Algoritmos y Estructura de Datos

Unidad 4: Técnicas Avanzadas de POO

Tema 13: Interfaz

Tema 13: Interfaz

Índice

Capacidades

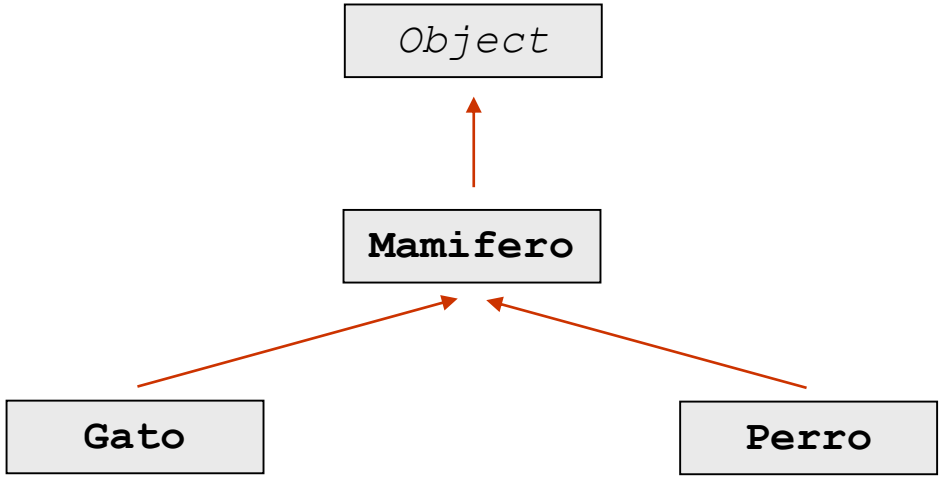
- Implementa herencia utilizando superclases abstractas.
- Implementa herencia múltiple a través de interfaces.

4.2.1 Clases y métodos abstractos

- Una clase abstracta es aquella que nos se puede instanciar. Se usa únicamente para crear subClases
- Un método es abstracto cuando no tiene implementación y solamente se define con el objetivo de obligar a que en cada clase **Hijo** que deriva de la clase abstracta se realice la correspondiente implementación.
- En Java, tanto la clase como el método abstracto se etiqueta con la palabra reservada *abstract*.
- Si una clase tiene por lo menos un método abstracto, entonces la clase tiene que ser abstracta, de lo contrario el compilador mostrará un mensaje de error.
- Sin embargo, una clase abstracta no está obligada a tener métodos abstractos.

4.2.2 Polimorfismo y enlace dinámico

- El polimorfismo y enlace dinámico se da cuando se elige el método a ejecutar en tiempo de ejecución, en función de la clase del objeto; es la implementación del polimorfismo.
- Por ejemplo:



4.2.2 Polimorfismo y enlace dinámico

```java
package padre;

public abstract class Mamifero {

    // Método público
    public String mensaje() {
        return "soy mamífero";
    }
    // Método público abstracto (no se implementa)
    public abstract String hacerRuido();

}
```

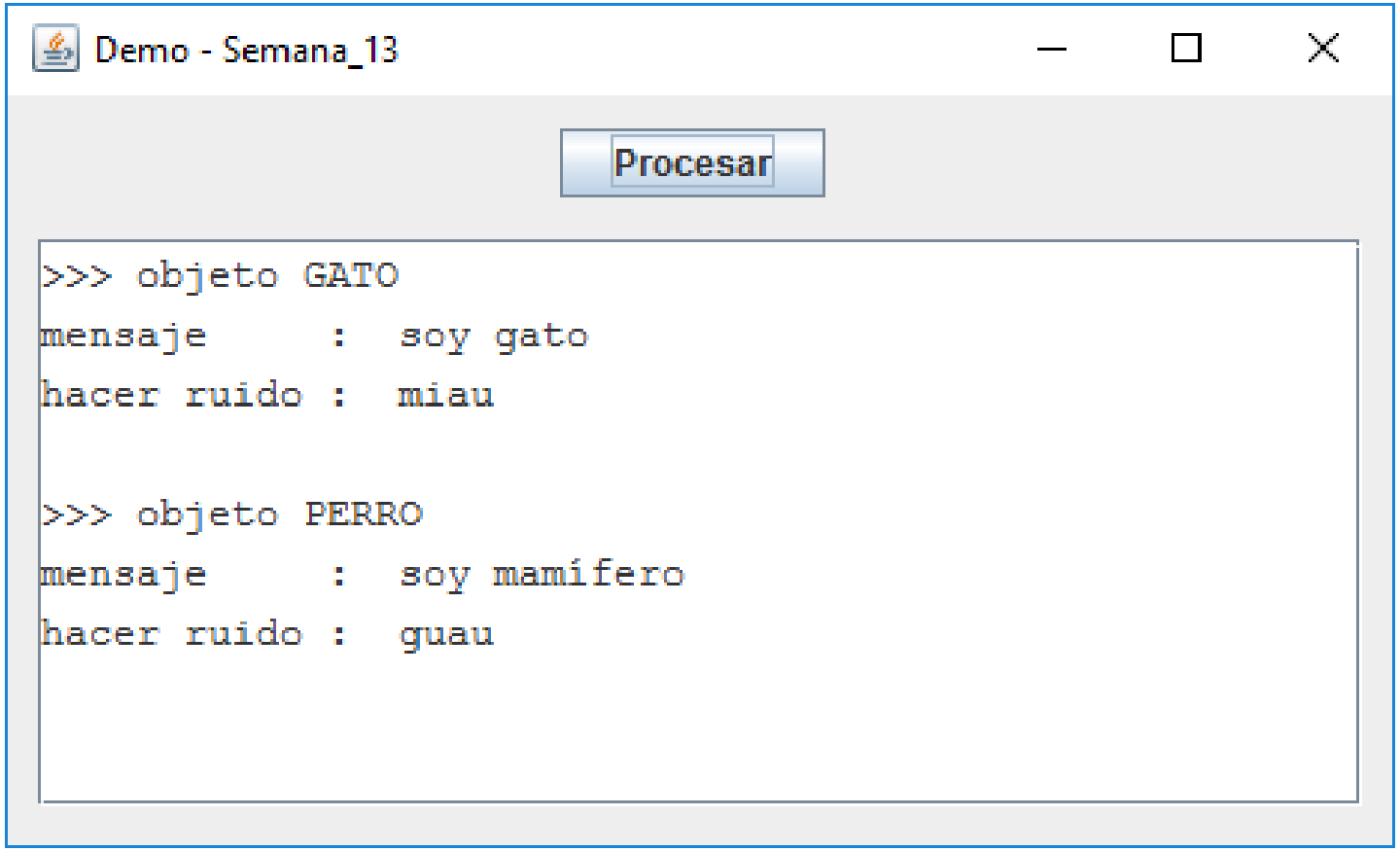
4.2.2 Polimorfismo y enlace dinámico

```java
package hijos;

import padre.Mamifero;

public class Gato extends Mamifero {

    // Método público
    public String mensaje() {
        return "soy gato";
    }
    // Método público (obligatorio)
    public String hacerRuido() {
        return "miau";
    }

}
```

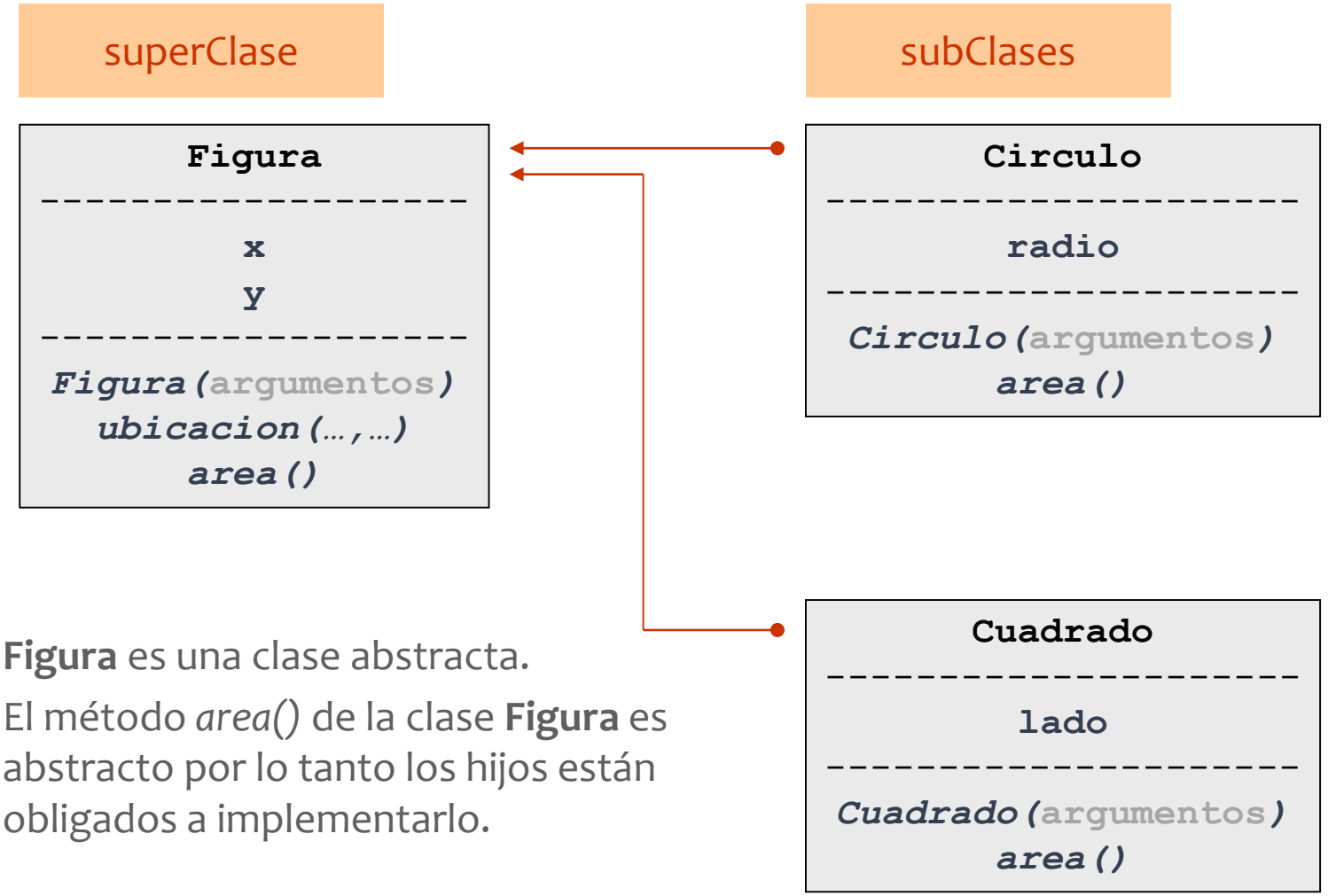
```java
package hijos;

import padre.Mamifero;

public class Perro extends Mamifero {

    // Método público (obligatorio)
    public String hacerRuido() {
        return "guau";
    }

}
```

4.2.2 Polimorfismo y enlace dinámico

- En el ejemplo, el método de la clase principal es un método polimórfico, ya que cuando es llamado por primera vez, lista el comportamiento de un Gato y la segunda vez, lista el comportamiento de un Perro.

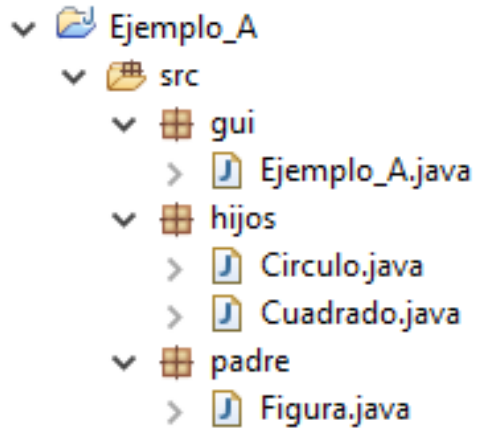


4.2.2 Polimorfismo y enlace dinámico



- **Figura** es una clase abstracta.
- El método *area()* de la clase **Figura** es abstracto por lo tanto los hijos están obligados a implementarlo.

Ejemplo_A



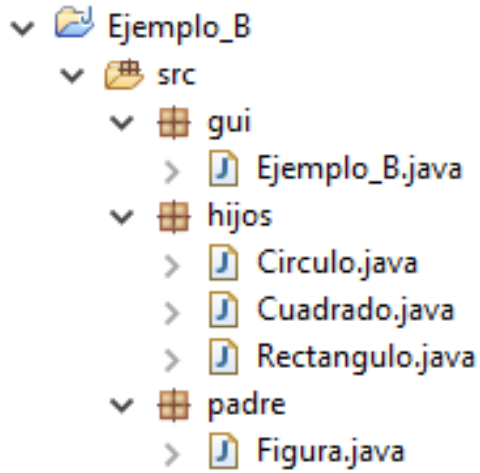
a) Implementa la clase **Figura** en el *package* **padre** los atributos protegidos x, y de tipo entero; un constructor que inicializa los atributos y el método *ubicacion()* que retorna las coordenadas de una figura. Anuncia dos métodos abstractos: *medida()* y *area()* para que las clases **Hijo** las implementen.

b) Implementa la clase **Circulo** en el *package* **hijos** que hereda información de la clase **Figura**, invoca al constructor de la clase **Figura** y declara como privado el ***radio*** e implementa por obligación los métodos *medida()* y *area()*.

c) Implementa la clase **Cuadrado** en el *package* **hijos** que hereda información de la clase **Figura**, invoca al constructor de la clase **Figura** y declara como privado el ***lado*** e implementa por obligación los métodos *medida()* y *area()*.

d) En **Ejemplo_A** declara dos objetos: de tipo Circulo y Cuadrado respectivamente. Finalmente a través de un método listado visualiza la información completa de cada objeto.

Ejemplo_B



a) Implementa la clase **Figura** en el *package* **padre** los atributos protegidos x, y de tipo entero; un constructor que inicializa los atributos y el método *ubicacion()* que retorna las coordenadas de una figura. Anuncia dos métodos abstractos: *area()* y *datosCompletos()* para que las clases **Hijo** las implementen.

b) Implementa la clase **Circulo** en el *package* **hijos** que hereda información de la clase **Figura**, invoca al constructor de la clase **Figura** y declara como privado el ***radio*** e implementa por obligación los métodos *area()* y *datosCompletos()*.

c) Implementa la clase **Cuadrado** en el *package* **hijos** que hereda información de la clase **Figura**, invoca al constructor de la clase **Figura** y declara como privado el ***lado*** e implementa por obligación los métodos *area()* y *datosCompletos()*.

d) Implementa la clase **Rectangulo** en el *package* **hijos** que hereda información de la clase **Figura**, invoca al constructor de la clase **Figura** y declara como privados el ***ancho*** y ***alto*** e implementa por obligación los métodos *area()* y *datosCompletos()*.

e) En **Ejemplo_B** declara un arreglo lineal que contiene a los objetos de tipo Circulo, Cuadrado y Rectangulo. Finalmente a través de un solo método listado visualiza la información completa de cada objeto.

4.2.3 Técnicas de casting

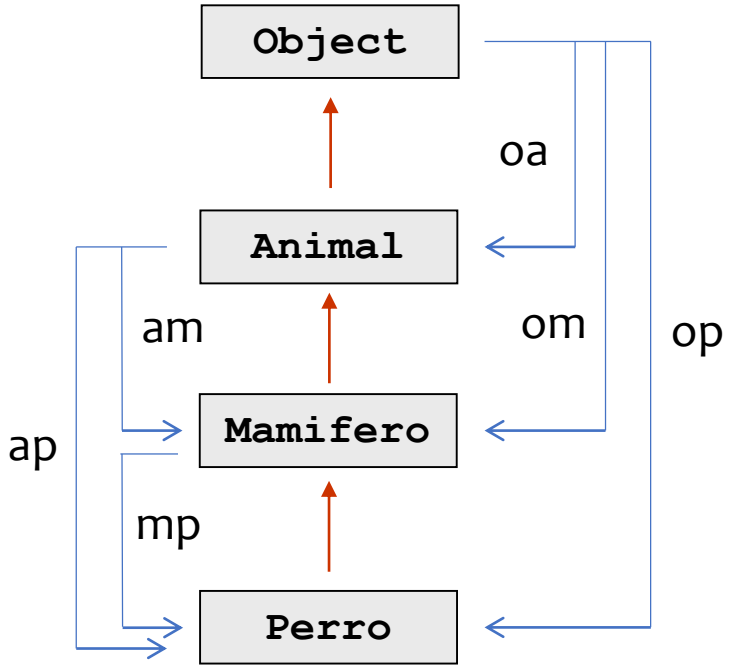
```java
package herencia;

public class Animal {

    // Método público
    public String hacerRuido() {
        return "no definido";
    }

}
```

```java
package herencia;

public class Mamifero extends Animal {

    // Método público
    public String mensaje() {
        return "soy mamífero";
    }

}
```

```java
package herencia;

public class Perro extends Mamifero {

    // Métodos públicos
    public String mensaje() {
        return "soy perro";
    }
    public String hacerRuido() {
        return "guau";
    }

}
```

4.2.3.1 Técnicas de casting: upcasting

- ***Upcasting***: Permite interpretar un objeto de una clase derivada como del mismo tipo que la clase base. También se puede ver como la conversión de un tipo en otro **superior** en la jerarquía de clases. No hace falta especificarlo.



Ejemplos:

```
Object     oa = new Animal();
Object     om = new Mamifero();
Object     op = new Perro();
Animal     am = new Mamifero();
Animal     ap = new Perro();
Mamifero   mp = new Perro();
```

4.2.3.1 Técnicas de casting: upcasting

- **oa, om, op** de tipo *Object* no reconocen los métodos *mensaje()* ni *hacerRuido()* porque no aparecen en la clase *Object*.
- **am, ap** de tipo *Animal* reconocen al método *hacerRuido()*. El primero lo busca en la clase *Mamifero*, pero al no encontrarlo lo ejecuta en la clase *Animal*. El segundo lo busca y ejecuta en la clase *Perro*.
- **mp** de tipo *Mamifero* identifica a los métodos *mensaje() y hacerRuido(), ambos* se ejecutan en la clase *Perro*.

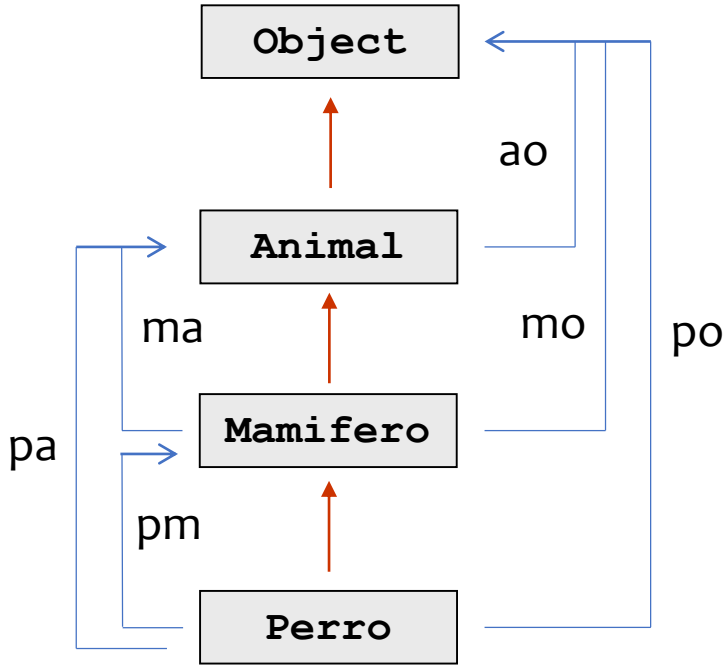
```java
package herencia;

public class Animal {

    // Método público
    public String hacerRuido() {
        return "no definido";
    }

}
```

```java
package herencia;

public class Mamifero extends Animal {

    // Método público
    public String mensaje() {
        return "soy mamífero";
    }

}
```

```java
package herencia;

public class Perro extends Mamifero {

    // Métodos públicos
    public String mensaje() {
        return "soy perro";
    }
    public String hacerRuido() {
        return "guau";
    }

}
```

4.2.3.2 Técnicas de casting: downcasting

- ***Downcasting***: Permite interpretar un objeto de una clase base como del mismo tipo que su clase derivada. También se puede ver como la conversión de un tipo en otro **inferior** en la jerarquía de clases. Se especifica precediendo al objeto a convertir con el nuevo tipo entre paréntesis.



Ejemplos:

```
Animal     ao = (Animal) oa;
Mamifero   mo = (Mamifero) om;
Perro      po = (Perro) op;
Mamifero   ma = (Mamifero) am;
Perro      pa = (Perro) ap;
Perro      pm = (Perro) mp;
```

4.2.3.2 Técnicas de casting: downcasting

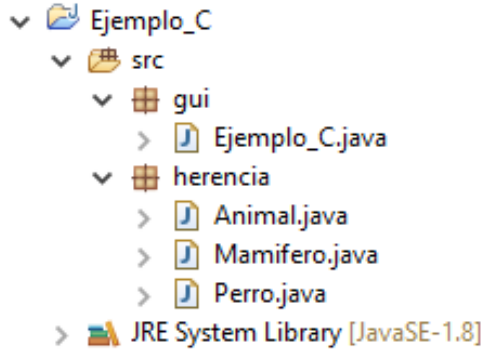
- **ao** de tipo *Animal* reconoce al método *hacerRuido()*, lo busca en dicha clase y lo ejecuta.
- **mo, ma** de tipo *Mamifero* reconocen los métodos *mensaje() y hacerRuido()*, el primero lo ejecuta en la clase *Mamífero* y el segundo en la clase *Animal*.
- **po, pa, pm** de tipo *Perro* reconocen los métodos *mensaje() y hacerRuido()*, ambos se ejecutan en la clase *Perro*.

```java
package herencia;

public class Animal {

    // Método público
    public String hacerRuido() {
        return "no definido";
    }

}
```

```java
package herencia;

public class Mamifero extends Animal {

    // Método público
    public String mensaje() {
        return "soy mamífero";
    }

}
```

```java
package herencia;

public class Perro extends Mamifero {

    // Métodos públicos
    public String mensaje() {
        return "soy perro";
    }
    public String hacerRuido() {
        return "guau";
    }

}
```

Ejemplo_C



a) Implementa en la clase **Animal** (dentro del *package* **herencia),** el método público ***hacerRuido().***

b) Implementa en la clase **Mamifero** (dentro del *package* **herencia),** el método público *mensaje().*

c) Implementa en la clase **Perro** (dentro del *package* **herencia),** los métodos públicos ***mensaje()*** y ***hacerRuido().***

d) En **Ejemplo_C** se aplica Técnicas de casting.

4.2.4 Definición de interfaz

- Una interfaz es una clase completamente abstracta, es decir no tiene implementación. Lo único que puede tener son declaraciones de métodos y definiciones de constantes simbólicas.
- En Java, las interfaces se declaran con la palabra reservada **interface**.
- La clase que implementa una o más interfaces utiliza la palabra reservada *implements*. Para ello, es necesario que la clase implemente todos los métodos definidos por la interfaz.
- Una interfaz podrá verse simplemente como una forma, es como un molde; solamente permite declarar nombres de métodos. En este caso, no es necesario definirlos como abstractos, puesto que lo son implícitamente. Y si adicionalmente tiene miembros datos, éstos serán constantes, es decir, *static* y *final*.
- Al utilizar *implements* para el interfaz es como si se hiciese una acción de copiar y pegar el código de la interfaz, con lo cual no se hereda nada, solamente se pueden usar los métodos.
- La ventaja principal del uso de interfaces es que puede ser implementada por cualquier número de clases, permitiendo a cada clase compartir el interfaz de programación sin tener que ser consciente de la implementación que hagan las otras clases que implementen el interfaz.
- La principal diferencia entre interfaz y clase abstracta es que la interfaz posee un mecanismo de encapsulamiento sin forzar al usuario a utilizar la herencia.

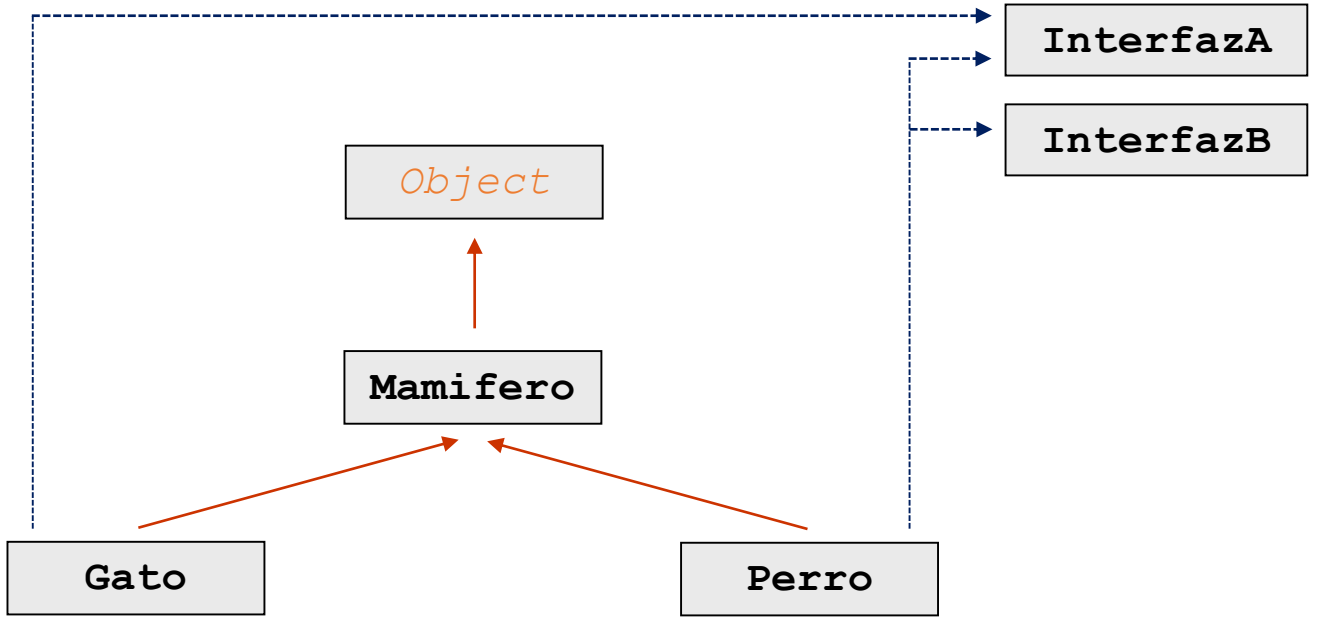
4.2.5 Herencia múltiple

- En Java, realmente no existe la herencia múltiple.
- Lo que se puede hacer es crear una clase que implemente (*implements*) más de un interfaz, pero sólo puede extender una clase (*extends*).
- Por ejemplo:

```
public class Proyecto extends JFrame implements ActionListener, ItemListener {

    …
    public void actionPerformed(ActionEvent arg0) {
        …
    }
    …
    public void itemStateChanged(ItemEvent arg0) {
        …
    }
    …

}
```

4.2.5 Herencia múltiple

Ejemplo:



4.2.5 Herencia múltiple

```
package padre;

public abstract class Mamifero {

    public String mensaje() {
        return "soy mamífero";
    }
    public abstract String hacerRuido();

}
```

```
package interfaces;

public interface InterfazA {

    public double vacunaA = 42.75;
    public String cuidado();

}
```

```
package interfaces;

public interface InterfazB {

    public double vacunaB = 96.28;
    public String peligro();

}
```

```
package hijos;

import padre.Mamifero;
import interfaces.InterfazA;

public class Gato extends Mamifero implements InterfazA {

    public String mensaje() {
        return "soy gato";
    }
    public String hacerRuido() {
        return "miau";
    }
    public String cuidado() {
        return "el gato puede tener rabia";
    }

}
```

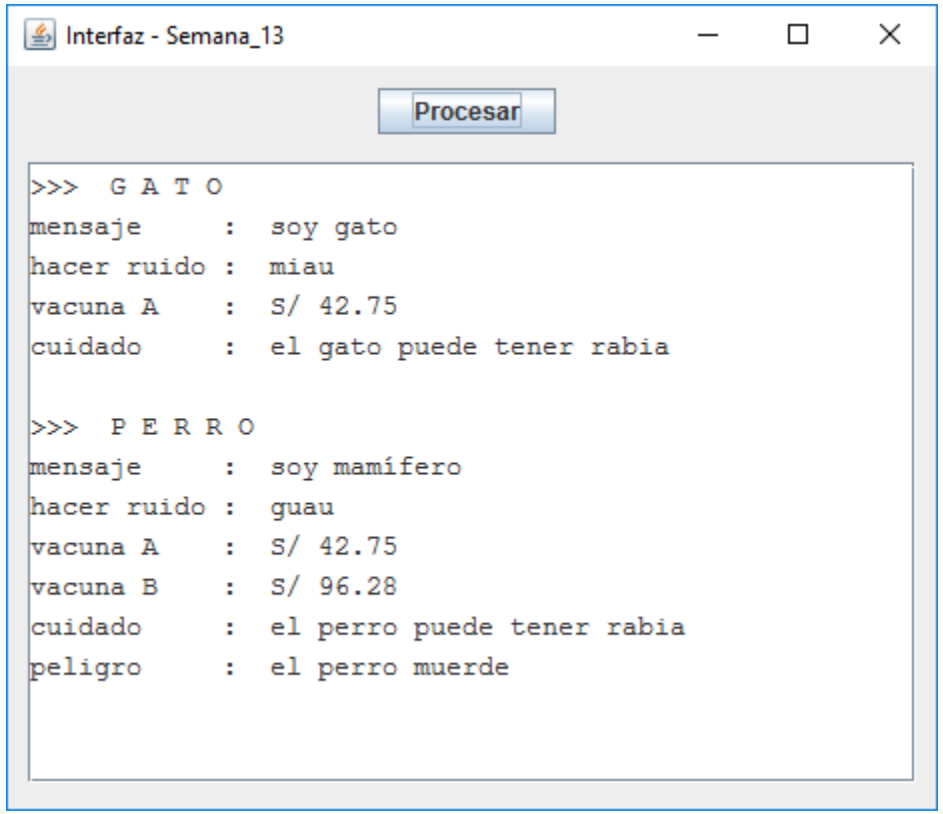
```
package hijos;

import padre.Mamifero;
import interfaces.*;

public class Perro extends Mamifero implements InterfazA, InterfazB {

    public String hacerRuido() {
        return "guau";
    }
    public String cuidado() {
        return "el perro puede tener rabia";
    }
    public String peligro() {
        return "el perro muerde";
    }

}
```

4.2.5 Herencia múltiple

```
protected void actionPerformedBtnProcesar(ActionEvent arg0) {
    Gato g = new Gato();
    listado(g);

    Perro p = new Perro();
    listado(p);
}
```

```
void listado(Mamifero x) {
    if (x instanceof Gato)
        imprimir(">>>  G A T O");
    else
        imprimir(">>>  P E R R O");

    imprimir("mensaje     :  " + x.mensaje());
    imprimir("hacer ruido :  " + x.hacerRuido());

    if (x instanceof Gato) {
        imprimir("vacuna A    :  S/ " + ((Gato) x).vacunaA);
        imprimir("cuidado     :  " + ((Gato) x).cuidado());
    }
    else {
        imprimir("vacuna A    :  S/ " + ((Perro) x).vacunaA);
        imprimir("vacuna B    :  S/ " + ((Perro) x).vacunaB);
        imprimir("cuidado     :  " + ((Perro) x).cuidado());
        imprimir("peligro     :  " + ((Perro) x).peligro());
    }
    imprimir("");
}
```

4.2.5 Herencia múltiple



GRACIAS

SEDE MIRAFLORES
Calle Diez Canseco Cdra 2 / Pasaje Tello
Miraflores – Lima
Teléfono: 633-5555

SEDE INDEPENDENCIA
Av. Carlos Izaguirre 233
Independencia – Lima
Teléfono: 633-5555

SEDE BREÑA
Av. Brasil 714 – 792
(CC La Rambla – Piso 3)
Breña – Lima
Teléfono: 633-5555

SEDE TRUJILLO
Calle Borgoño 361
Trujillo
Teléfono: (044) 60-2000

SEDE SAN JUAN DE LURIGANCHO
Av. Próceres de la Independencia 3023-3043
San Juan de Lurigancho – Lima
Teléfono: 633-5555

SEDE LIMA CENTRO
Av. Uruguay 514
Cercado – Lima
Teléfono: 419-2900

SEDE BELLAVISTA
Av. Mariscal Oscar R. Benavides 3866 – 4070
(CC Mall Aventura Plaza)
Bellavista – Callao
Teléfono: 633-5555

SEDE AREQUIPA
Av. Porongoche 500
(CC Mall Aventura Plaza)
Paucarpata - Arequipa
Teléfono: (054) 60-3535