escribir un archivo
- Leer un archivo
- Cerrar un archivo



Concepto de Buffering

"Si no hubiera buffers, sería como comprar sin un carrito: debería llevar los productos uno a uno hasta la caja. Los buffers te dan un lugar en el que dejar temporalmente las cosas hasta que esté lleno. Por eso se realizarán menos viajes cuando usas el carrito"

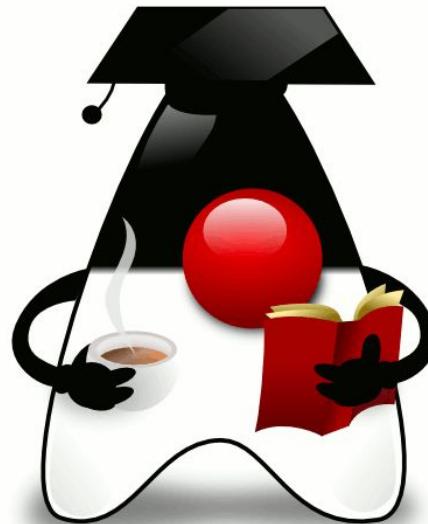
- Cualquier operación que implique acceder a memoria externa es muy costosa, por lo que es interesante reducir las operaciones de lectura/escritura que realizamos sobre los archivos.

Concepto de Buffering (2)

- Un buffer es una estructura de datos que permite el acceso por trozos a una colección de datos.
- Los buffers son útiles para evitar almacenar en memoria grandes cantidades de datos, en lugar de ello, se va pidiendo al buffer porciones pequeñas de los datos que se van procesando por separado. También resultan muy útiles para que la aplicación pueda ignorar los detalles concretos de eficiencia de hardware subyacente, la aplicación puede escribir al buffer cuando quiera, que ya se encargará el buffer de escribir a disco siguiendo los ritmos más adecuados y eficientes.

Agenda

- ~~- Concepto de archivo~~
- ~~- Archivos en Java~~
- ~~- Concepto de buffering~~
- **Escribir un archivo**
- Leer un archivo
- Cerrar un archivo



Escribir archivo

- Para abrir un archivo usaremos un BufferedWriter, que se apoya en un FileWriter, que a su vez usa un "File" al que se le indica el nombre del archivo

```
BufferedWriter f = new BufferedWriter(  
    new FileWriter(new File("mi_archivo.txt")));
```

- En el caso de un archivo de texto, no escribiremos con "println()", como hacíamos en pantalla, sino con "write()". Cuando queramos avanzar a la línea siguiente, deberemos usar "newLine()":

Escribir archivo (2)

```
try {  
    BufferedWriter fSalida = new BufferedWriter(new FileWriter(new File("mi_archivo.txt")));  
    fSalida.write("Hola");  
    fSalida.newLine();  
    fSalida.write("Mundo!");  
    fSalida.newLine();  
    fSalida.close();  
} catch (IOException e) {  
    System.out.println("Se produjo un error al escribir en el archivo: " + e.getMessage());  
}
```

Agenda

- ~~Concepto de archivo~~
- ~~Archivos en Java~~
- ~~Concepto de buffering~~
- ~~Escribir un archivo~~
- **Leer un archivo**
- Cerrar un archivo



Leer archivo

- Para leer de un fichero de texto usaremos "readLine()", que nos devuelve un String. Si ese String es null, quiere decir que se ha acabado el archivo y no se ha podido leer nada. Por eso, lo habitual es usar un "while" para leer todo el contenido de un fichero.
- Existe otra diferencia con la escritura: no usaremos un BufferedWriter, sino un BufferedReader, que se apoyará en un FileReader:

```
BufferedReader fEntrada = new BufferedReader(  
    new FileReader(new File("mi_archivo.txt")));
```

Leer archivo (2)

- La clase `java.io.BufferedReader` resulta ideal para leer archivos de texto y procesarlos. Permite leer eficientemente caracteres aislados, arrays o líneas completas como Strings. Cada lectura a un BufferedReader provoca una lectura en el archivo correspondiente al que está asociado. Es el propio BufferedReader el que se va encargando de recordar la última posición del fichero leído, de forma que posteriores lecturas van accediendo a posiciones consecutivas del fichero.

Leer archivo (3)

```
if (!(new File("mi_archivo.txt")).exists()) {  
    return;  
}  
try {  
    BufferedReader fEntrada = new BufferedReader(new FileReader(new File("mi_archivo.txt")));  
    String linea=null;  
    while ((linea=fEntrada.readLine()) != null) {  
        System.out.println(linea);  
    }  
    fEntrada.close();  
} catch (IOException e) {  
    System.out.println("Se produjo un error al escribir en el archivo: " + e.getMessage());  
}
```

Agenda

- ~~Concepto de archivo~~
- ~~Archivos en Java~~
- ~~Concepto de buffering~~
- ~~Escribir un archivo~~
- ~~Leer un archivo~~
- **Cerrar un archivo**



Cerrar

```
if (!(new File("mi_archivo.txt")).exists()) {  
    return;  
}  
try {  
    BufferedReader fEntrada = new BufferedReader(new FileReader(new File("mi_archivo.txt")));  
    String linea=null;  
    while ((linea=fEntrada.readLine()) != null) {  
        System.out.println(linea);  
    }  
    fEntrada.close();  
} catch (IOException e) {  
    System.out.println("Se produjo un error al escribir en el archivo: " + e.getMessage());  
}
```

Cerrar

```
if (!(new File("mi_archivo.txt")).exists()) {  
    return;  
}  
try {  
    BufferedReader fEntrada = new BufferedReader(new FileReader(new File("mi_archivo.txt")));  
    String linea=null;  
    while ((linea=fEntrada.readLine()) != null) {  
        System.out.println(linea);  
    }  
} catch (IOException e) {  
    System.out.println("Se produjo un error al escribir en el archivo: " + e.getMessage());  
} finally {  
    fEntrada.close();  
}
```

Bibliografía oficial

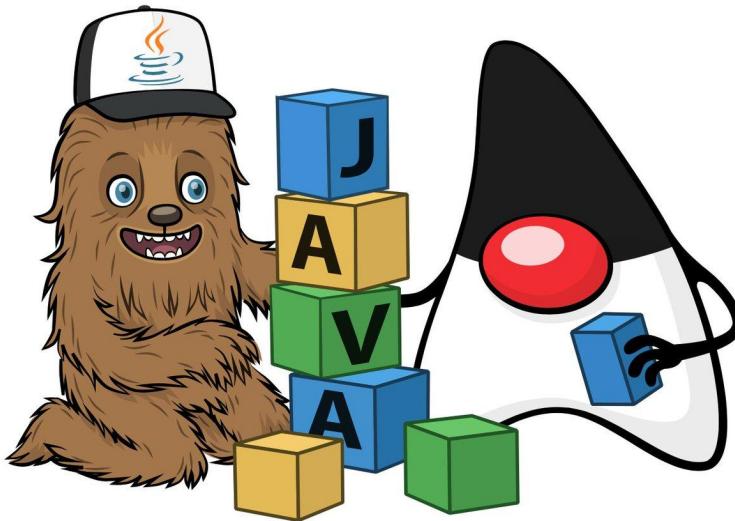
- <https://docs.oracle.com/javase/tutorial/essential/io/file.html>
- <https://docs.oracle.com/javase/7/docs/api/java/io/File.html>

Clase 19: Serialización & JSON

— Programación & Laboratorio III —

Agenda

- **Serialización**
- Interfaz Serializable
- Serial Version UID
- Modificador transient
- Serialización y archivos
 - ObjectOutputStream
 - ObjectInputStream
- JSON
- Librerías para procesamiento de JSON
 - Jackson
 - Gson



Serialización

- La serialización permite convertir cualquier objeto (que implemente la interfaz Serializable) en una secuencia de bytes que pueden ser posteriormente leídos para restaurar el objeto original.
- Esta característica se mantiene incluso a través de la red, por lo que podemos crear un objeto en una máquina que corre bajo el sistema operativo Windows, serializarlo y enviarlo a través de la red a una máquina que corre bajo UNIX donde será correctamente reconstruido.

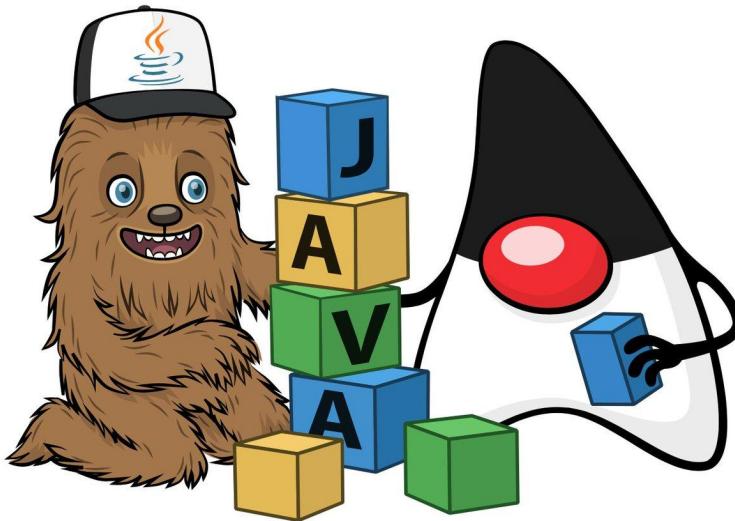
Serialización (2)

- La serialización es una característica añadida al lenguaje Java para dar soporte a:
 - La invocación remota de objetos
 - La persistencia.
- La invocación remota de objetos permite a los objetos que “viven” en otras máquinas comportarse como si vivieran en nuestra propia máquina.
- Para la persistencia, la serialización nos permite guardar el estado de un componente en disco, abandonar el IDE y restaurar el estado de dicho componente cuando se vuelve a correr el IDE.

Agenda

— Serialización

- **Interfaz Serializable**
- Serial Version UID
- Modificador transient
- Serialización y archivos
 - ObjectOutputStream
 - ObjectInputStream
- JSON
- Librerías para procesamiento de JSON
 - Jackson
 - Gson



Interfaz Serializable

- Un objeto se puede serializar si implementa la interfaz Serializable. Esta interfaz no declara ninguna función, se trata de una interfaz vacía.

```
public interface Serializable{  
}
```

- Para hacer una clase serializable simplemente debemos implementar esta interfaz:

```
public class Persona implements Serializable{  
    private String nombre;  
    ...  
}
```

Interfaz Serializable (2)

- Si dentro de la clase hay atributos que son otras clases, éstos a su vez también deben implementar Serializable. Con los tipos de java (String, Integer, etc.) no hay problema porque lo son. Por ejemplo:

```
public class Persona implements Serializable {  
    private String nombre;  
    private Direccion direccion;  
}  
  
public class Direccion implements Serializable {  
    private String calle;  
    private String ciudad;  
    ...  
}
```

Interfaz Serializable (3)

- La serialización de un objeto como atributo de otro objeto se puede tomar como un árbol, donde las hojas son objetos que forman parte de otro objeto como un atributo, y todos ellos son marcados como serializables, para así entonces serializar primero las hojas del árbol y finalizar con la raíz.

Agenda

— ~~Serialización~~

— ~~Interfaz Serializable~~

- **Serial Version UID**

- Modificador transient

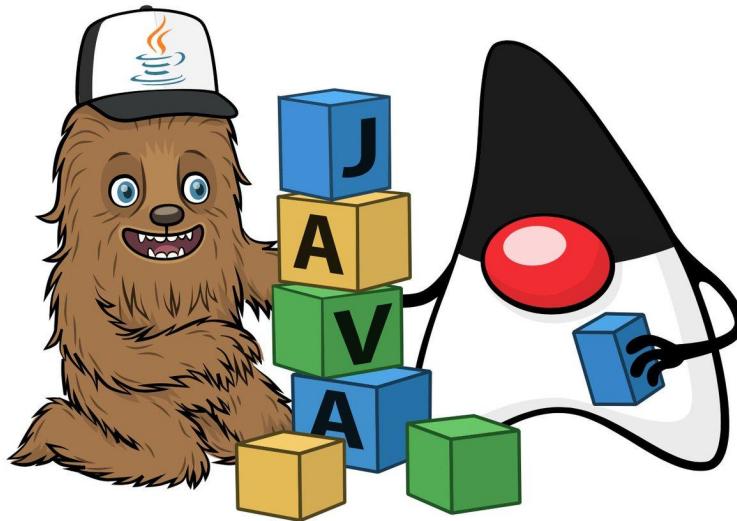
- Serialización y archivos

- ObjectOutputStream
- ObjectInputStream

- JSON

- Librerías para procesamiento de JSON

- Jackson
- Gson



Serial Version UID

- Cuando pasamos objetos serializables de un lado a otro, puede llegar a pasar que en las distintas versiones de nuestro programa la clase cambie. De esta manera es posible que un lado tenga una versión más antigua que en el otro lado. Si esto sucede, la reconstrucción de la clase en el lado que recibe es imposible.
- Para evitar este problema, se aconseja que la clase que queremos serializar tenga un atributo de la siguiente forma:

```
public class Persona implements Serializable {  
    private static final long serialVersionUID = 8799656478674716638L;  
  
    ...  
}
```

Serial Version UID (2)

- El número del final debe ser distinto para cada versión de compilado que tengamos.
- De esta forma, la JVM es capaz de detectar rápidamente que las versiones en ambos lados son diferentes.

Agenda

~~— Serialización~~

~~— Interfaz Serializable~~

~~— Serial Version UID~~

- **Modificador transient**

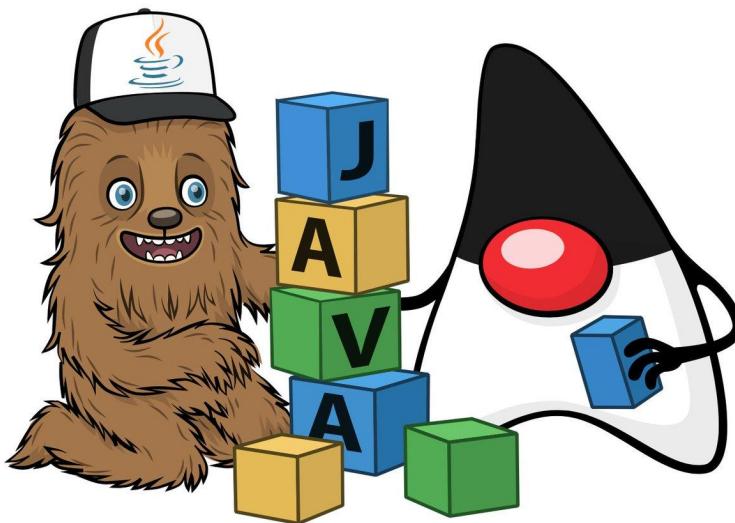
- Serialización y archivos

- ObjectOutputStream
- ObjectInputStream

- JSON

- Librerías para procesamiento de JSON

- Jackson
- Gson



Modificador transient

- Habrá ocasiones donde no será necesario incluir un atributo del objeto en la serialización, y para esto se encuentra el modificador transient.
- Este modificador le indica a la JVM que dicho atributo deberá ser exento de la serialización, en otras palabras, ignorará este atributo.
- Por otro lado, los atributos que lleven el modificador static nunca se tomarán en cuenta al serializar un objeto, ya que este atributo pertenece a la clase y no al objeto.

Modificador transient (2)

- Ejemplo:

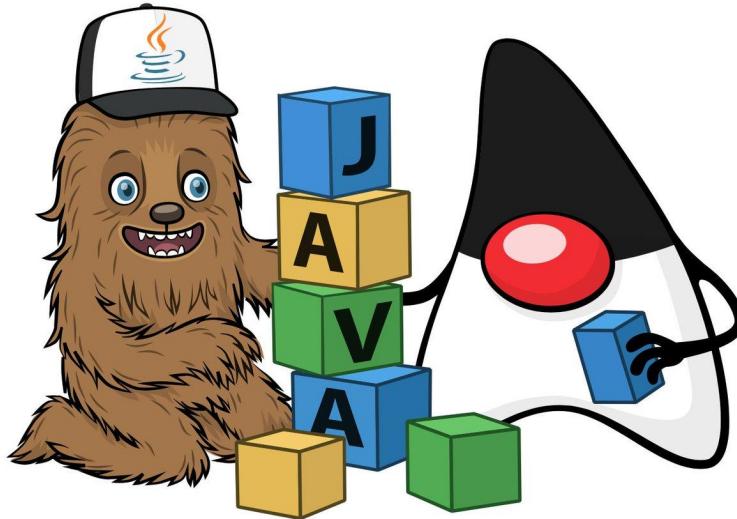
```
public class Persona implements Serializable {  
    private String nombre;  
    private String apellidoCasada;  
    private transient String apellidoSoltera;  
  
    ...  
}
```

Agenda

~~— Serialización~~
~~— Interfaz Serializable~~

~~— Serial Version UID~~
~~— Modificador transient~~

- **Serialización y archivos**
 - ObjectOutputStream
 - ObjectInputStream
- JSON
- Librerías para procesamiento de JSON
 - Jackson
 - Gson



Serialización y archivos

- Un uso de la serialización es la lectura y escritura de objetos en un archivo.
- Ejemplo:

```
public class Persona implements Serializable {  
    private String nombre;  
    private Direccion direccion;  
}  
  
public class Direccion implements Serializable {  
    private String calle;  
    private String ciudad;  
    ...  
}
```

Serialización y archivos (2)

- Escribir en el archivo:

```
File file = new File("mi_archivo.txt");
ObjectOutputStream objOutputStream = new ObjectOutputStream(new
FileOutputStream(file));

Persona p = new Persona();
objOutputStream.writeObject(p);

objOutputStream.close();
```

Serialización y archivos (3)

- Leer archivo:

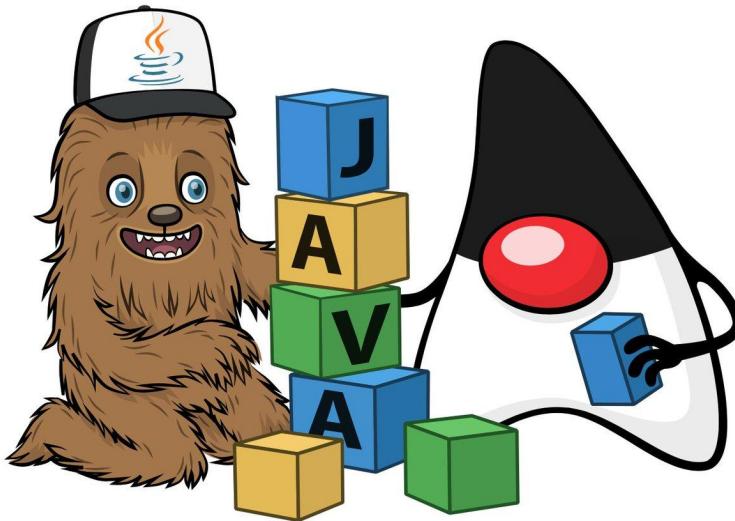
```
File file = new File("mi_archivo.txt");
ObjectInputStream objInputStream = new ObjectInputStream(new
FileInputStream(file));

Object aux = objInputStream.readObject();
while (aux!=null) {
    if (aux instanceof Persona)
        System.out.println(aux);
    aux = objInputStream.readObject();
}

objInputStream.close();
```

Agenda

- ~~Serialización~~
- ~~Interfaz Serializable~~
- ~~Serial Version UID~~
- ~~Modificador transient~~
- ~~Serialización y archivos~~
 - **ObjectOutputStream**
 - **ObjectInputStream**
- **JSON**
- **Librerías para procesamiento de JSON**
 - Jackson
 - Gson

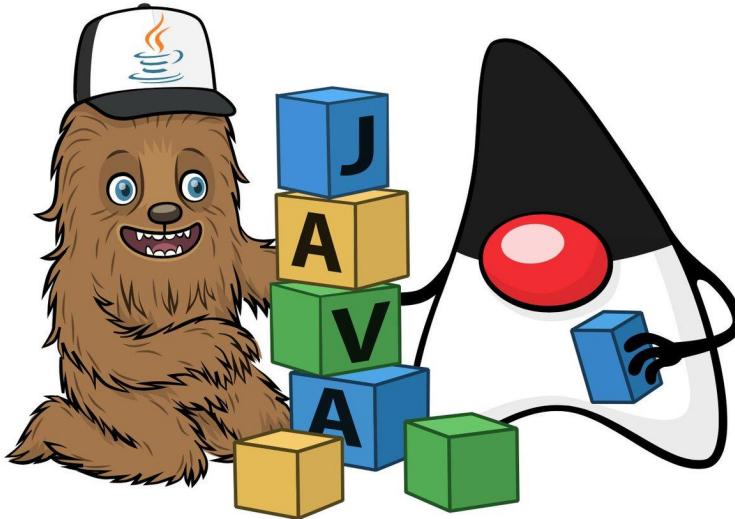


ObjectOutputStream

- Se utiliza para escribir objetos en un OutputStream en lugar de escribir el objeto convertido a bytes. De esta manera se encapsula el OutputStream en un ObjectOutputStream.
- Sólo los objetos que implementen la interfaz Serializable pueden escribirse en los streams.

Agenda

- ~~Serialización~~
- ~~Interfaz Serializable~~
- ~~Serial Version UID~~
- ~~Modificador transient~~
- ~~Serialización y archivos~~
 - `ObjectOutputStream`
 - `ObjectInputStream`
- JSON
 - Jackson
 - Gson

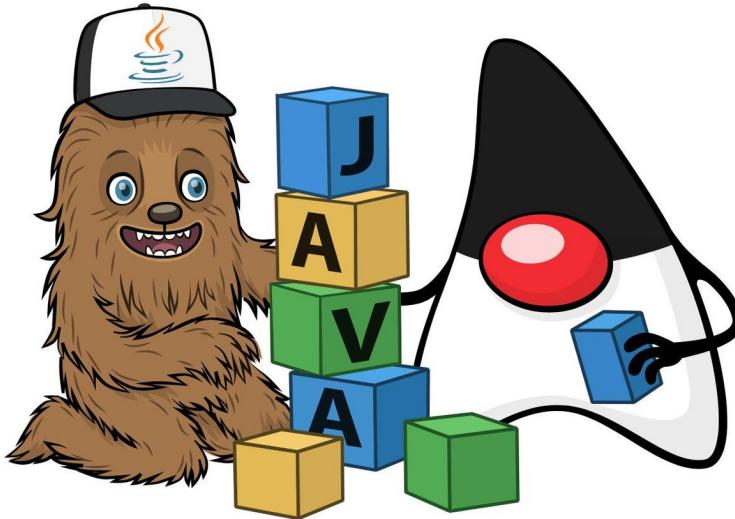


ObjectInputStream

- Se utiliza en conjunto con ObjectOutputStream para leer los objetos que fueron escritos.
- Se debe notar que al leer los objetos tal vez sea necesario realizar un casting sobre los mismos.

Agenda

- ~~Serialización~~
- ~~Interfaz Serializable~~
- ~~Serial Version UID~~
- ~~Modificador transient~~
- ~~Serialización y archivos~~
 - ObjectOutputStream
 - ObjectInputStream
- **JSON**
- Librerías para procesamiento de JSON
 - Jackson
 - Gson



JSON

- JSON (JavaScript Object Notation) es un formato para intercambiar datos liviano, basado en texto e independiente del lenguaje de programación fácil de leer tanto para seres humanos como para las máquinas.
- Puede representar dos tipos estructurados: objetos y matrices.
 - Un objeto es una colección no ordenada de cero o más pares de nombres/valores.
 - Una matriz es una secuencia ordenada de cero o más valores.
- Los valores pueden ser cadenas, números, booleanos, nulos y estos dos tipos estructurados.

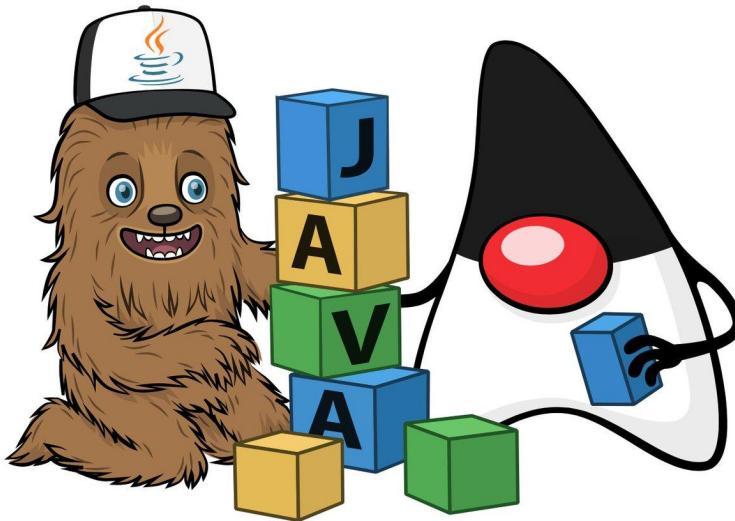
JSON (2)

```
{  
    "nombre": "Homero",  
    "apellido": "Simpson",  
    "edad": 35,  
    "direccion": {  
        "calle": "Av. Colón 1145",  
        "ciudad": "Mar del Plata",  
        "codigoPostal": 7600  
    },  
    "telefonos": [  
        {  
            "tipo": "casa",  
            "numero": "212 555-1234"  
        },  
        {  
            "tipo": "celular",  
            "numero": "646 555-4567"  
        }  
    ]  
}
```

- Ejemplo: representación JSON de un objeto Persona, que tiene valores de cadena para nombre y apellido, un valor numérico para la edad, un valor de objeto que representa el domicilio de la persona y un valor de matriz de objetos de números telefónicos.

Agenda

- ~~Serialización~~
- ~~Interfaz Serializable~~
- ~~Serial Version UID~~
- ~~Modificador transient~~
- ~~Serialización y archivos~~
 - ObjectOutputStream
 - ObjectInputStream
- ~~JSON~~
- **Librerías para procesamiento de JSON**
 - Jackson
 - Gson



Librerías para procesamiento de JSON

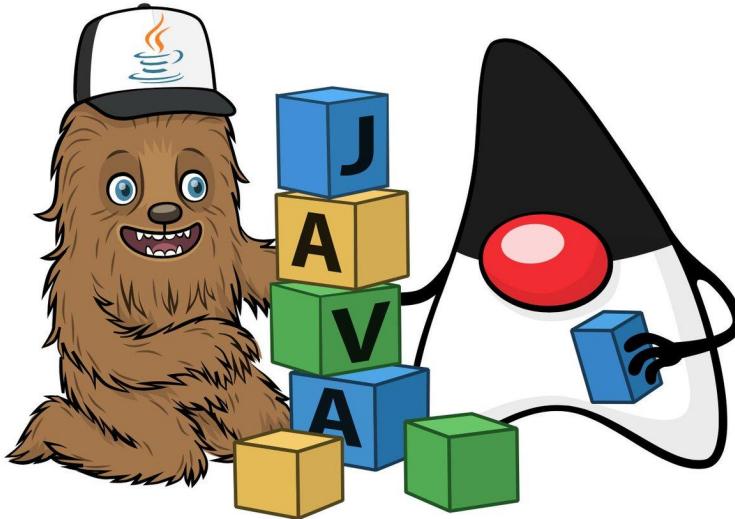
- En la actualidad existen varias librerías para transformar un objeto Java en una cadena JSON. Entre ellas:
- 1) **Jackson:** es una librería de utilidad de Java que nos simplifica el trabajo de serializar (convertir un objeto Java en una cadena de texto con su representación JSON), y deserializar (convertir una cadena de texto con una representación de JSON de un objeto en un objeto real de Java) objetos JSON. Para ello usa básicamente la introspección de manera que si en el objeto JSON tenemos un atributo “name”, para la serialización buscará un método “getName()” y para la deserialización buscará un método “setName(String s)”.

Librerías para procesamiento de JSON (2)

2) **Gson**: el uso de esta librería se basa en el uso de una instancia de la clase Gson. Dicha instancia se puede crear de manera directa (`new Gson()`) para transformaciones sencillas o de forma más compleja con GsonBuilder para añadir distintos comportamientos.

Agenda

- ~~Serialización~~
- ~~Interfaz Serializable~~
- ~~Serial Version UID~~
- ~~Modificador transient~~
- ~~Serialización y archivos~~
 - ObjectOutputStream
 - ObjectInputStream
- ~~JSON~~
- ~~Librerías para procesamiento de JSON~~
 - Jackson
 - Gson



Jackson

- Escribir en archivo:

```
File file = new File("mi_archivo.json");

ObjectMapper mapper = new ObjectMapper();
Persona persona = new Persona();

//Object to JSON in file
mapper.writeValue(file, persona);
```

Jackson (2)

- Leer desde archivo:

```
File file = new File("mi_archivo.json");

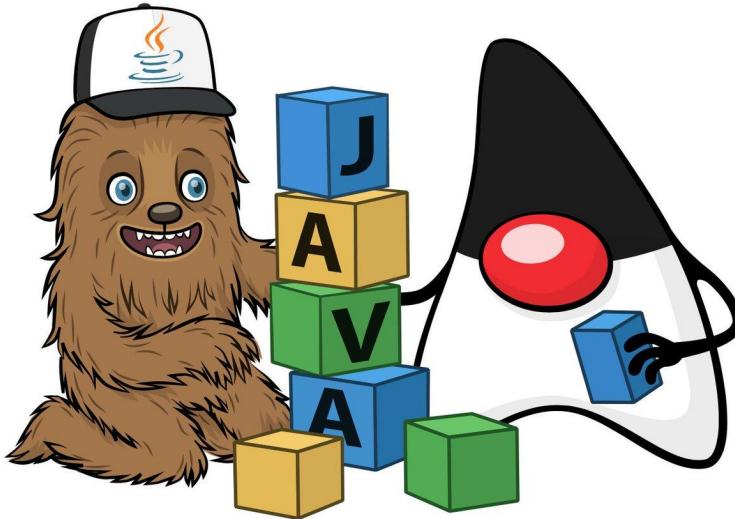
ObjectMapper mapper = new ObjectMapper();

Persona p = mapper.readValue(file, Persona.class);

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

Agenda

- ~~Serialización~~
- ~~Interfaz Serializable~~
- ~~Serial Version UID~~
- ~~Modificador transient~~
- ~~Serialización y archivos~~
 - ObjectOutputStream
 - ObjectInputStream
- ~~JSON~~
- ~~Librerías para procesamiento de JSON~~
 - Jackson
 - Gson



Gson

- Escribir en archivo:

```
File file = new File("mi_archivo.json");
```

```
BufferedWriter bufferedWriter = new BufferedWriter(new FileWriter(file));
```

```
Persona persona = new Persona("Juan", "Gson");
```

```
Gson gson = new Gson();
```

```
gson.toJson(persona, Persona.class, bufferedWriter);
```

Gson (2)

- Leer desde archivo:

```
File file = new File("mi_archivo.json");

BufferedReader bufferedReader = new BufferedReader(new FileReader(file));

Gson gson = new Gson();

Persona persona = gson.fromJson(bufferedReader, Persona.class);

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

Bibliografía oficial

- https://www.tutorialspoint.com/java/java_serialization.htm
- <https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html>
- <https://docs.oracle.com/javase/7/docs/api/java/io/ObjectOutputStream.html>
- <https://docs.oracle.com/javase/7/docs/api/java/io/ObjectInputStream.html>
- <http://jackson.codehaus.org/>
- <https://sites.google.com/site/gson/gson-user-guide>