CICLO 2

[FORMACIÓN POR CICLOS]

Programación Básica JAVA

Semana 3

Ingeni@ Soluciones TIC

UNIVERSIDAD DE ANTIOQUIA
Facultad de Ingeniería

Programa

Semana 3	Herencia e interfaces	2	5	7
	Polimorfismo	2	5	7
	Colecciones en Java	3	7	10

Contenido

- Herencia e interfaces
- Quizziz

- Polimorfismo

- Colecciones en Java
- Quizizz

- Ejemplo de Implementación

Herencia

La herencia es la habilidad de crear clases mediante la absorción de los miembros de una clase existente (sin copiar y pegar el código), mejorándolos con nuevas Capacidades.

Es compartir atributos y métodos entre clases. Se parte de una clase y se crea una nueva clase que puede adquirir los miembros de una existente, y se mejora con nuevas capacidades o modificando las capacidades ya existentes

SUPERCLASE:

Clase padre o clase base, no necesita de subclases para existir, y las modificaciones realizadas en la subclase no afecta el comportamiento ni los elementos de la superclase.

Herencia

SUPERCLASE DIRECTA:
es la superclase a partir de la cual la subclase hereda en forma explícita

SUPERCLASE INDIRECTA:
es cualquier clase arriba de la superclase directa en la jerarquía de clases

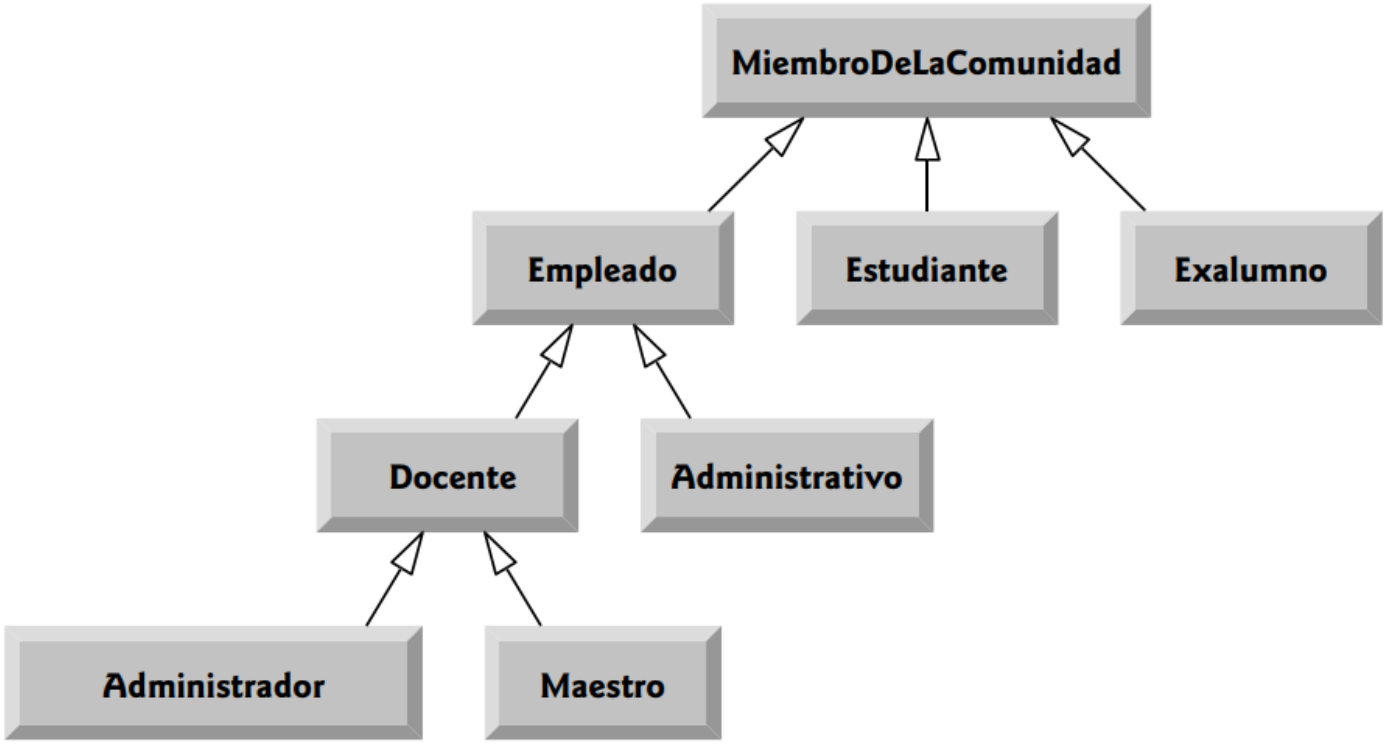
Herencia

SUBCLASE:

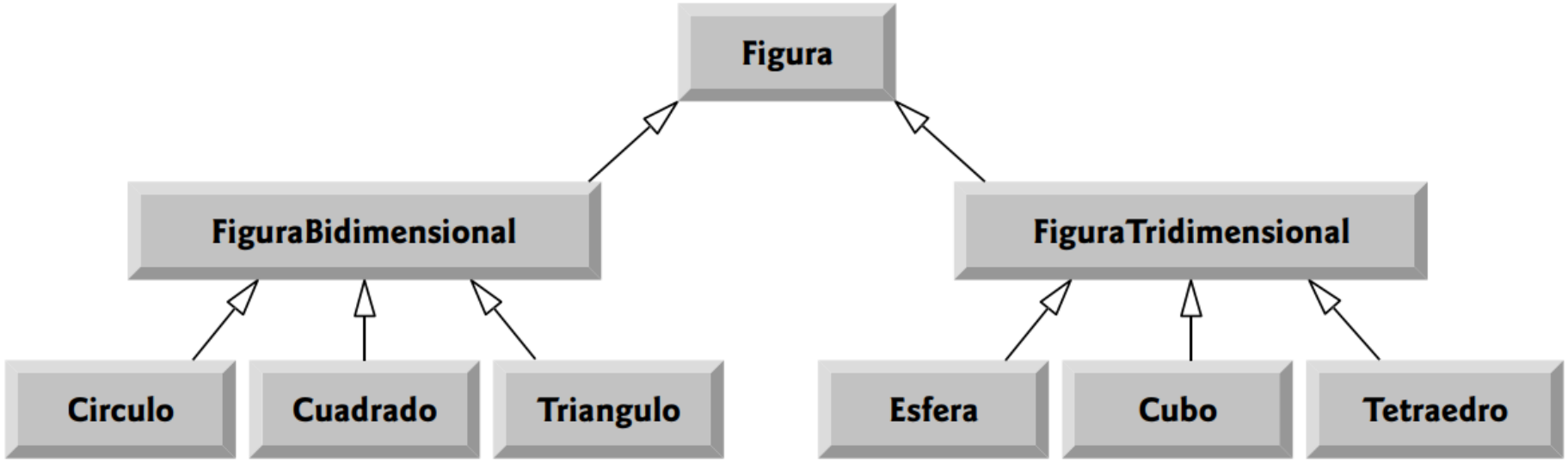
Llamada clase hija, hereda todos los atributos y métodos de la clase padre. Dependiendo en que se definan estos atributos y métodos, además del lenguaje de programación, la subclase puede tener o no acceso a ciertos elementos heredados. La clase hija puede agregar o redefinir elementos heredados.

Una subclase requiere que exista la superclase.

Relación de Herencia



Relación de Herencia



Herencia

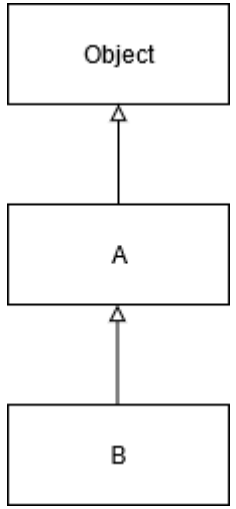
Todas las clases heredan de **Object**, de forma implícita o explicita

Equivalencia:

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

Si se define una clase que herede de A, llamada B

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



Constructor sin parámetros

Por defecto en Java se define un constructor sin parámetros

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

Aunque si se define por lo menos un constructor (con un parámetro), el constructor por defecto no será creado automáticamente (debe hacerse de forma manual).

Super

Super es una palabra clave que permite hacer el llamado al constructor de la superclase desde una subclase, además acceder a métodos y atributos

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

Quizziz

https://quizizz.com/join?gc=45647526

Polimorfismo

Pilares de la programación orientada a objetos:

• La **abstracción** consiste en aislar un elemento de su contexto (lo que lo rodea), así como simplificarlo. Esto, es lo que hacemos cuando definimos qué atributos, métodos y clases asociadas tendrá una clase que estamos diseñando, y qué otras cosas, definitivamente, no tendrá.

• El **encapsulamiento** consiste en el ocultamiento del estado de los objetos. Este principio propende por que los atributos de las clases sean privados y sus métodos sean públicos. Así, los atributos de las clases se deberían de poder acceder sólo mediante getters o setters, u otros métodos públicos ideados para accederlos.

• La **herencia**, como hemos visto, consiste en la posibilidad de que una clase sea creada a partir de otra ya existente, obteniendo mediante herencia métodos y atributos de dicha clase (su clase padre).

Polimorfismo

En la programación orientada a objetos, el **polimorfismo** permite que un objeto pueda responder de diferentes maneras a un mismo mensaje. Es decir, permite que, al llamar un método de un objeto dado, este funcione de manera diferente.

En Java, el polimorfismo se da de dos formas, mediante **ligadura estática** y mediante **ligadura dinámica**. En ambos casos, se trata de que el programa "decida" qué instrucciones (o método) se ejecuten en un momento dado

Polimorfismo

• En la **ligadura estática**, el código a ejecutar al llamar un método se especifica en tiempo de compilación, es decir, en el momento en que el código se compila antes de ejecutarse. La ligadura estática se da, por ejemplo, al presentarse ***sobrecarga de métodos***: si una clase tiene varios métodos del mismo nombre y se llama uno de ellos, es en tiempo de compilación que el programa determina exactamente cuál método es el que se está llamando y, por lo tanto, que instrucciones se deben ejecutar.

• En la **ligadura dinámica**, el código a ejecutar al llamar un método se especifica en tiempo de ejecución. Es decir, una vez el programa ya ha sido compilado y está en ejecución. Normalmente, este tipo de polimorfismo se da al "variar" entre implementaciones de un mismo método en una jerarquía de clases.

Polimorfismo

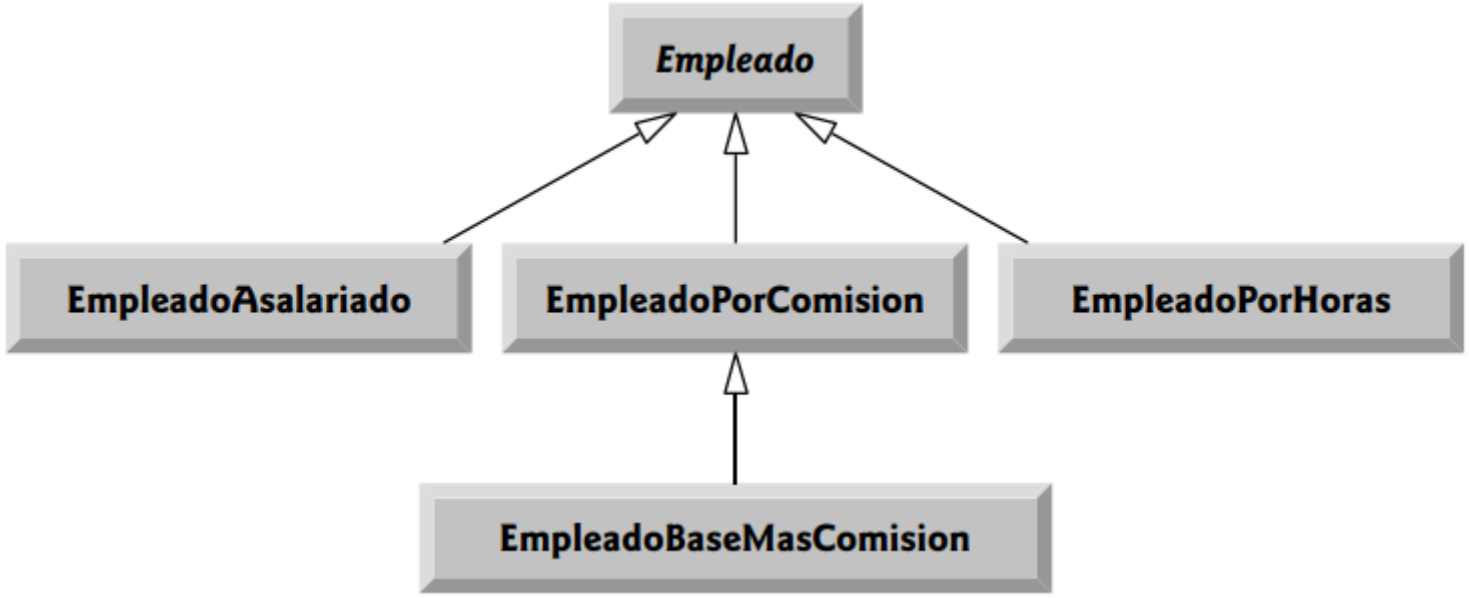
```
Bicicleta cicla1, cicla2, cicla3;

cicla1 = new Bicicleta();
cicla2 = new BicicletaMontania(5, "Trek", "Rojo", 0, "SRAM");
cicla3 = new BicicletaRuta(32, "Specialized", "Azul", 0, "Shimano");

System.out.println(cicla1.getDescripcion());
System.out.println(cicla2.getDescripcion());
System.out.println(cicla3.getDescripcion());
```

En las tres líneas finales se llamó el mismo método (getDescripcion), pero se ejecutó siempre aquel correspondiente a la instancia llamada.

Polimorfismo

El polimorfismo nos permite “programar en forma general ”, en vez de “programar en forma específica”. En particular, nos permite escribir programas que procesen objetos que compartan la misma superclase (ya sea de manera directa o indirecta) como si todos fueran objetos de la superclase; esto puede simplificar la programación.

Polimorfismo



Codifique y encuentre el error:

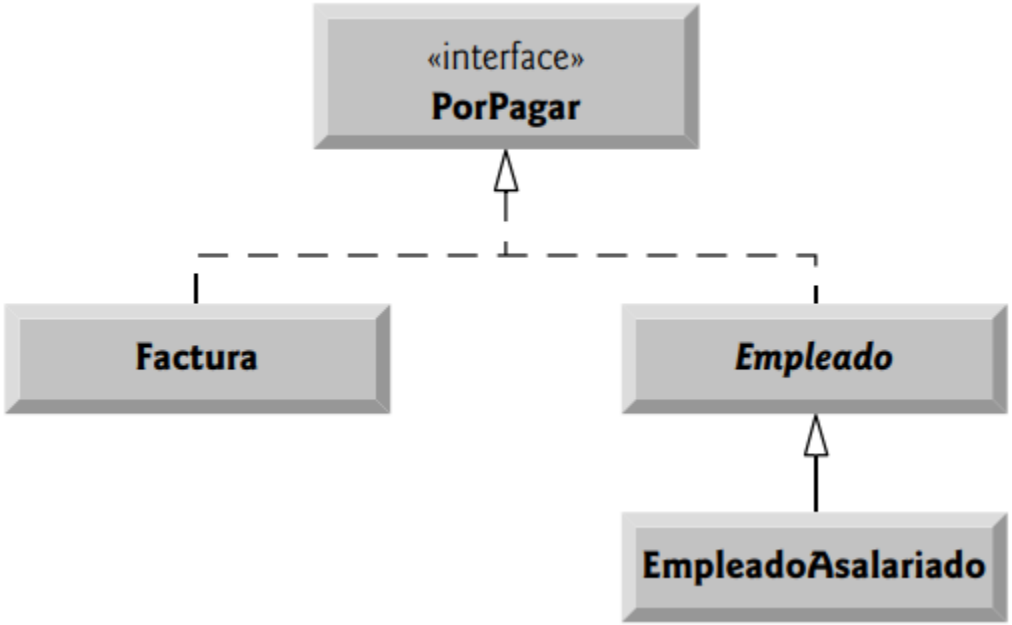
```java
package polimorfismo;
public class Polimorfismo {
    public static void main(String[] args) {
        FiguraBidimensional fig1;
        FiguraBidimensional fig2;

        fig1 = new FiguraBidimensional("Primer_cuadrado");
        System.out.println(fig1.area());

        fig2 = new Cuadrado("Segundo_cuadrado");
        fig2.lado = 5;

        System.out.println(fig2.area());
    }
}
```

```java
public class Cuadrado extends FiguraBidimensional {
    public double lado;

    public Cuadrado(String nombre){
        super(nombre);
    }
    public Cuadrado(String nombre, double lado){
        super(nombre);
        this.lado = lado;
    }
    public Cuadrado(double lado){
        this.lado = lado;
    }

    @Override
    public String area(){
        double a = lado*lado;
        return ("El área del cuadrado es: "+ String.valueOf(a));
    }
}
```

```java
package polimorfismo;
public class FiguraBidimensional {
    public String nombre;

    public FiguraBidimensional(){}
    public FiguraBidimensional(String nombre){
        this.nombre = nombre;
    }

    public String area(){
        return ("No se puede calcular el área");
    }
}
```

Interfaces

Una interfaz de Java describe un conjunto de métodos que pueden llamarse sobre un objeto; por ejemplo, para indicar al objeto que realice cierta tarea, o que devuelva cierta pieza de información.



Interfaces

En Java, una interfaz es un tipo similar a una clase (no confundir con interfaces gráficas de usuario), que sólo puede contener:

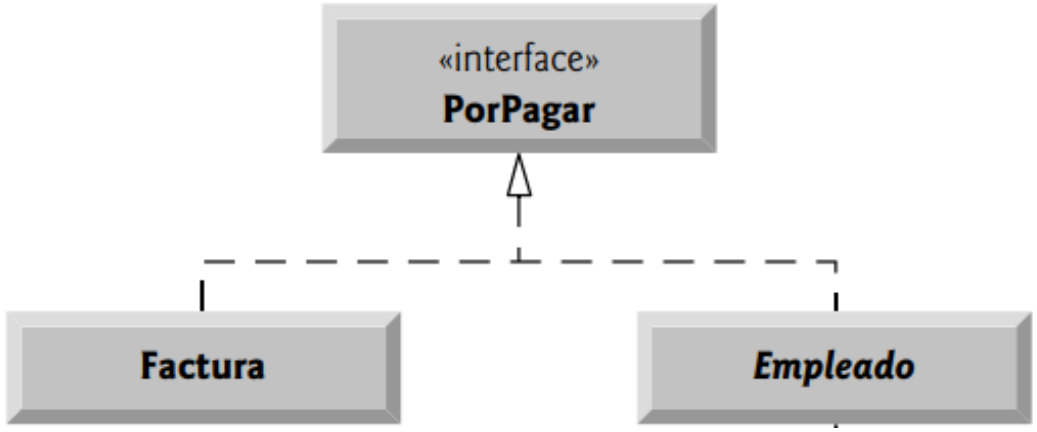
- Constantes (ya que los atributos que les agreguemos serán public static final de manera implícita y por defecto).
- Declaraciones de métodos públicos (es decir, sin cuerpo), que por defecto y de manera implícita serán abstract final.
- Métodos estáticos.
- Métodos default.

Interfaces

```
interface SerPadre {
        void criar();
       }
interface SerEsposo {
        void amar();
       }
interface SerEmpleado {
        void trabajar();
       }
class EmpleadoDeOracle {
        public void trabajar(){}
       }
class Programador extends EmpleadoDeOracle implements serPadre,
serEsposo, serEmpleado{
      public void criar(){}
      public void amar(){}
      }
```

Interfaces

Codifique la siguiente interfaz:



Clases Abstractas

Una **clase abstracta** es una clase que cuenta con uno o varios métodos abstractos. Un método abstracto es aquel en el cual se declara el nombre, su tipo de retorno, y sus parámetros, pero se especifica como abstracto y como tal no tiene cuerpo (no cuenta como implementación)

Implemente el código y halle el error

```java
package asbtracta;

public class Asbtracta {

    public static void main(String[] args) {

        pajaro canario = new pajaro();
        persona cantante= new persona();

        canario.cantar();
        cantante.cantar();

    }
}
```

```java
package asbtracta;

public abstract class acciones {

    public abstract void cantar();
    public abstract void caminar();


}
```

```java
package asbtracta;

public class pajaro extends acciones {

    public void cantar(){
        System.out.println("Pajaro cantando");

    }

}
```

```java
package asbtracta;
public class persona extends acciones {

    public void cantar(){
        System.out.println("Persona cantando");

    }
}
```

Colección: ArrayList

En Java, una **colección** es un objeto que agrupa varios elementos del mismo tipo en una sola unidad. Las colecciones se suelen usar para almacenar, obtener, manipular y comunicar datos agrupados. Por lo anterior, son ampliamente usadas en todos los lenguajes de programación, si bien cada lenguaje tiene su propio conjunto de colecciones. Por lo general, dentro de una aplicación en desarrollo, una colección representa un grupo natural de elementos.

La clase ARRAYLIST

Un ***ArrayList*** es un objeto de la clase del mismo nombre, con un desempeño aceptable en la mayoría de los casos y que funciona como un *array* (o arreglo) expandible de forma dinámica. Es decir, un ArrayList se comporta como un arreglo al cual le podremos agregar, modificar y quitar elementos de acuerdo a nuestras necesidades. Algo a tener presente es que los elementos de un *ArrayList* deben ser del mismo tipo, de manera similar a como ocurre con los *arrays* convencionales.

La clase
ARRAYLIST

Tipo primitivo	Wrapper class
boolean	Boolean
byte	Byte
short	Short
char	Character
int	Integer
long	Long
float	Float
double	Double

La clase
ARRAYLIST

https://quizizz.com/join?gc=33851046

Codifique en un archivo de Java

```java
import java.util.ArrayList;
public class Arreglos {

    public static void main(String[] args) {
        // declaración de arreglos
        ArrayList<Integer> precio = new ArrayList<Integer>();
        precio.add(30);
        precio.add(40);
        precio.add(50);
        precio.remove(2);
        precio.add(60);
        precio.remove(0);
        precio.add(70);
        precio.set(0, Integer.SIZE);

        System.out.println("Precio 1: " + precio.get(1));
        System.out.println("Precio 1: " + precio.get(0));
    }
}
```

Codifique en un archivo de Java

```
8  import java.util.ArrayList;
9  public class Arreglos {
10 //public static ArrayList<Producto> productos = new ArrayList<Producto>();
11     public static void main(String[] args) {
12         // arreglos dinámicos con objetos
13         ArrayList<Producto> productos = new ArrayList<Producto>();
14         Producto nombre1 = new Producto("Pan");
15         Producto nombre2 = new Producto("Café");
16         Producto nombre3 = new Producto("Azucar");
17         productos.add(nombre1);
18         productos.add(nombre2);
19         productos.add(nombre3);
           for(Producto item:productos){
21             System.out.printf("el producto es %s%n",item.getProducto());
22         }
23         productos.remove(0);
24         System.out.println(productos.get(0).getProducto());
25     }
26 }
```