El futuro digital es de todos

MinTIC

UNIVERSIDAD DE ANTIOQUIA
Facultad de Ingeniería

Mision TIC 2022

Mision TIC 2022

SEMANA 3

INICIAMOS 8:05PM

UNIVERSIDAD DE ANTIOQUIA
1803
Facultad de Ingeniería

Luisa Fernanda Restrepo.

Agenda

- Introducción
- Herencia en Java
- Polimorfismo
- Clase abstracta
- Interfaz

Herencia

Luisa Fernanda Restrepo.

Introducción

- La programación orientada a objetos le permite definir nuevas clases a partir de clases existentes. A esto se le llama **herencia**.

- La herencia es una característica importante y poderosa para **reutilizar software**.

- Suponga que necesita definir clases para modelar círculos, rectángulos y triángulos. Estas clases tienen muchas características comunes.

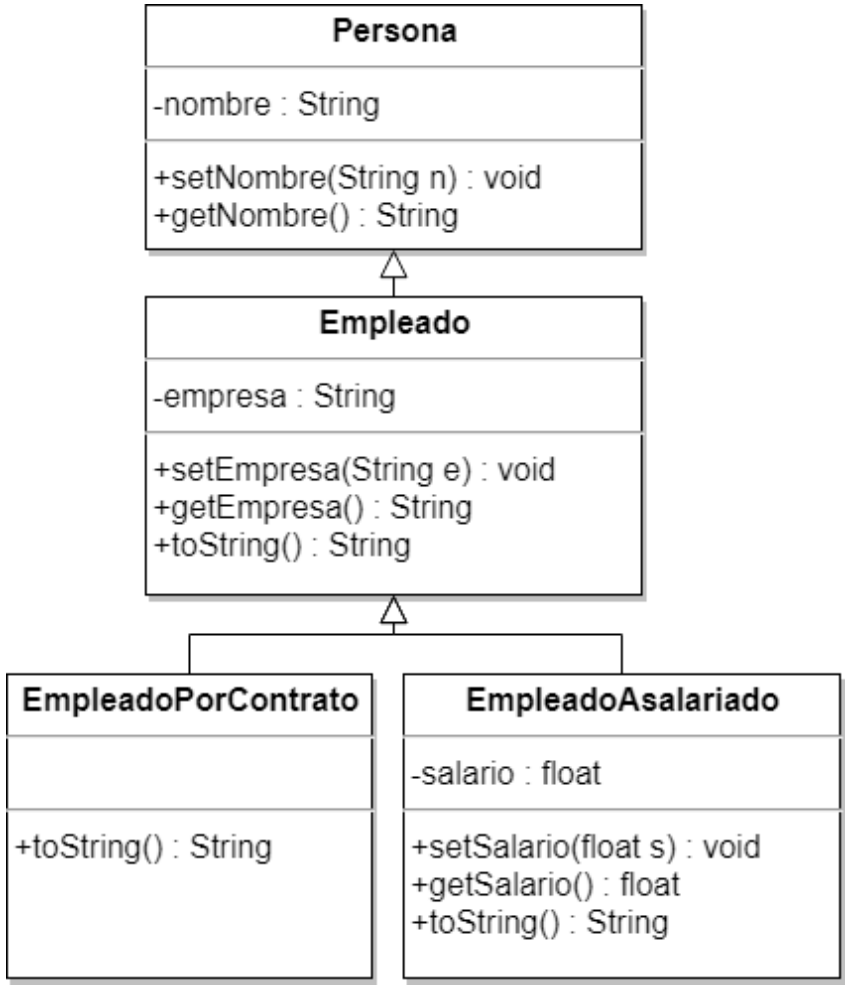
- ¿Cuál es una buena manera de diseñar estas clases para evitar la redundancia y hacer que el sistema sea fácil de comprender y mantener*?
  - La respuesta es utilizar herencia.

**Algunos autores consideran el uso de la herencia peligroso, y sugieren en su lugar, utilizar: (i) interfaces, (ii) delegación, o (iii) mixins.*

A continuación, veamos algunos conceptos claves para poder entender la herencia

Diagrama de clases

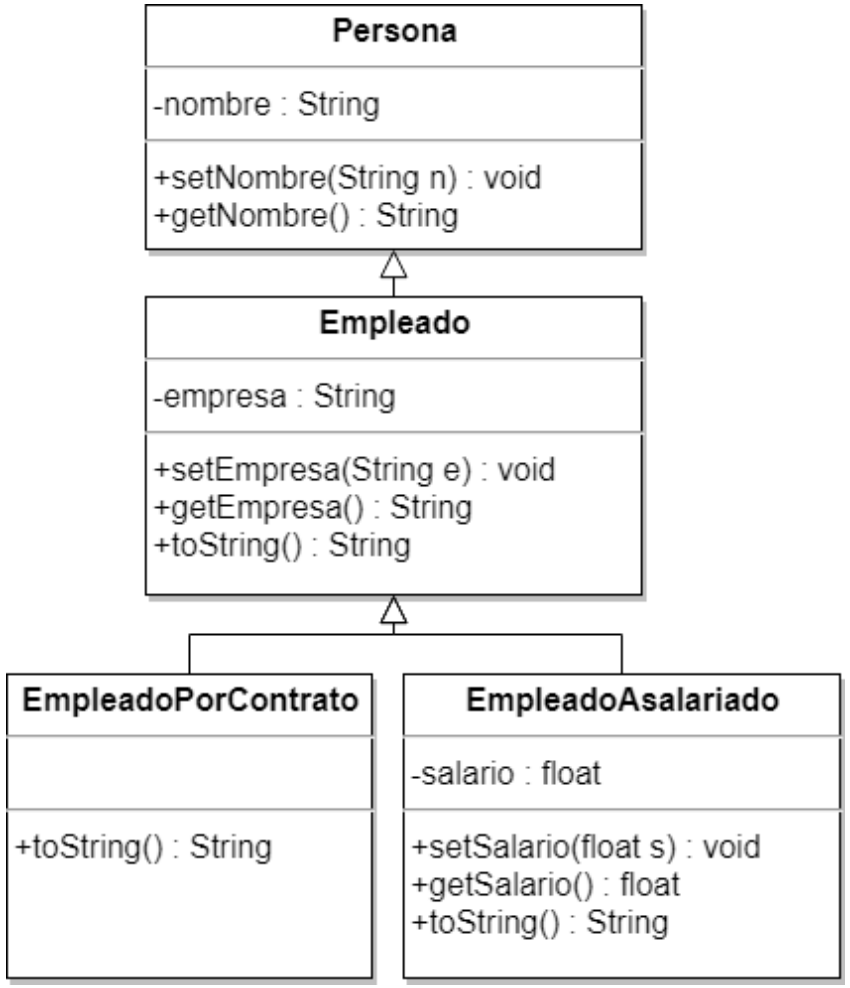
- Para representar una herencia en un diagrama de clases se utiliza una **flecha con punta blanca**, que va desde la clase hija hacia la clase padre.



Superclase

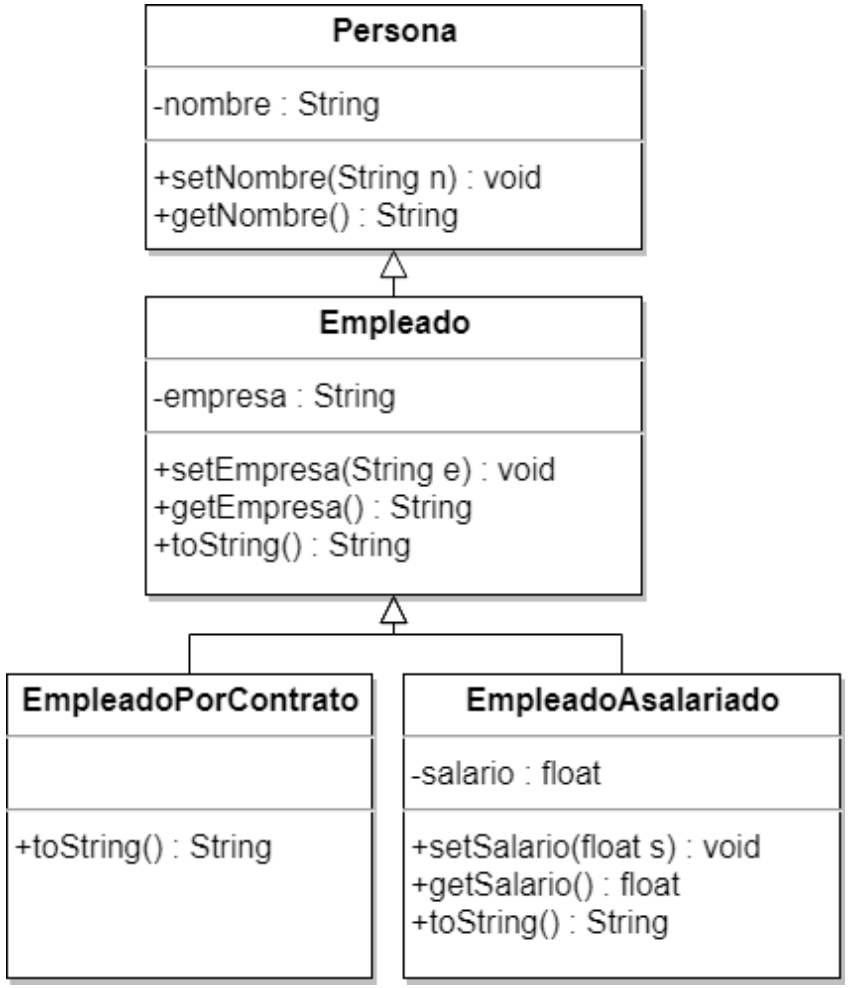
- **Superclase:** también llamada clase **padre** o clase **base**.

- Una superclase **no necesita a las subclases para existir**, y cualquier modificación en cualquiera de las subclases no afecta el comportamiento, ni los elementos de la superclase.



Subclase

- **Subclase:** también llamada clase **hija**.
- Una subclase **hereda** todos los atributos y métodos de la clase padre.
- Sin embargo, dependiendo de la manera en que se definan estos métodos o atributos, y del lenguaje de programación, la subclase **puede tener o no acceso** a ciertos elementos heredados.
- La subclase, también podrá agregar o redefinir elementos heredados.
- Una subclase **requiere que exista** la superclase.

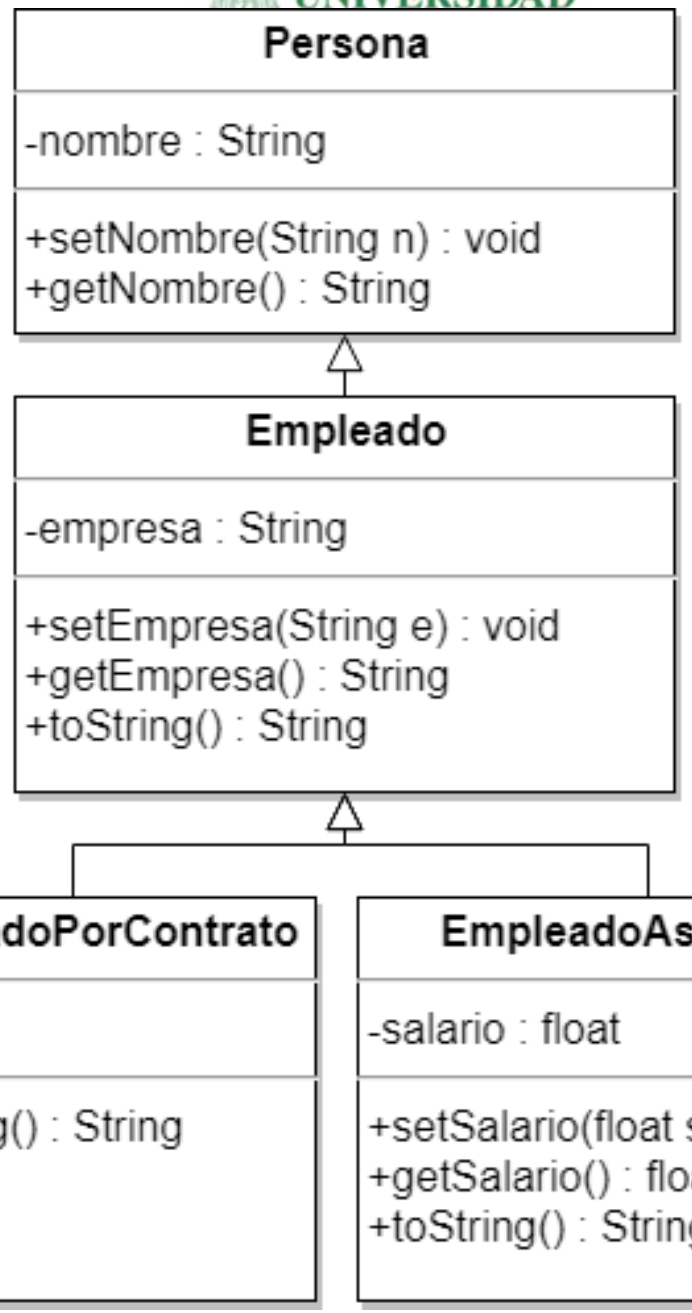


¿Cuáles son clases padre?

Persona y empleado

¿Cuáles son clases hijas?

Empleado, EmpleadoPorContrato y EmpleadoAsalariado



¿Cuáles son los atributos de empleado?

Nombre y empresa

¿Cuáles clases pueden existir independiente de lo que pase con las otras?

Persona

¿Cuáles son los atributos de EmpleadoAsalariado?

Nombre, empresa y salario

¿Cuáles son los métodos de Empleado?

setNombre, getNombre setEmpresa, getEmpresa y toString

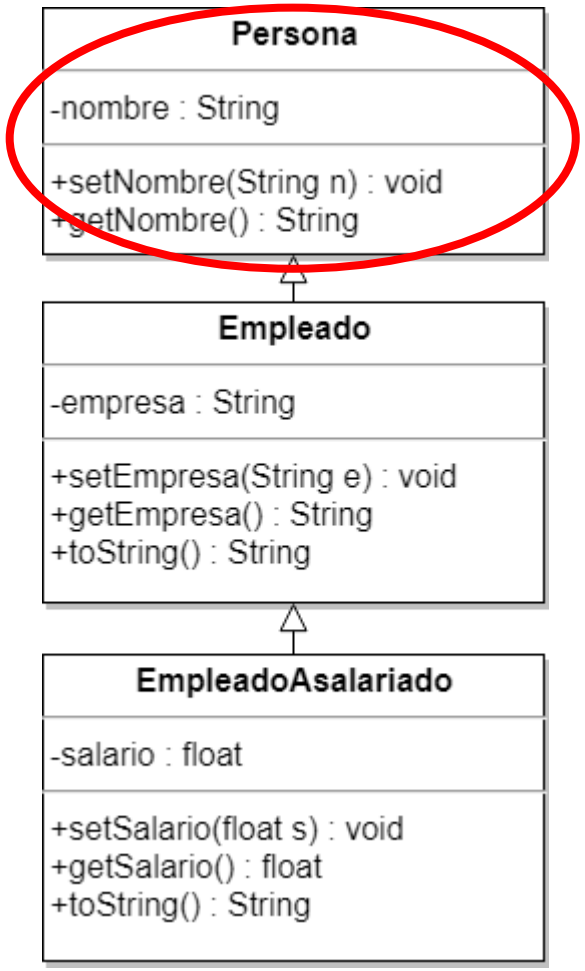
¿Cuál método redefine EmpleadoPorContrato?

toString

A continuación, veamos como implementar herencia en Java

Clase Persona

```java
public class Persona
{
    private String nombre;

    public void setNombre(String n){
        this.nombre = n;
    }

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



Clase Empleado

Subclase

Superclase

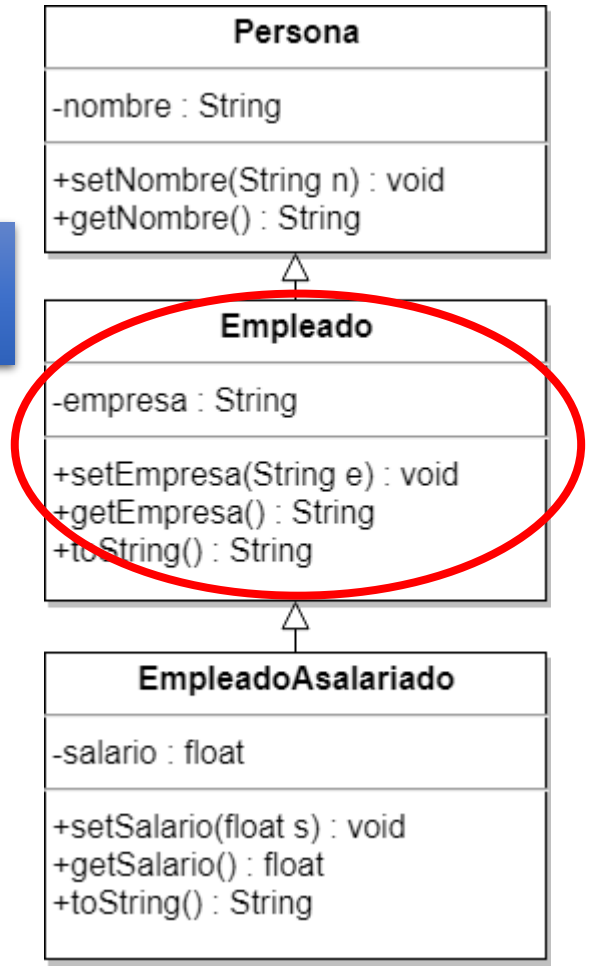
```
public class Empleado extends Persona
{
    private String empresa;

    public void setEmpresa(String e){
        this.empresa = e;
    }

    public String getEmpresa(){
        return this.empresa;
    }

    @Override
    public String toString(){
        String texto = "Empleado{" + "nombre=" + getNombre();
        texto = texto + " empresa=" + this.empresa + "}";
        return texto;
    }
}
```

Forma de definir herencia en Java

¿Por qué se accede mediante método?

Por ser privado

¿?



Main – I

¿Qué imprime?

```
public class PrincipalHerencia
{
    public static void main(String[] args){
        Empleado e1 = new Empleado();
        e1.setEmpresa("Mc Donals");
        e1.setNombre("Luis");
        System.out.println(e1.getEmpresa());
        System.out.println(e1.toString());
    }
}
```

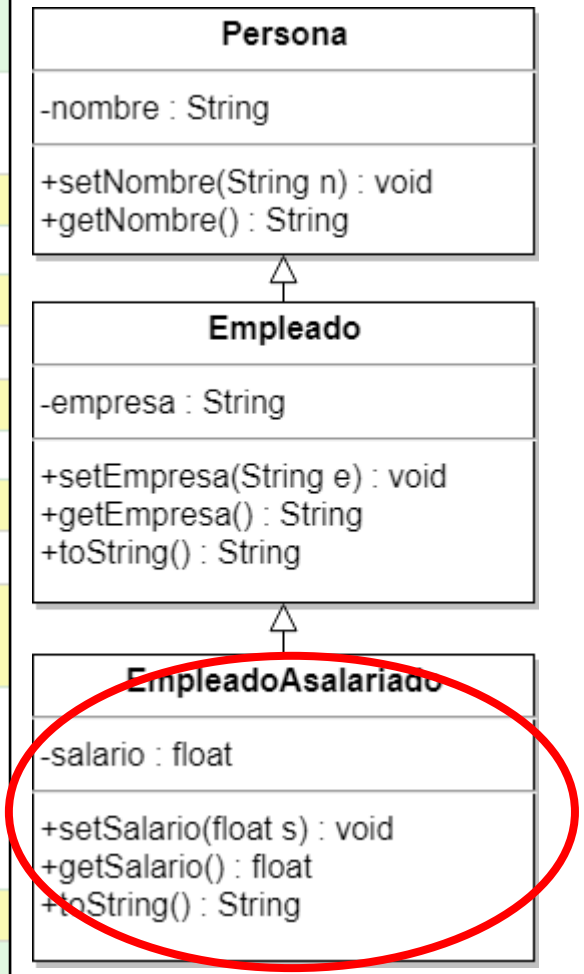
Opciones

```
Mc Donals
Empleado{nombre=Luis empresa=Mc Donals}
```

Clase EmpleadoAsalariado

```
public class EmpleadoAsalariado extends Empleado
{
    private float salario;

    public float getSalario(){
        return this.salario;
    }

    public void setSalario(float s){
        this.salario = s;
    }

    @Override
    public String toString(){
        String texto = "EmpleadoAsalariado{" + "nombre=" + getNombre();
        texto = texto + " empresa=" + getEmpresa();
        texto = texto + " salario=" + this.salario + "}";
        return texto;
    }
```

En Java, solo se puede extender de una sola clase, no existe herencia múltiple



En Java, esta anotación no es necesaria, pero es muy recomendada utilizarla

Esta anotación sirve para indicar que se esta "sobreescribiendo" un método "igual" que viene de alguno de sus ancestros

Main – II

¿Qué imprime?

```
public class PrincipalHerencia
{
    public static void main(String[] args){
        EmpleadoAsalariado ea1 = new EmpleadoAsalariado();
        ea1.setEmpresa("Apple");
        ea1.setNombre("Maria");
        ea1.setSalario(100);
        System.out.println(ea1.getNombre());
        System.out.println(ea1.toString());
    }
}
```

Opciones

```
Maria
EmpleadoAsalariado{nombre=Maria empresa=Apple salario=100.0}
```

Override Empleado toString

```java
public class Empleado extends Persona
{
    private String empresa;

    public void setEmpresa(String e){
        this.empresa = e;
    }

    public String getEmpresa(){
        return this.empresa;
    }

    @Override
    public String toString(){
        String texto = "Empleado{" + "nombre=" + getNombre();
        texto = texto + " empresa=" + this.empresa + "}";
        return texto;
    }
}
```

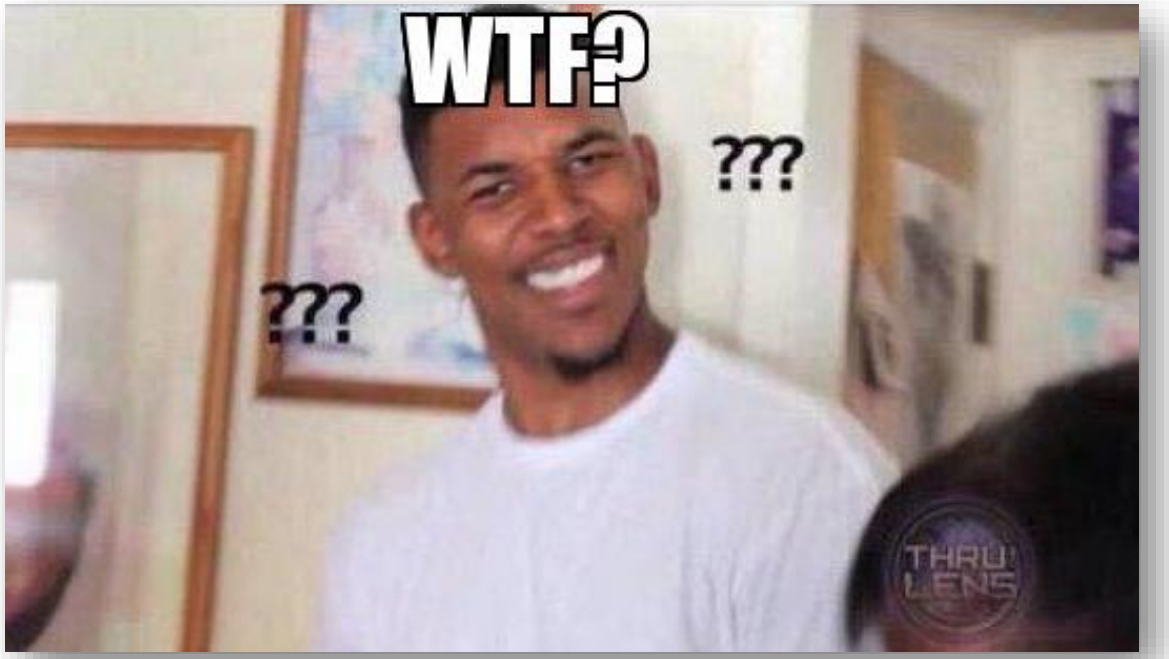
En Java, todas las clases heredan de Object

Estamos sobreescribiendo el método toString de la clase Object

¿De donde viene? ¿Qué método de que clase estamos "sobreescribiendo"?

Pregunta

- ¿Si en Java no existe la herencia múltiple, como es posible que una clase pueda heredar de otra, pero a su vez todas hereden de la clase Object?

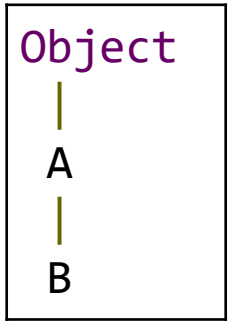


Respuesta

- **Todas las clases heredan de Object**, ya sea de manera implícita o explicita. Las siguientes definiciones de clases son equivalentes.

```
public class A extends Object {
public class A {
```

- Luego, que pasa si defino una clase que herede de la clase A.




```
public class B extends A {
```

- B heredaría de A, pero A a su vez hereda de Object, por lo tanto B hereda y puede acceder a los métodos de Object.

Constructor sin parámetros

- **Por defecto, Java define el constructor sin parámetros** (vacío). Las siguiente definiciones son equivalentes:

```
public class A { }
public class A { A(){} }
```

- Pero OJO, si se define un constructor (que tenga al menos un parámetro), el constructor sin parámetros **NO SERÁ definido automáticamente** (tocaría definirlo manualmente).

```
public class Circulo {
    private double radio;

    Circulo(double r){
        this.radio=r;
    }

    public static void main(String[] args) {
        Circulo c1 = new Circulo();
    }
}
```

```
constructor Circulo in class Circulo cannot be applied to
given types;
  required: double
  found: no arguments
  reason: actual and formal argument lists differ in length
```

The compiler automatically provides a no-argument, default constructor for any class without constructors.

Super

- **super** es una palabra clave que nos permite acceder a métodos y/o atributos de una superclase desde una subclase.

```
super.method(parameters);
super.attribute
```

- *The form super.Identifier refers to the field named Identifier of the current object, but* ***with the current object viewed as an instance of the superclass of the current class****.*

Según el párrafo anterior, ¿es o no posible utilizar la palabra super dentro de definiciones de métodos static?

R:// No

Constructor con super

- Un constructor puede invocar a: (i) **otro constructor sobrecargado** (hermano) o (ii) **al constructor de su superclase**.
- Si ninguno se invoca explícitamente, el compilador automáticamente agrega **super()** como la primera instrucción en el constructor.

```
public ClassName() {
  // some statements
}
```

Equivalent

```
public ClassName() {
  super();
  // some statements
}
```

```
public ClassName(parameters) {
  // some statements
}
```

Equivalent

```
public ClassName(parameters) {
  super();
  // some statements
}
```

¿Qué imprime?

```
public class Faculty extends Employee {
    public static void main(String[] args) {new Faculty();}

    public Faculty() {System.out.println("(*) Performs Faculty's tasks");}
}
class Employee extends Person {
    public Employee() {
        this("(*) Invoke Employee's overloaded constructor");
        System.out.println("(*) Performs Employee's tasks ");
    }
    public Employee(String s) {
        System.out.println(s);
    }
}
class Person {
    public Person() {System.out.println("(*) Performs Person's tasks");}
}
```

Agrega un super()

Invoca el constructor “hermano”

Agrega un super()

BlueJ: Ventana de Terminal - taller-01

Opciones

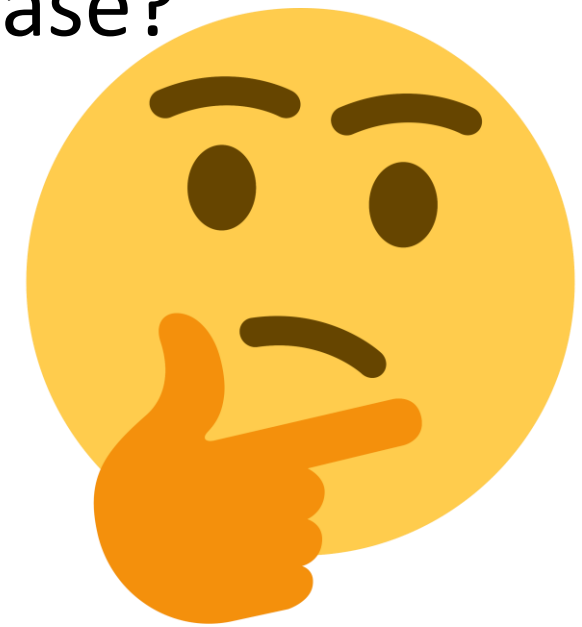
```
(*) Performs Person's tasks
(*) Invoke Employee's overloaded constructor
(*) Performs Employee's tasks
(*) Performs Faculty's tasks
```

¿Queda faltando algo?

Agregaría otro super() que invocaría el constructor vació de Object

¿En Java, se podría ejecutar un constructor de una subclase, y evitar que se haga un llamado a algún constructor del padre de esa subclase?

R/ = NO.
Siempre se invocará por lo menos un constructor de cada ancestro de la subclase



¿Según el ejemplo anterior (y en general cuando hay herencia), cuales son los constructores que finalizan primero su ejecución completa?

R/ = Se finaliza primero la ejecución de algún constructor del ancestro mas antiguo, luego el que le sigue, y así sucesivamente hasta que llegue a la clase hija.

Ejercicio – super constructores

- Suponga que tiene el siguiente código, ¿Cómo se debe definir el constructor de Cuadrado para el programa se pueda compilar?.

```
public class Figura {
    private String nombre;

    Figura(String n){
        this.nombre=n;
    }
}
```

```
public class Cuadrado extends Figura {
    private float lado;

    Cuadrado(???){
        ???
    }

}
```

Solución

Se llama el constructor del padre. Ese llamado siempre se debe hacer en la primera línea.

En este caso el constructor de Cuadrado debe llamar obligatoriamente el Constructor de Figura enviándole un parámetro. Ya que si se omite, el programa trataría de llamar el Constructor vacío de Figura, el cual no existe.

Recomendación: siempre cree el constructor vacío de una clase que usted sospecha que se utilizará como superclase

```
public class Cuadrado extends Figura {
    private float lado;
    Cuadrado(String nombre, float lado){
        super(nombre);
        this.lado=lado;
    }
}

class Figura {
    private String nombre;
    Figura(String n){
        this.nombre=n;
    }
}
```

Super para invocar métodos

```
public class Cuadrado extends Figura {
    private float lado;
    public Cuadrado(String nombre, float lado){
        super(nombre);
        this.lado=lado;
    }
    public void area(){
        super.area();
        System.out.println("Area: "+(this.lado*this.lado));
    }
    public static void main(String[] args){
        Cuadrado c1 = new Cuadrado("C1",4);
        c1.area();
    }
}
class Figura {
    private String nombre;
    public Figura(String n){
        this.nombre=n;
    }
}
```

¿Qué imprime?

Error, no existe un método “area” definido en ninguno de los ancestros

```
cannot find symbol - method area()
```

¿Qué imprime?

```java
public class Cuadrado extends Figura {
    private float lado;
    public Cuadrado(String nombre, float lado){
        super(nombre);
        this.lado=lado;
    }
    @Override
    public void area(){
        super.area();
        System.out.println("Area: "+(this.lado*this.lado));
    }
    public static void main(String[] args){
        Cuadrado c1 = new Cuadrado("C1",4);
        c1.area();
    }
}
class Figura {
    private String nombre;
    public Figura(String n){
        this.nombre=n;
    }
    public void area(){
        System.out.println("No se puede calcular, solo el de las hijas");
    }
}
```

Invoca el método “area” del padre

Opciones

```
No se puede calcular, solo el de las hijas
Area: 16.0
```

```java
public class Vehiculo
{
    private String nombre;
    public void info(){
        System.out.println("Mensaje desde Vehiculo");
    }
    public static void main(String[] args){
        Aereo a1 = new Aereo(); a1.info();
        Moto m1 = new Moto(); m1.info();
    }
}
class Terrestre extends Vehiculo{}
class Aereo extends Vehiculo{
    @Override
    public void info(){
        super.info();
        System.out.println("Mensaje desde Aereo");
    }
}
class Moto extends Terrestre{
    @Override
    public void info(){
        super.info();
        System.out.println("Mensaje desde Moto");
    }
}
```

¿Qué imprime?

Opciones

```
Mensaje desde Vehiculo
Mensaje desde Aereo
Mensaje desde Vehiculo
Mensaje desde Moto
```

El super siempre trata de buscar la definición del método en el ancestro mas cercano (“el padre”); si no la encuentra, sigue con “el abuelo”, y así sucesivamente.

Redefinición de atributos

- En Java, es posible redefinir atributos (previamente definidos en alguno de los antecesores.
- Es un practica poco común, ya que "enreda" mucho el seguimiento del código.

Se reserva un espacio en memoria distinto, para cada definición

c1

Vehiculo.velocidad	double	5.0
velocidad	double	25000.0

¿Qué imprime?

```
public class Vehiculo
{
    public double velocidad = 5.0;
    public void info(){
        System.out.println("Vel:"+ this.velocidad);
    }
    public static void main(String[] args){
        Cohete c1 = new Cohete(); c1.info();
    }
}

class Aereo extends Vehiculo {
}

class Cohete extends Aereo{
    public double velocidad = 25000.0;
    @Override
    public void info(){
        super.info();
        System.out.println("Vel:"+ this.velocidad);
    }
}
```

Opciones

```
Vel:5.0
Vel:25000.0
```

¿Qué imprime?

```
public class Vehiculo
{
    protected double velocidad = 5.0;
    public void info(){
        System.out.println("Vel:"+ this.velocidad);
    }
    public static void main(String[] args){
        Cohete c1 = new Cohete(); c1.info();
    }
}

class Aereo extends Vehiculo {
}

class Cohete extends Aereo{
    protected double velocidad = 25000.0;
    @Override
    public void info(){
        System.out.println("Vel:"+ this.velocidad);
        System.out.println("Vel:"+ super.velocidad);
    }
}
```

Opciones

```
Vel:25000.0
Vel:5.0
```

Debe ser de visibilidad publica o protegida para poder funcionar

Ejercicio

- Cree la clase Vehiculo con los atributos nombre y precio (privados).
  - Cree el método getPrecio(), que retorne el precio del vehiculo.
- Cree la clase Maritimo (que herede de Vehiculo) con el atributo de clase IVA = 20 (considérelo un porcentaje).
  - Cree el método getPrecio(), que retorne el precio de la instancia, más el IVA.
- Cree la clase Buque (que herede de Maritimo)
  - Cree el método getPrecio(), que retorne la suma del getPrecio() del padre, más 40.
- Cree una clase principal con un método main, luego cree 4 vehículos (2 marítimos y 2 buques / invéntese valores de nombre y precios). En el proceso cree los constructores necesarios.
- Imprima los precios finales de cada instancia.

Polimorfismo

Luisa Fernanda Restrepo.

Introducción

- Los 3 pilares de la POO son:

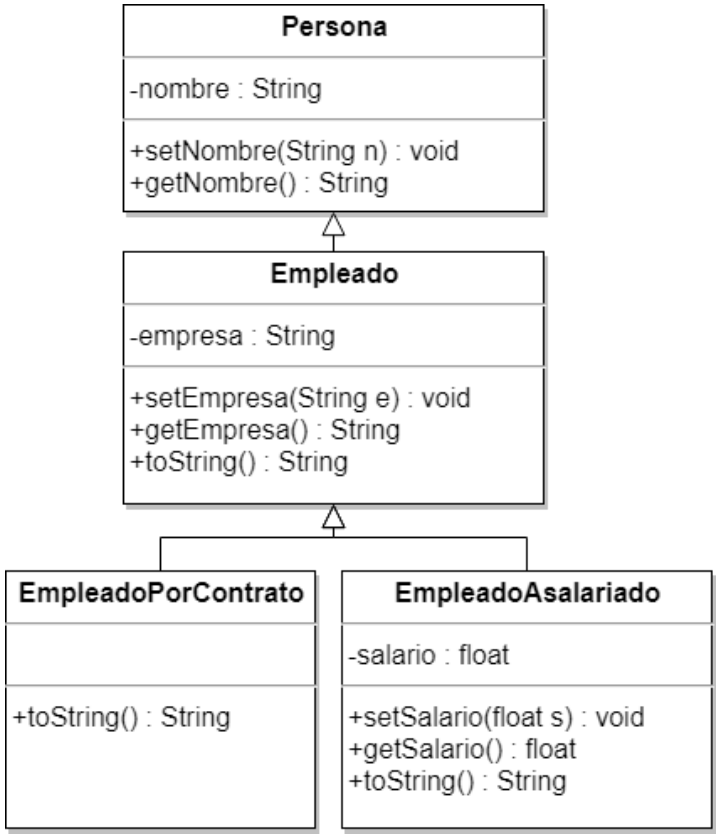
Encapsulamiento ***Herencia*** ***Polimorfismo***

- Los dos primeros pilares ya los vimos en clases pasadas. Ahora veamos polimorfismo.

Polimorfismo

Polimorfismo: que posee varias formas.

- La relación de herencia nos permite que una **subclase herede las características de su superclase**, y además que defina sus propias nuevas características adicionales.
- En este sentido, una subclase es una especialización de su superclase; lo cual implica que, **cada instancia de una subclase es también una instancia de su superclase**. Pero no al revés**.**
- **Por ejemplo:** una instancia de *EmpleadoAsalariado*, es también una instancia de *Empleado* y a su vez una instancia de *Persona*. Pero una *Persona* no es una instancia de *Empleado*.

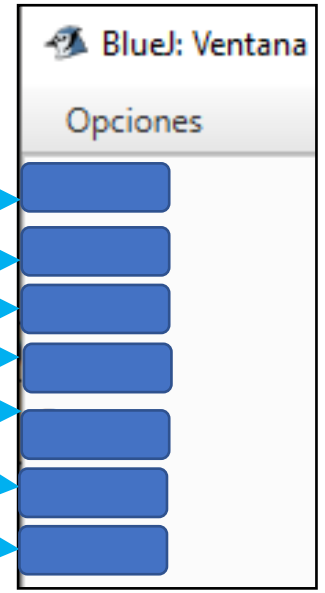


instanceof

- Con la palabra reservada ***instanceof*** podemos verificar si un objeto es instancia de alguna clase en particular.
- Veamos el siguiente ejemplo:

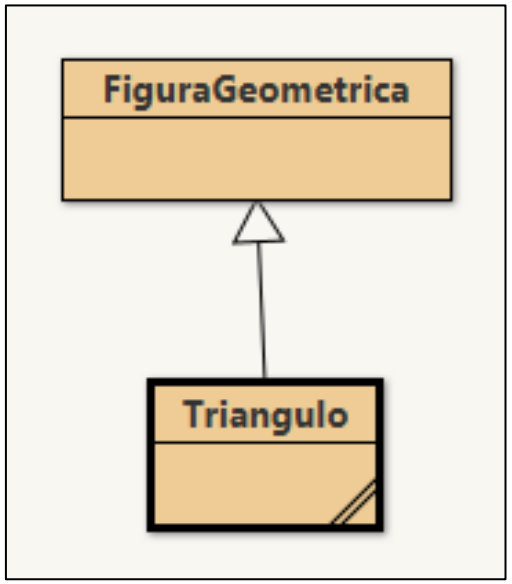
¿Qué imprime?

```
public class Polimorfismo
{
    public static void main(String[] args){
        EmpleadoAsalariado ea1 = new EmpleadoAsalariado();
        Persona p1 = new Persona();
        System.out.println(ea1 instanceof Persona);
        System.out.println(ea1 instanceof Empleado);
        System.out.println(ea1 instanceof EmpleadoAsalariado);
        System.out.println(ea1 instanceof Object);
        System.out.println(p1 instanceof Empleado);
        System.out.println(p1 instanceof EmpleadoAsalariado);
        System.out.println(p1 instanceof Object);
    }
}
```



Ejemplo

```java
public class FiguraGeometrica
{
    private int lado;

    public FiguraGeometrica(int lado){
    this.lado = lado;
    }

    public double calcularFGArea(){
        double a;
        a = Math.pow(lado,2);
        return a;
    }
}
```



Codifíquelo

```java
public class Triangulo extends FiguraGeometrica
{
    public Triangulo(int lado){
    super(lado);
    }

    public double calcularFGArea(){
        double area;
        area = Math.sqrt(3.0)/4.0*super.calcularFGArea();
        return area;
    }
}
```

Ejemplo

```
public class Principal
{
    public static void main(String [] args)
    {
        FiguraGeometrica figura1;
        FiguraGeometrica figura2;

        double area1;
        figura1 = new FiguraGeometrica(5);
        area1 = figura1.calcularFGArea();
        System.out.println("Area figura 1: "+area1);

    }
}
```

Esto coincide con lo que hemos visto hasta el momento, se crea una variable del tipo de clase que hace referencia a una instancia de la misma clase

Codifíquelo

¿Se puede crear una variable de tipo de clase que haga referencia a una instancia de otra clase?

Ejemplo

```
public class Principal
{
    public static void main(String [] args)
    {
        FiguraGeometrica figura1;
        FiguraGeometrica figura2;

        double area1;
        figura1 = new FiguraGeometrica(5);
        area1 = figura1.calcularFGArea();
        System.out.println("Area figura 1: "+area1);

        double area2;
        figura2 = new Triangulo(2);
        area2 = figura2.calcularFGArea();
        System.out.println("Area figura 2: "+area2);
```

R:// Si, siempre y cuando la instancia sea subclase de la superclase asociada a la variable

Codifíquelo

¿Qué imprime?

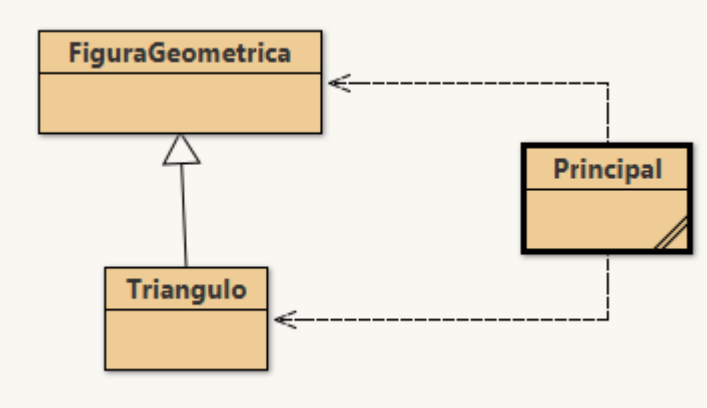
```
Options
Area figura 1: 25.0
Area figura 2: 1.7320508075688772
```

El tipo de objeto que invoca el método calcularFGArea() determina que método calcularFGArea() se llama. En este caso, para la figura2 se invoca el método de la clase Triangulo

Polimorfismo

Polimorfismo: significa que una variable de un superclase puede referirse a un objeto de una subclase.

Este es un ejemplo de polimorfismo. Las variables figura1 y figura2 podrían hacer referencia a un objeto FiguraGeometrica o un objeto Triangulo. En tiempo de compilación, no se puede determinar a qué tipo de objeto harán referencia. Sin embargo, en tiempo de ejecución cuando el objeto invoca el método calcularFGArea(), se determina el tipo de objeto y se llama al método calcularFGArea() apropiado.

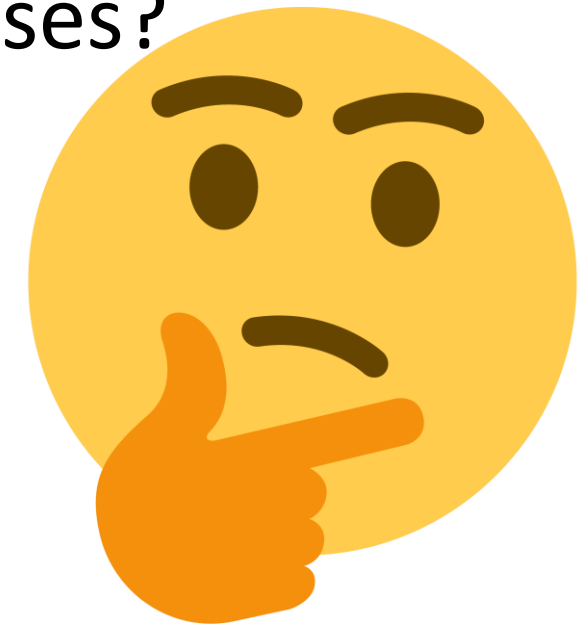


Según lo explicado anteriormente, ¿Se podría agregar el siguiente segmento de código a la clase principal?

```
Triangulo figura3;
 figura3 = new FiguraGeometrica(6);
```

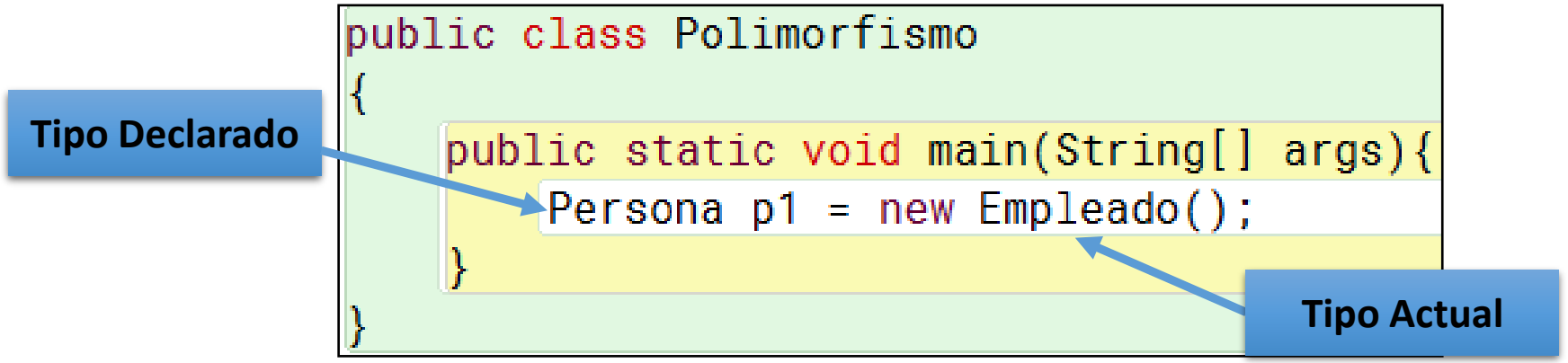
R:// No, por que una variable de referencia de una subclase no puede hacer referencia a un objeto de su superclase

¿Qué implicaciones tiene que una instancia de una clase pueda ser también instancia de sus superclases?



Tipo actual y tipo declarado - II

- Siguiendo el ejemplo de la clase pasada, podríamos hacer lo siguiente:

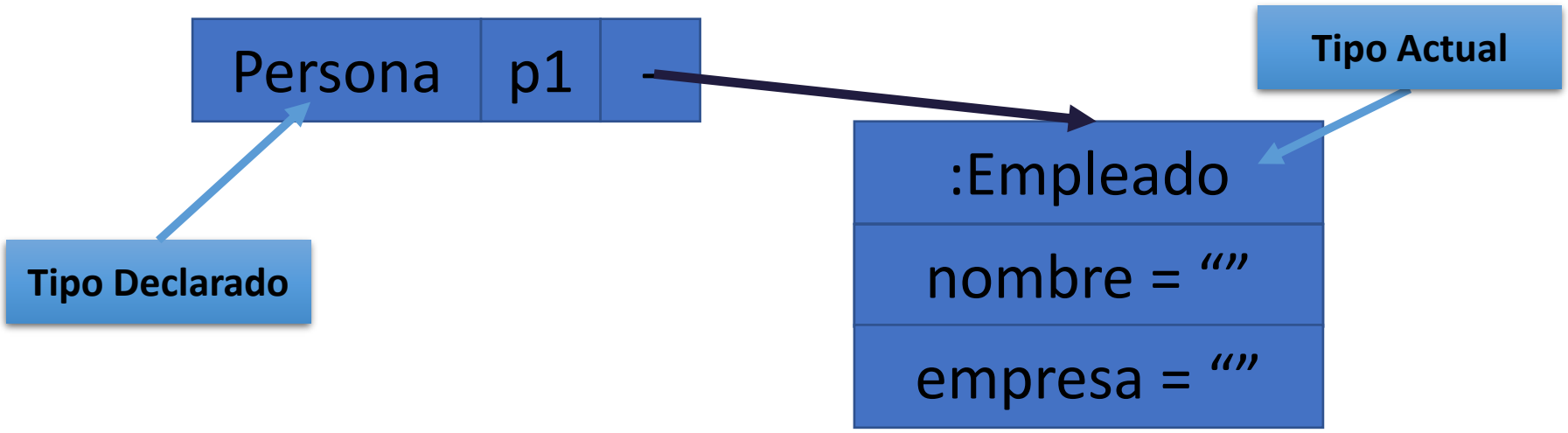


- **Tipo declarado:** es el tipo del cual se declara la **variable**, en este caso “p1” es de tipo declarado “Persona”.
- **Tipo actual:** es el tipo del **objeto referenciado**. En este caso “p1” esta haciendo referencia a un objeto tipo “Empleado”. Por lo tanto, “p1” es de tipo actual “Empleado”.

Tipo actual y tipo declarado – III

- Veamos como se almacena el memoria el siguiente código:

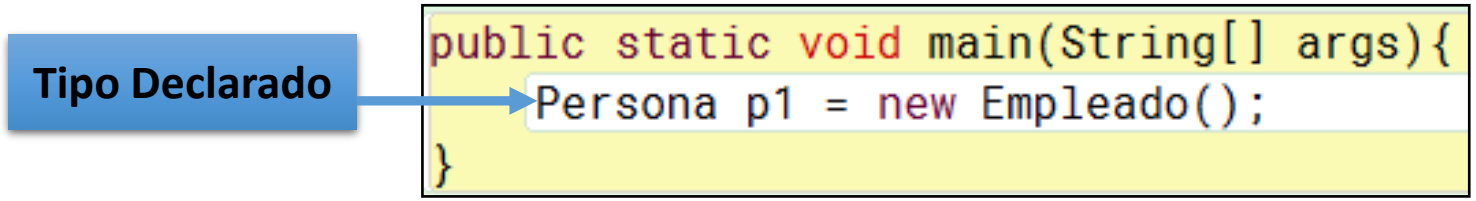
```
public class Polimorfismo
{
    public static void main(String[] args){
        Persona p1 = new Empleado();
    }
}
```



Tipo actual y tipo declarado – IV

¿Para que sirve el Tipo Declarado?

- El Tipo Declarado (variable) me permite acceder a los métodos/atributos del objeto referenciado. Dependiendo del Tipo Declarado, algunas veces podré acceder a todos los métodos/atributos del objeto referenciado; en otros casos, solo podré acceder a algunos de ellos.



- En el ejemplo, como p1 es de "Tipo Declarado: Persona", desde esa variable solo se podrá acceder a los métodos definidos en la clase Persona.
  - Esto quiere decir, que desde la variable p1 podré acceder a los métodos:
    getNombre(), setNombre()
  - Esto también quiere decir, que desde la variable p1 **no** podré acceder a los métodos:
    getEmpresa(), setEmpresa()

Conversiones

- Cuando a una variable se le asigna una nueva instancia que tiene un “Tipo Actual” diferente al “Tipo Declarado” de la variable, se habla de una conversión.

- A cualquier Tipo Declarado (variable), no se le puede asignar: (i) cualquier Tipo Actual (objeto o instancia nueva); (ii) cualquier variable existente que tenga cualquier Tipo Declarado.

- Cuando a una variable se le asigna otra variable ya existente, y sus Tipos Declarados no coinciden (son diferentes), probablemente se hable de una conversión.

- Existen 2 tipos de conversiones que discutiremos a continuación:
  - Conversión implícita.
  - Conversión explicita.

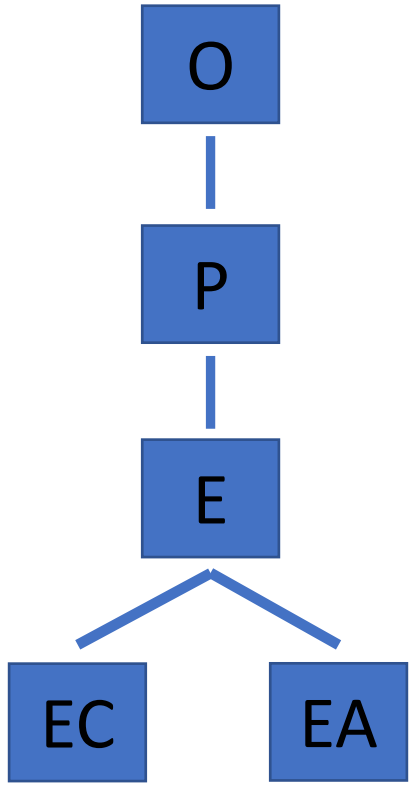
Conversión Implícita – Caso 1 (new al lado derecho)

- En el primer caso de conversión implícita veamos que pasa cuando se hace un **"new" al lado derecho de la asignación**.
- En este caso, la conversión implícita es valida cuando se define una **variable con "Tipo Declarado" que es ancestro del "Tipo Actual" del objeto que se referencia** (del objeto que se está creando).

Veamos un ejemplo:

Object o = new Empleado();

- El código anterior es valido, ya que el Tipo Declarado (Object), es ancestro del Tipo Actual (Empleado).
- El código anterior también es valido, ya que una instancia de *Empleado*, también es una instancia de un *Object*.



Se "convierte" un Empleado en Object

Solo a toString(), y a los otros métodos que defina Object

¿A qué métodos (del objeto Empleado) podríamos acceder con la variable "o"?

Ejemplo 1

¿Cuáles conversiones son invalidas?

```
public class Polimorfismo
{
    public static void main(String[] args){
        Persona p1 = new Object();
        Persona p2 = new EmpleadoAsalariado();
        Empleado e1 = new Persona();
        Empleado e2 = new EmpleadoAsalariado();
        Object o1 = new EmpleadoAsalariado();
        EmpleadoAsalariado ea1 = new EmpleadoAsalariado();
    }
}
```

Conversión explicita – único caso (variable al lado derecho)

```
1 public static void main(String[] args){
2     Empleado e1 = new EmpleadoAsalariado();
3     EmpleadoAsalariado ea1 = new EmpleadoAsalariado();
4     ea1 = e1; //FALLA
5     ea1 = (EmpleadoAsalariado) e1; //NO FALLA
6 }
```

La conversión explicita se identifica fácilmente, por que se define mediante el uso de paréntesis el cambio al Tipo Declarado de la variable a asignar

- El la línea 4 del código anterior, vemos que estamos tratando de hacer una conversión implícita invalida. Esto se debe a que el Tipo Declarado de ea1 (EmpleadoAsalariado), no es ancestro del Tipo declarado de e1 (Empleado).
- Sin embargo, vemos que e1, realmente apunta a un Tipo Actual (EmpleadoAsalaradio), el cual es totalmente valido para que se asignase en ea1.
- En estos casos lo que se puede hacer es una **conversión explicita**. En la conversión explicita se cambia **EXPLICITAMENTE** el Tipo Declarado de la variable a asignar, por un tipo valido para la asignación.
- Al cambiar el Tipo Declarado de e1 de (Empleado) a (EmpleadoAsalariado) -> línea 5, ahora si es valida la asignación.

Ejemplo 3

```
public class Polimorfismo
{
    public static void main(String[] args){
        Object o1 = new Empleado();
        System.out.print(o1.getEmpresa());
    }
}
```

¿Algún problema?

o1 es de Tipo Declarado Object, por lo tanto no puede acceder al método getEmpresa()

Ligadura dinámica

- La **ligadura dinámica** se da cuando existe **sobreescritura** de métodos (redefinición de métodos).

- Recordemos que un método puede ser redefinido (sobreescrito) en varias clases a lo largo de la cadena de herencia.

- Ahora, la JVM debe decidir cual de todos esos métodos redefinidos se invoca **en tiempo de ejecución**.

¿Cuál invocar?

- Cuando hay sobreescritura, la JVM escogerá para invocar aquel método que pertenezca a la clase del **Tipo Actual** de la instancia en ejecución. Y si esa clase no tiene definido ese método, entonces buscará en el ancestro mas cercano, y así sucesivamente.

Ejemplo – ligadura dinámica

Veamos el siguiente ejemplo:

- Recordemos, un método puede ser definido en una superclase y reemplazado en su subclase.
- Por ejemplo, el método toString() se define en la clase Object y se reemplaza (sobreescribe) en la clase Empleado.

¿En el siguiente caso, que se imprime y cuales métodos se invocan?

```
public static void main(String[] args){
    Object o1 = new Object();
    System.out.println(o1.toString());
    Empleado e1 = new Empleado();
    System.out.println(e1.toString());
    Object o2 = new Empleado();
    System.out.println(o2.toString());
}
```

Opciones

```
java.lang.Object@152a795
Empleado{nombre=null empresa=null}
Empleado{nombre=null empresa=null}
```

Método toString de la clase Object

Método toString de la clase Empleado

Método toString de la clase Empleado (se basa en el Tipo Actual // Ligadura dinámica)

Repasemos

¿Cuál método se invoca?

```
public class Polimorfismo
{
    public static void main(String[] args){
        Object o = new Empleado();
        System.out.println(o.toString());
    }
}
```

Respuesta

- Aquel que corresponda al "Tipo Actual". El de Empleado.
- **Recordar:** en caso de que la clase correspondiente al Tipo Actual no tenga método toString(), seguirá buscando en la superclase inmediata.

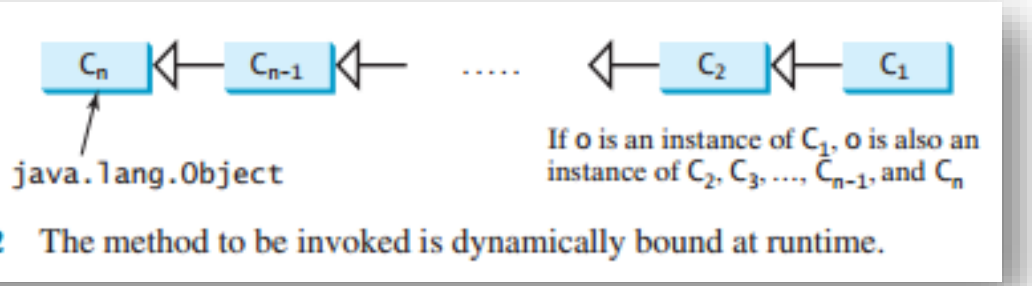


FIGURE 11.2 The method to be invoked is dynamically bound at runtime.

Pregunta

- ¿Por qué en el ejemplo anterior, en ejecución se imprimía la función toString() de Empleado al acceder desde "o", pero no se puede acceder directamente desde "o", a la función getEmpresa()?

```
public class Polimorfismo
{
    public static void main(String[] args){
        Object o = new Empleado();
        System.out.println(o.toString());
        System.out.println(o.getEmpresa());
    }
}
```

Object posee método toString, por lo tanto no sale error al compilar. Sin embargo, Object no tiene método getEmpresa, por lo tanto sale error al tratar de compilar.

Una cosa es que el compilador de Java trate de hacer coincidir el llamado de un método con su definición, y otra que el JVM decida en tiempo de ejecución cual método ejecutar.

Ligadura estática

Se da cuando:

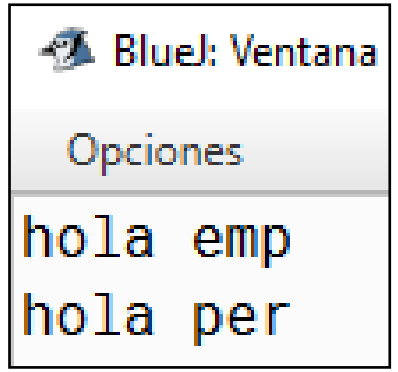
- Se trabaja con métodos estáticos.
- Se trabaja con atributos.

- En la ligadura estática nos enfocamos en verificar cual es el **Tipo Declarado**. Y con base en ese Tipo Declarado, determinamos que método se ejecutará.

Ejemplo – Ligadura estática

¿Qué imprime?

```
public class Polimorfismo
{
    public static void main(String[] args){
        Persona p1;
        Empleado e1 = new Empleado();
        p1=e1;
        e1.hola();
        p1.hola();
    }
}
class Empleado extends Persona {
    public static void hola(){
        System.out.println("hola emp");
    }
}
class Persona {
    public static void hola(){
        System.out.println("hola per");
    }
}
```



¿Y a fin de cuentas para que sirve todo?

Clase Animal

```java
public class AnimalN
{
    private String nombre;

    public String getNombre(){
      return nombre;
    }

    public void setNombre(String n){
      nombre= n;
    }

    public void sonido(){
    System.out.println("No tiene sonido, solo tienen sonido las subclases");
    }
}
```

¿Alguna duda?

Gato y Perro

¿Alguna duda?

```java
public class GatoN  extends AnimalN
{
    public GatoN(){}

    public GatoN(String nombre){
        this.setNombre(nombre);
    }

  @Override
    public void sonido(){
        System.out.println("Miau Miau");
    }
}
```

```java
public class PerroN extends AnimalN
{
    public PerroN(){}

    public PerroN(String nombre){
        this.setNombre(nombre);
    }

    @Override
    public void sonido(){
        System.out.println("Guau Guau");
    }
}
```

Ejercicio: Cree una clase principal donde debe:

- Crear 2 instancias de perros.
- Crear 2 instancias de gatos.
- Guárdelos en un arreglo.
- Invoque otro método que recorra el arreglo y muestre el sonido que hace cada.

Solución SIN polimorfismo

```
public class PrincipalAnimalN
{
    public static void main(String[] args){
        GatoN g1 = new GatoN("Jackie");
        GatoN g2 = new GatoN("Ozzy");
        PerroN p1 = new PerroN("Galaxia");
        PerroN p2 = new PerroN("Trosqui");
        GatoN[] gatos = {g1,g2};
        PerroN[] perros = {p1,p2};
        sonidosGatos(gatos);
        sonidosPerros(perros);
    }

    public static void sonidosGatos(GatoN[] gatos){
        for(GatoN gato:gatos){
            System.out.println(gato.getNombre()+ ", tiene sonido: ");
            gato.sonido();
        }
    }

    public static void sonidosPerros(PerroN[] perros){
        for(PerroN perro:perros){
            System.out.println(perro.getNombre()+ ", tiene sonido: ");
            perro.sonido();
        }
    }
}
```

Tocaría definir tantos métodos, y arreglos como tipos de animales existan.

Suponga que no son solo 2 tipos de animales, si no 100.

Tocaría definir 100 arreglos, y 100 métodos diferentes.

¿Qué imprime?

```
BlueJ: Ventana de Ter...
Opciones
Jackie, tiene sonido:
Miau Miau
Ozzy, tiene sonido:
Miau Miau
Galaxia, tiene sonido:
Guau Guau
Trosqui, tiene sonido:
Guau Guau
```

¿Qué observa?

Solución CON polimorfismo

```
public class PrincipalAnimal
{
    public static void main(String[] args){
    Gato g1 = new Gato("Jackie");
    Gato g2 = new Gato("Ozzy");
    Perro p1 = new Perro("Galaxia");
    Perro p2 = new Perro("Trosqui");

    Animal[] animales = {g1,g2,p1,p2};
    sonidos(animales);

    }

    public static void sonidos(Animal[] animales){
        for(Animal animal: animales){
            System.out.println(animal.getNombre() + ", Tiene sonido: ");
            animal.sonido();
        }
```

¿Qué observa?

BlueJ: Ventana de Ter...

Opciones

```
Guau Guau
Jackie, Tiene sonido:
Miau Miau
Ozzy, Tiene sonido:
Miau Miau
Galaxia, Tiene sonido:
Guau Guau
Trosqui, Tiene sonido:
Guau Guau
```

Ahora solo existe un método y un solo arreglo

¿Y si queremos administrar 100 tipos de animales diferentes? Ese mismo arreglo y ese mismo método continúan sirviendo

Esta misma solución serviría para sistemas de juegos que manejen múltiples personajes, sistemas con múltiples tipos de productos, etc.

Retomando – Clase Animal

- En el ejemplo anterior, vimos como nada prohibía crear instancias de la clase Animal.
- Aunque realmente la clase Animal por si sola no produce ningún sonido.
- Esa clase en realidad nos **servía más como instrumento de herencia** que en si como clase para ser instanciada.
- Luego nos dimos cuenta que las implementaciones interesantes se realizan era a nivel de las subclases (Gato, Perro).
- Entonces la pregunta que resulta es: ¿Podemos definir clases que nos sirvan como instrumentos de herencia, pero las cuales no se puedan instanciar?.
- Si, y eso se conoce como **Clase Abstracta**.

Clase Abstracta

Luisa Fernanda Restrepo.

Clase Abstracta

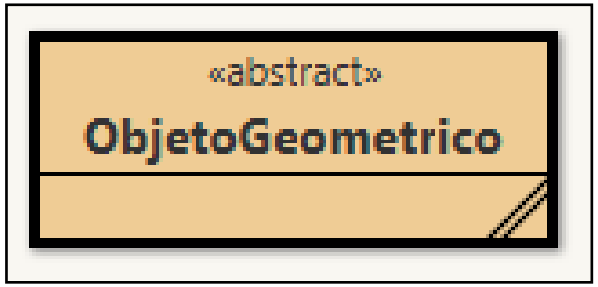
- Las **clases abstractas**, son un tipo especial de clase que agrupa características o elementos que deberían implementar sus subclases.

Particularidades:

- **No se pueden instanciar** (no se puede hacer new MiClaseAbstracta()).
- En una clase abstracta se pueden definir métodos abstractos. Un **método abstracto** se define sin cuerpo, y su implementación será proporcionada por las subclases.
- Una clase que **contiene métodos abstractos debe definirse como abstracta**.
- El **constructor en la clase abstracta se define como protegido**, ya que sólo lo utilizan las subclases.
- Una clase abstracta **puede implementar métodos normales** (no abstractos) que serán heredados por las subclases.

ObjetoGeometrico

```java
import java.util.Date;
public abstract class ObjetoGeometrico {
    private String color = "white";
    private Date dateCreated;
    protected ObjetoGeometrico() {
        dateCreated = new Date();
    }

    protected ObjetoGeometrico(String color) {
        dateCreated = new Date();
        this.color = color;
    }

    public String getColor() {return color;}

    public abstract double getArea();

    public abstract double getPerimetro();
}
```



Forma de definir una clase Abstracta

Constructores protegidos, pero también podrían ser públicos (aunque colocarlos públicos no tiene mucho sentido por que igual no se podrán invocar desde fuera)

Métodos “normales” que son heredados por las subclases

Métodos abstractos, su implementación se debe definir en las subclases

Circulo

```
class Circle extends ObjetoGeometrico {
    private double radius;
    public Circle() {}
    public Circle(double radius) {this.radius = r

    @Override
    public double getArea() {
        return this.radius * this.radius * Math.PI;
    }

    @Override
    public double getPerimetro() {
        return 2 * this.radius * Math.PI;
    }
}
```

Forma de heredar/extender de una clase abstracta

Se DEBEN sobreescribir los métodos abstractos, y se DEBE definir sus implementaciónes

¿Para qué sirven la clases abstractas?

```
public class PrincipalAnimal
{
    public static void main(String[] args) {
        Gato g1 = new Gato("Hernando");
        Gato g2 = new Gato("Maria");
        Perro p1 = new Perro("Juan");
        Perro p2 = new Perro("Daniel");

        Animal[] animales = {g1, g2, p1, p2};
        sonidos(animales);
    }

    public static void sonidos(Animal[] animales) {
        for(Animal animal: animales){
            System.out.println(animal.getNombre() + ",
            animal.sonido();
        }
    }
}
```

¿Cómo se podrían utilizar en este ejemplo, y cuales serian los beneficios?

La clase animal se podría definir como abstracta.

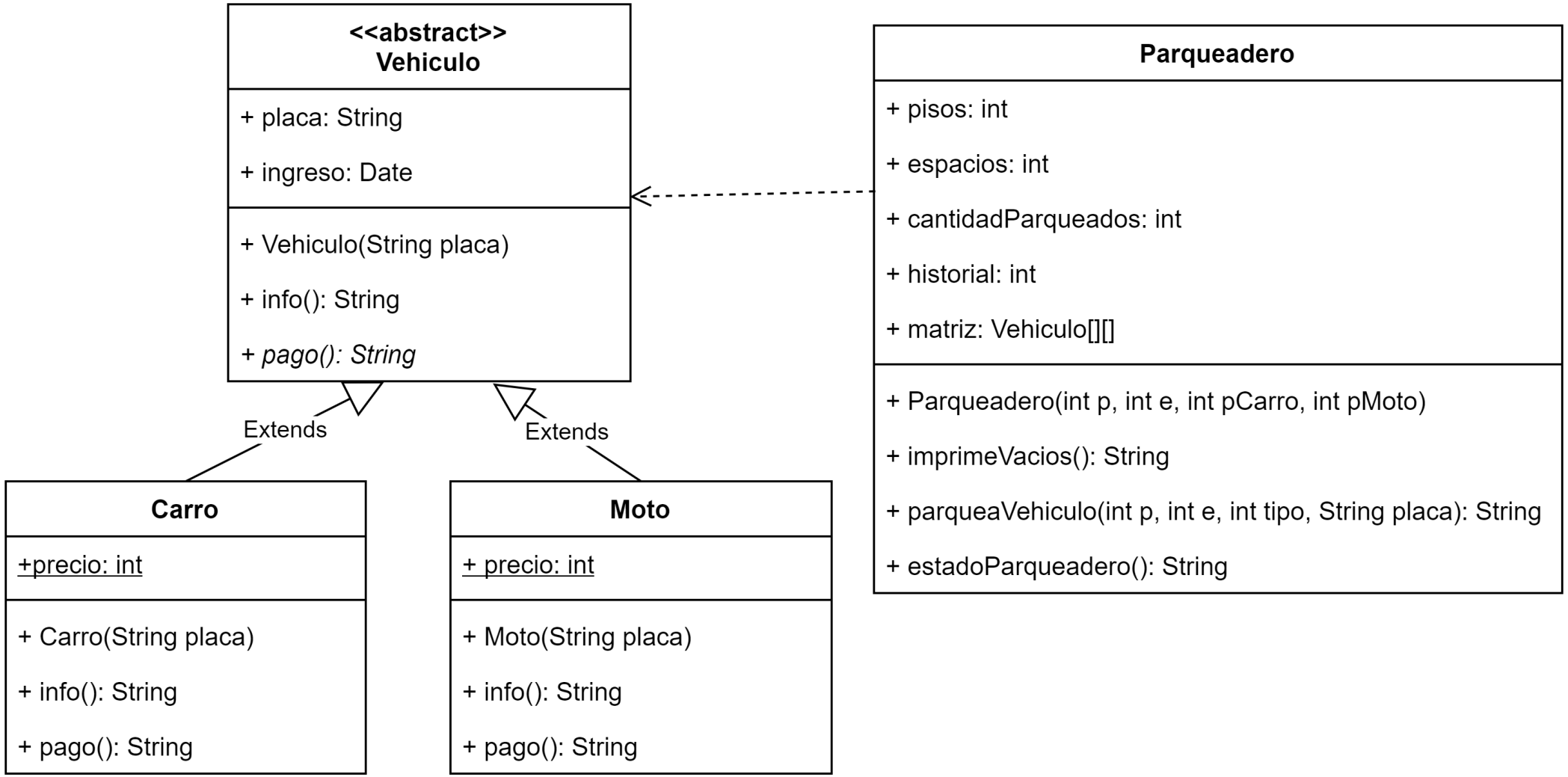
La clase animal debería definir el método sonido como abstracto. Y de esta manera garantizaríamos siempre, que cualquier clase que herede de Animal, deba implementar el método sonido

Nota importante: aunque no se pueda hacer un "new" de una instancia de una clase abstracta. Si se puede hacer un "new" de un arreglo de una clase abstracta.

Ejercicio

- Modifique el ejemplo de sonidos de animales, ponga la clase Animal como abstracta, y el método sonido como abstracto.

Modificaciones al Proyecto 2 - Parqueadero



Interfaces

Luisa Fernanda Restrepo.

Interfaces

Interfaces

- Recordemos Java no permite herencia múltiple, sin embargo por medio de las interfaces se puede obtener un comportamiento similar.
- Una **interfaz** es un concepto similar al de clase abstracta, sin embargo es mas restringida.
- Una interfaz **solo puede contener métodos públicos abstractos y/o constantes***.

Clases

- Una clase puede heredar de otra, y a su vez implementa una interfaces (simulación de herencia múltiple).
- Incluso una clase puede implementar múltiples interfaces (se separan por comas).

*A partir de Java 8, una interface puede definir métodos estáticos (los cuales no son heredados por la clase que implemente la interface) y además puede definir “métodos por defecto”.

Ejemplo interfaces

```
public interface Comestible {
    public abstract String howToEat();
}
abstract class Animal {
    public abstract String sound();
}
class Chicken extends Animal implements Comestible {
    @Override
    public String howToEat() {return "Chicken: Fry it";}
    @Override
    public String sound() {return "Chicken: cock-a-doodle-doo";}
}
```

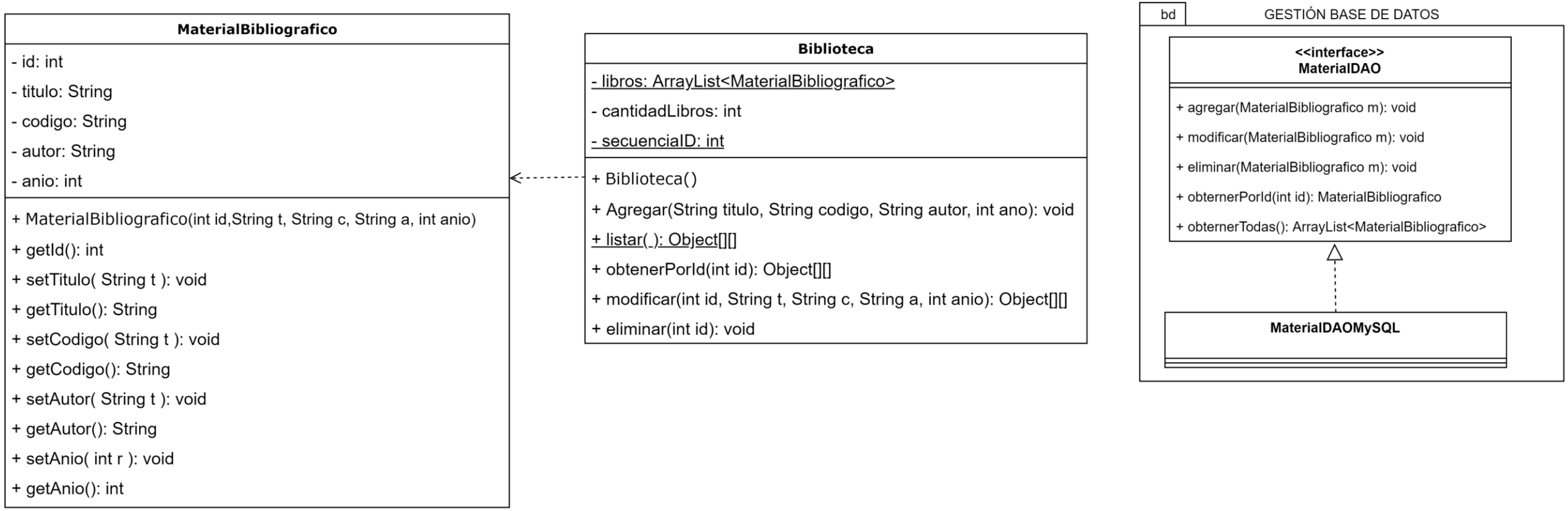
Forma de definir una interfaz

Se podría omitir tanto las palabras "public" como "abstract", ya que igual todos los métodos serán públicos abstractos.

Alternativa a herencia múltiple en Java

DEBE sobreescribir estos dos métodos o sacará error

Modificaciones al Proyecto 1 - Biblioteca



Referencias

Basado en el material elaborado por: Daniel Correa (docente EAFIT).

Liang, Y. D. (2017). Introduction to Java programming: comprehensive version. Eleventh edition. Pearson Education.

Streib, J. T., & Soma, T. (2014). *Guide to Java*. Springer Verlag.