

Universidad de Valparaíso
Facultad de Ingeniería



Escuela de
Ingeniería Informática

Programación orientada a objetos

Diseño de clases

Método no formal

Diseñar sin preocuparse
cómo se utilizará la clase

¿Qué servicios u operaciones
debería proporcionar la clase
de forma natural?

¿Qué datos debe mantener la
clase para proporcionar los
servicios deseados?

Funciones de "lista de
compras" (explicadas a
continuación) para clases
generales de bajo nivel

Funciones limitadas para
clases especializadas

Diseñar pensando en
cómo se utilizará la clase

No importa cómo funciona la
clase o qué datos mantiene.

Lo que importa es qué es lo
que hace la clase

Control de acceso a atributos y métodos

Visibilidad

public

Atributos y métodos que **forman parte de la interfaz** de la clase.

Son los puntos de acceso que otras clases y módulos pueden utilizar para interactuar con la instancia de la clase.

Son accesibles desde cualquier parte del programa. No hay restricciones sobre dónde se pueden usar.

private

Excluye por completo un **atributo o método de la interfaz** de la clase

Se utilizan para encapsular la implementación interna y evitar que el estado interno de la clase sea modificada en forma no controlada desde fuera de la clase.

Estos atributos y métodos son accesibles únicamente desde dentro de la misma clase en la que se declaran.

Ninguna otra clase, incluso las derivadas, puede acceder a estos miembros directamente.

protected

Excluye por completo un **atributo o método de la interfaz** de la clase

Son accesibles desde la misma clase y desde las clases derivadas. No son accesibles desde fuera de la jerarquía de herencia.

Como estos miembros son accesibles desde las clases derivadas, este control permite a las subclases interactuar con estos miembros y extender la funcionalidad de la clase base mientras mantienen el encapsulamiento.

Control de acceso a atributos y métodos

Atributos private

Bus

private:

`cantidadDiesel: double`

public:

`cargarCombustible(_cantidadDiesel: double): void`
`cantidadDiesel += _cantidadDiesel`

Otras clases

`Bus b0 = Bus(...)`

`...`

`b0.cantidadDiesel += carga`

`b0.cargarCombustible(carga)`

Microbus (Bus)

`gastarCombustible(_cantidadDiesel: double): void`

`cantidadDiesel -= _cantidadDiesel`

El objeto es la instancia de la clase.
Desde el punto de vista práctico, el
objeto sólo acceso a la interfaz de
la clase.

Control de acceso a atributos y métodos

Atributos protected

Bus

protected:

cantidadDiesel: double

public:

```
cargarCombustible(_cantidadDiesel: double): void  
    cantidadDiesel += _cantidadDiesel
```



Microbus (Bus)

```
gastarCombustible(_cantidadDiesel: double): void  
    cantidadDiesel -= _cantidadDiesel
```



Otras clases

```
Bus b0 = Bus(...)
```

```
...
```

```
b0.cantidadDiesel += carga
```



```
b0.cargarCombustible(carga)
```



Representación de clases

1	Tareas
2	-lista: Object[0..*] -cantidad: Integer = 0 -personaCargo: String
3	+create() +destroy() +agregar(t: Object): Void +eliminar(t: Object): Void +cantidad(): Integer +vaciar(): Bool +listar(): String +persona(): String +persona(nombre: String): Void

Nombre de la clase

Atributos

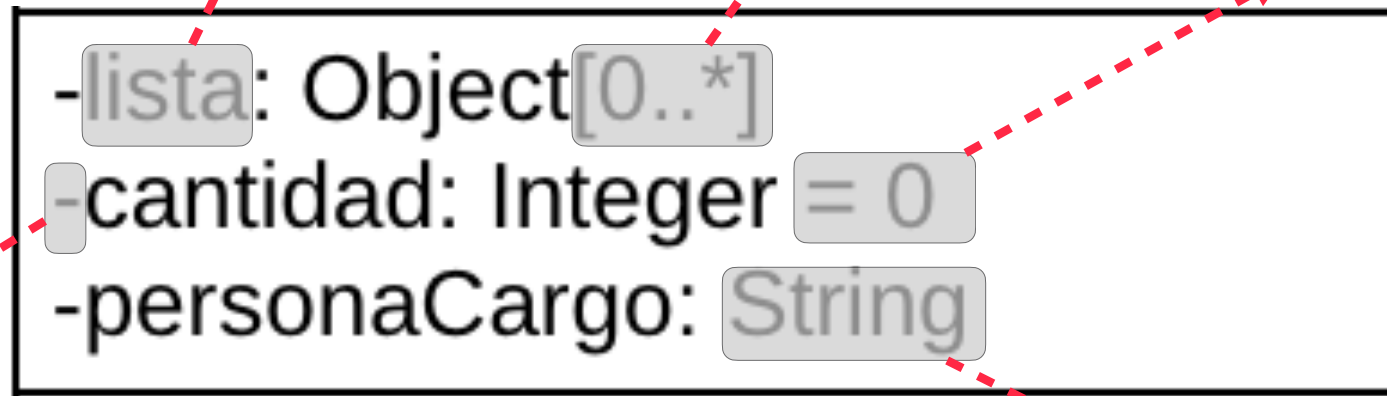
Métodos

Representación de clases

Nombre del atributo

Multiplicidad

Valor por omisión



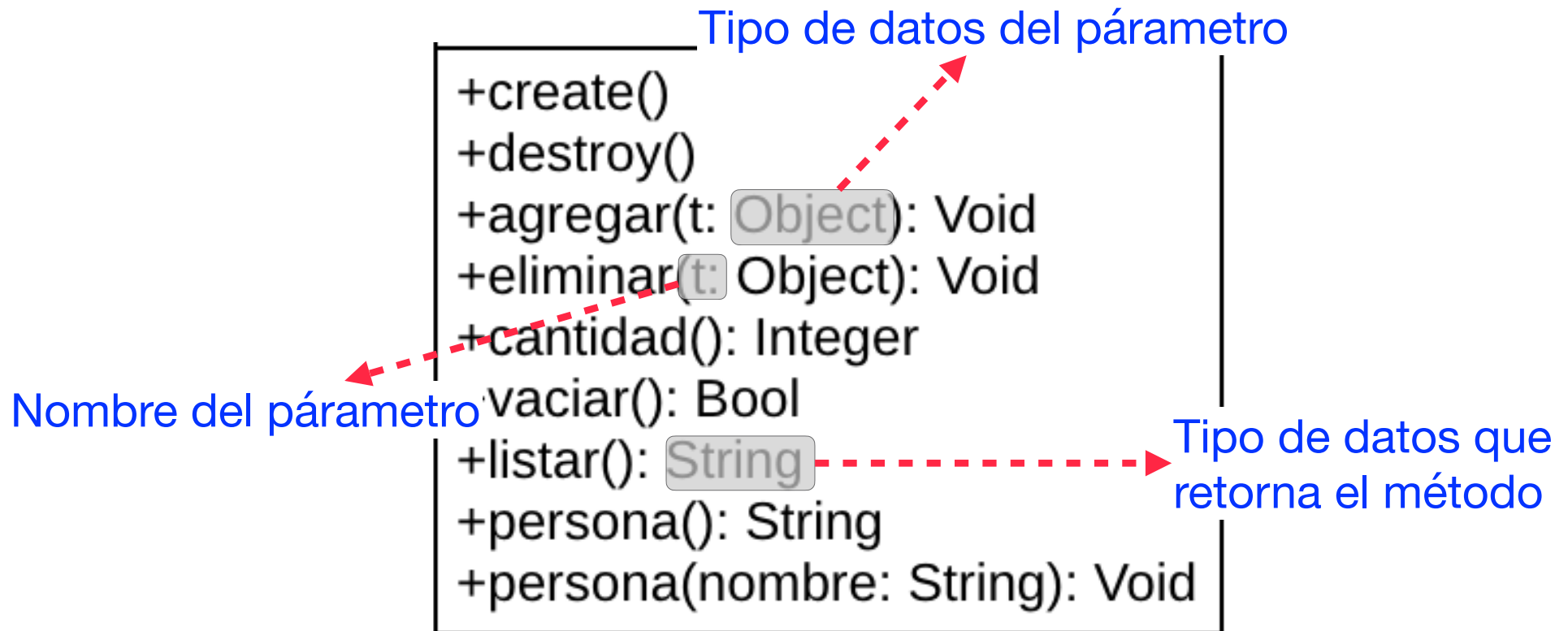
Visibilidad

- + public
- private
- # protected

Tipo de datos

- Boolean
- Integer
- Real
- String
- ...

Representación de clases



Ejemplo de análisis y diseño

Ejemplo de análisis y diseño

Los animales tienen un sonido que cuando lo emiten, se llama **"vocalización"**.

Existen muchos tipos de animales.

Análisis
orientado a
objeto

Entonces todos los animales tienen un acción que se llama **"vocalizar"**.

Pero un animal en concreto vocalizará un sonido que puede no ser igual a otro tipo de animal.

La clase **Animal** se diseña como una clase de nivel superior.

Abstracción

Esta clase tiene un método: **vocalizar()**

Encapsulación

Tiene un atributo que es el sonido que emite el animal: **sonido**

Diagrama de
clases
inicial

Animal

```

sonido: Sonido
create(): Animal
destroy(): Void
vocalizar(): String
  
```

Control de acceso

1. **sonido** se podrá modificar directamente en la misma clase y sus clases derivadas.
2. **vocalizar()** pertenece a la interfaz de la clase, para permitir conocer el sonido del animal desde cualquier parte del programa.

Animal

```

#sonido: Sonido
+create(): Animal
+destroy(): Void
+vocalizar(): String
  
```

Ejemplo de análisis y diseño

De todos los animales, algunos son domesticados por el ser humano y pasan a ser Mascotas. A éstas las personas le ponen un nombre que las identifica

— **Análisis
orientado a
objeto** —→

Aparte de las características de los animales, las mascotas tienen un nombre.

La clase **Mascota** se diseña como una subclase de **Animal**

**Diagrama de clases
simplificado**

Abstracción

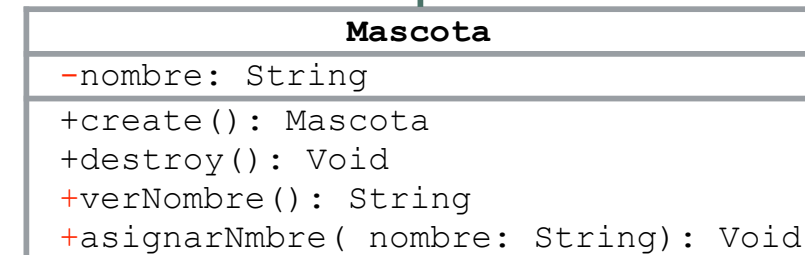
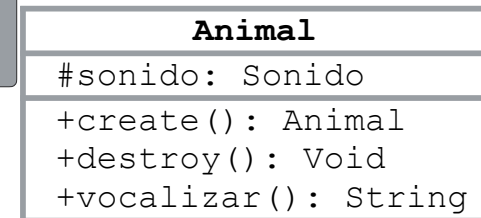
Esta clase se usa a través de:
M.1 **nombre()**: retorna el nombre de la mascota.
M.2 **nombrar()**: Asigna el nombre de la mascota.

Encapsulación

A.1 **nombreMascota**: mantiene el nombre de la mascota

Control de acceso

M.1 y M.2 deben pertenecer a la interfaz. Debido a esto, A.1 debe ser privado.



Ejemplo de análisis y diseño

Existen distinto tipos de mascotas, por ejemplo, perros, gatos y canarios.

Análisis
orientado a
objeto

Cada uno de estas mascotas tienen características en común y otras que son particulares a ellas.

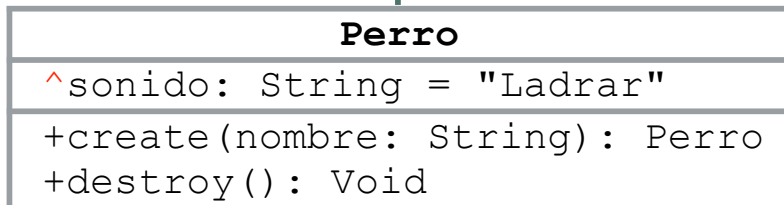
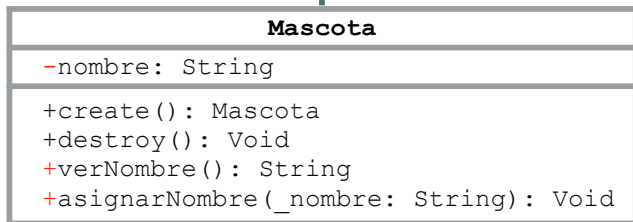
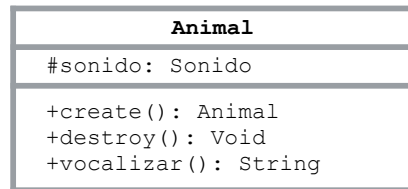
Los distintos tipos de mascotas se diseñan como subclases de **Mascota**

Por ejemplo, la clase **Perro** es derivada a partir de **Mascota**.

Lo único que se sabe de la clase **Perro** es que su sonido se llama "**Ladlar**".

Entones, la clase **Perro** debe especificar que el valor del atributo **_sonido** tiene un valor por omisión, que es "**Ladlar**".

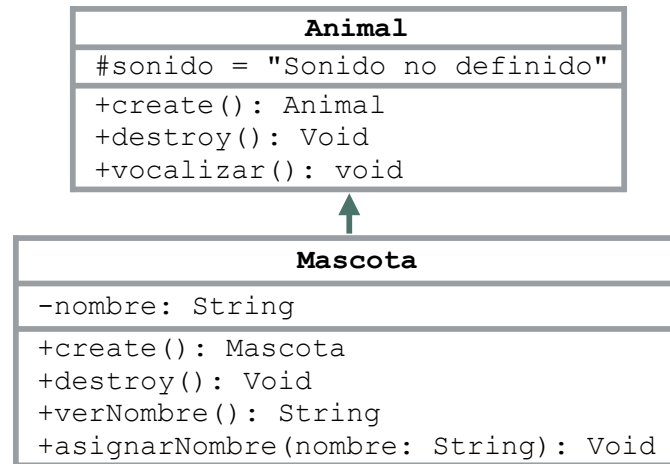
Además, debe dejar claro que **_sonido** es un atributo heredado. Por esta razón, se utiliza el símbolo "^".



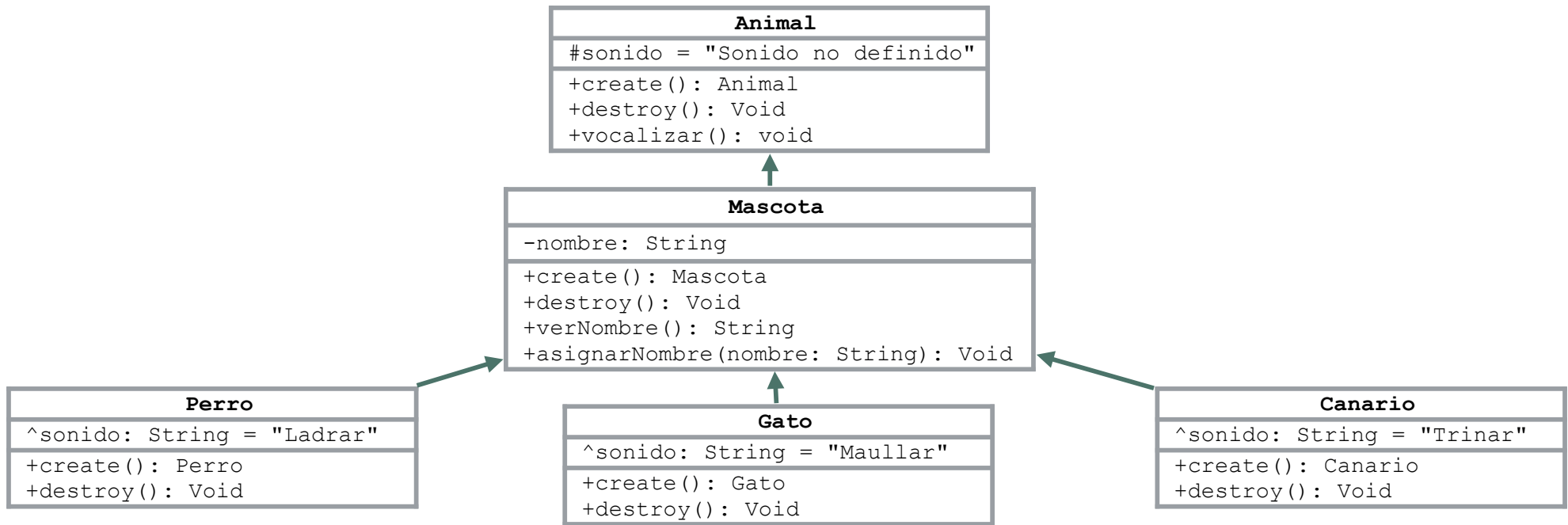
Ejemplo de análisis y diseño

Animal
#sonido = "Sonido no definido"
+create(): Animal +destroy(): Void +vocalizar(): void

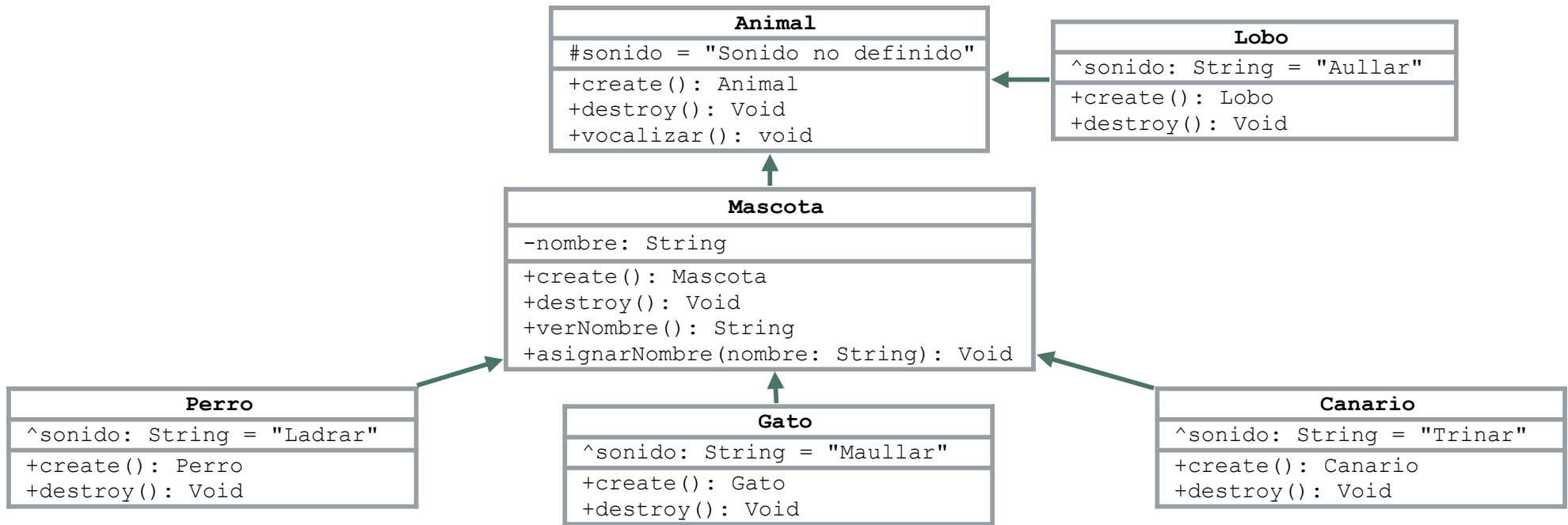
Ejemplo de análisis y diseño



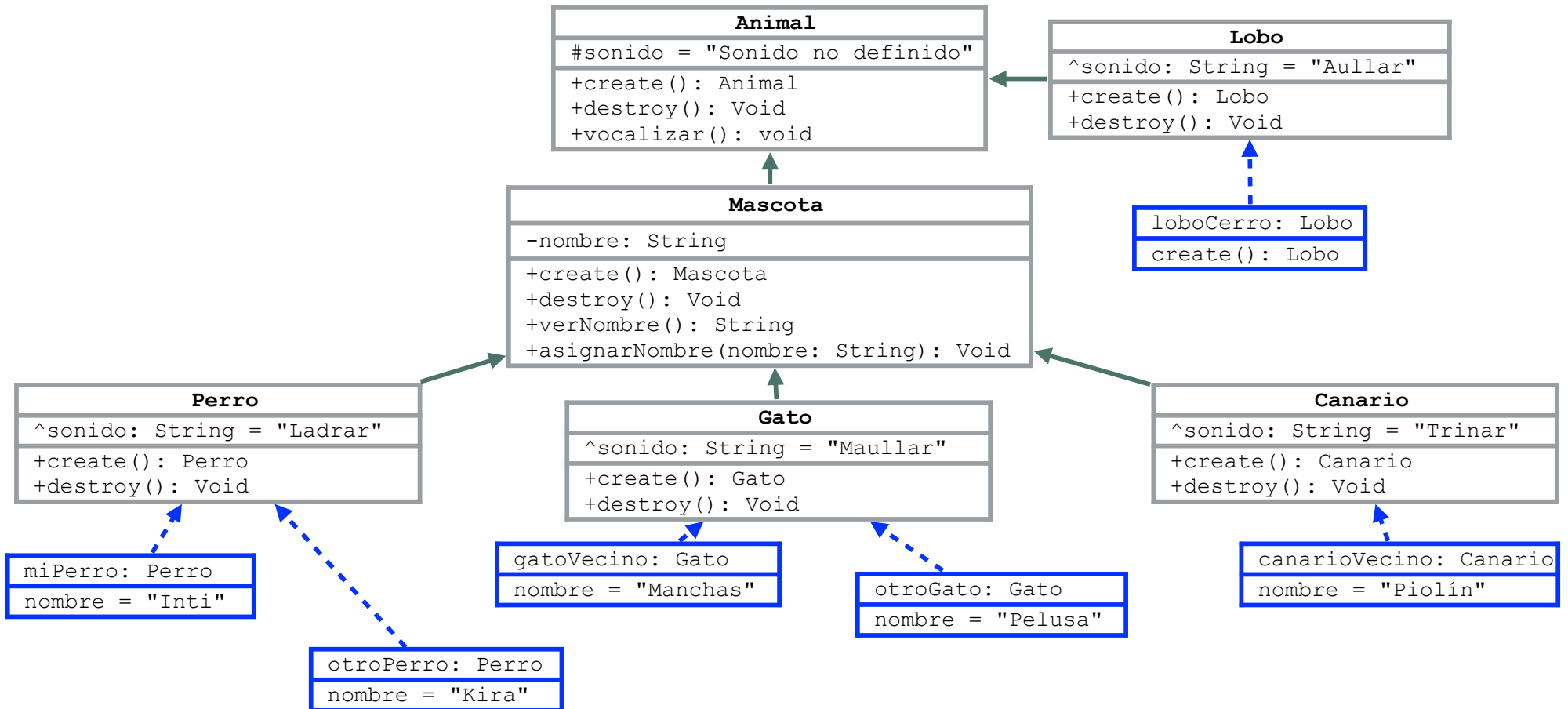
Ejemplo de análisis y diseño



Ejemplo de análisis y diseño



Ejemplo de análisis y diseño



Herencia + Polimorfismo

➡ Métodos virtuales

Métodos virtuales

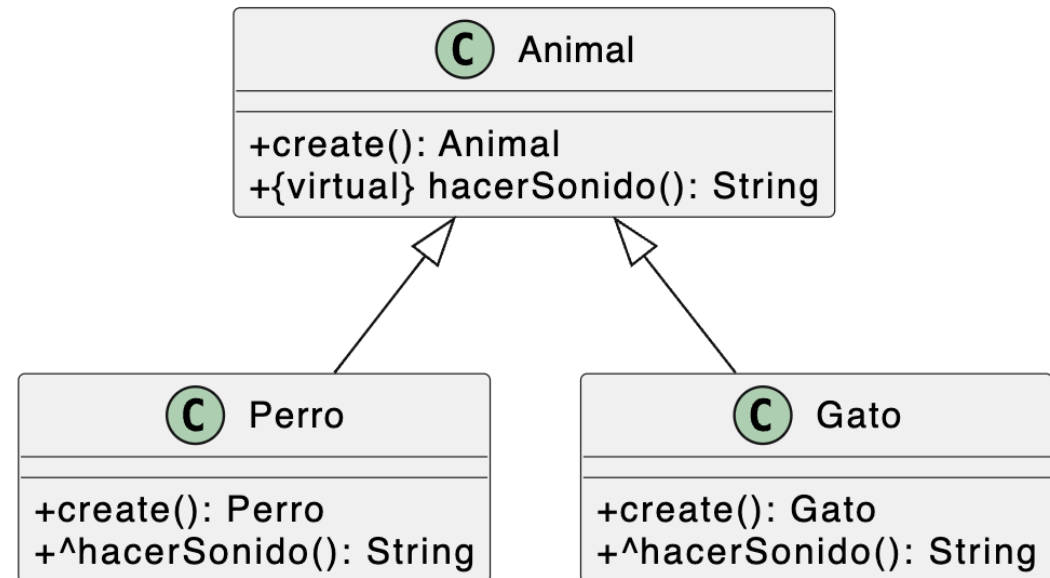
Un método que se declara como virtual en la clase base, lo que indica que puede ser sobrescrito en clases derivadas.

Las clases que heredan de la clase base pueden proporcionar su propia implementación del método virtual.

Permiten que se pueda invocar el método a través de un puntero o referencia de la clase base, y se ejecutará el método de la clase derivada si ha sido sobrescrito.

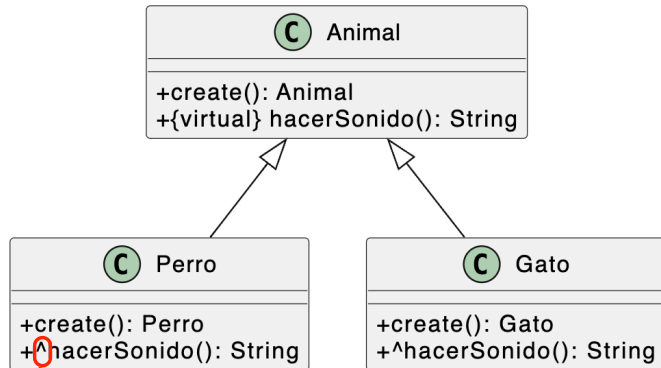


Necesariamente, cuando se necesita polimorfismo en tiempo de ejecución, es necesario trabajar con punteros o referencias a objetos.



El símbolo ^ indica que el método está sobrescrito

Métodos virtuales



```

CLASS Perro:
    INIT_CLASS(): Perro

    DESTROY_CLASS():

    OVERRIDE STRING hacerSonido():
        return "Guau"
  
```

```

CLASS Animal:
    INIT_CLASS(): Animal

    DESTROY_CLASS():

    VIRTUAL STRING hacerSonido():
        return "Sonido animal"
  
```

```

CLASS Gato:
    INIT_CLASS(): Gato

    DESTROY_CLASS():

    OVERRIDE STRING hacerSonido():
        return "Miau"
  
```

La palabra **OVERRIDE** indica que el método está sobrescrito

```

Animal p0 = MAKE_OBJECT Perro()
Animal p1 = MAKE_OBJECT Gato()
  
```

```

print(p0.hacerSonido)
print(p1.hacerSonido)
  
```



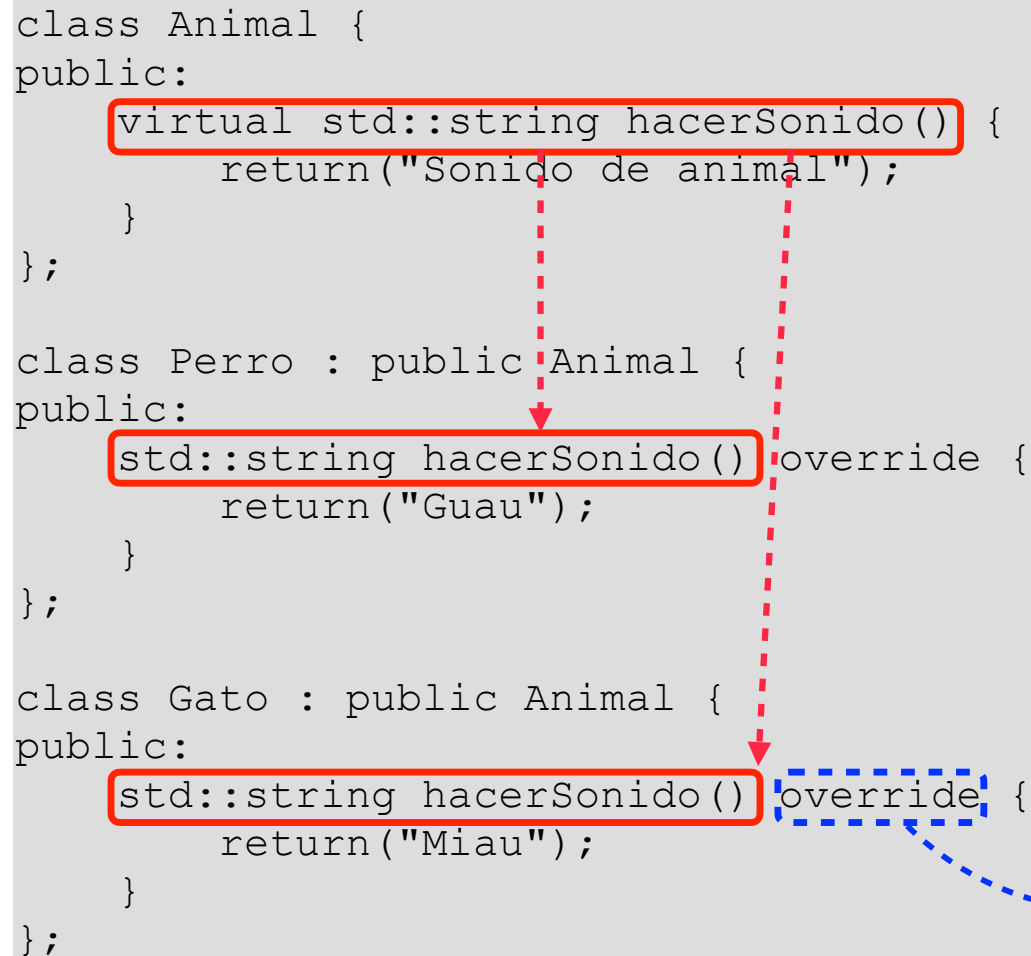
```

Guau
Miau
  
```

Salida

Métodos virtuales (C++)

```
class Animal {  
public:  
    virtual std::string hacerSonido() {  
        return("Sonido de animal");  
    }  
};  
  
class Perro : public Animal {  
public:  
    std::string hacerSonido() override {  
        return("Guau");  
    }  
};  
  
class Gato : public Animal {  
public:  
    std::string hacerSonido() override {  
        return("Miau");  
    }  
};
```



Un método declarado como virtual permite implementar polimorfismo en tiempo de ejecución.

C++ necesita declarar explícitamente los métodos que se sobre-escriben en las clases derivadas.

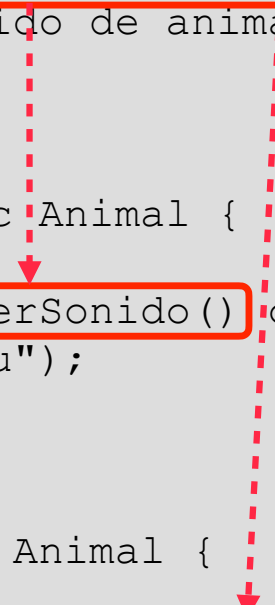
override es un "adorno". Es complementario al código.

Métodos virtuales (C++)

```
class Animal {
public:
    virtual std::string hacerSonido() {
        return("Sonido de animal");
    }
};

class Perro : public Animal {
public:
    std::string hacerSonido() override {
        return("Guau");
    }
};

class Gato : public Animal {
public:
    std::string hacerSonido() override {
        return("Miau");
    }
};
```



```
Animal p0 = Perro();
Animal p1 = Gato();
```

```
std::cout << p0.hacerSonido() << "\n";
std::cout << p1.hacerSonido() << "\n";
```

Salida

```
Sonido de animal
Sonido de animal
```

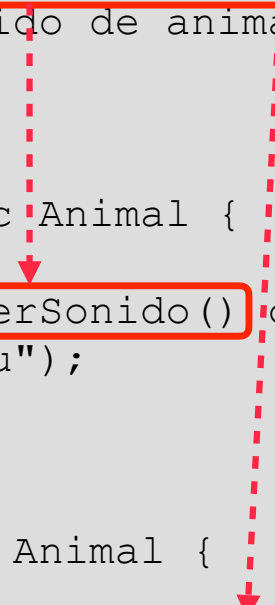
En este ejemplo, C++ no puede implementar correctamente la sobrecarga de métodos ya que **p0** y **p1** son objetos estáticos de tipo **Animal**.

Métodos virtuales (C++)

```
class Animal {
public:
    virtual std::string hacerSonido() {
        return("Sonido de animal");
    }
};

class Perro : public Animal {
public:
    std::string hacerSonido() override {
        return("Guau");
    }
};

class Gato : public Animal {
public:
    std::string hacerSonido() override {
        return("Miau");
    }
};
```



```
Animal* p0 = new Perro();
Animal* p1 = new Gato();
```

```
std::cout << p0.hacerSonido() << "\n";
std::cout << p1.hacerSonido() << "\n";
```

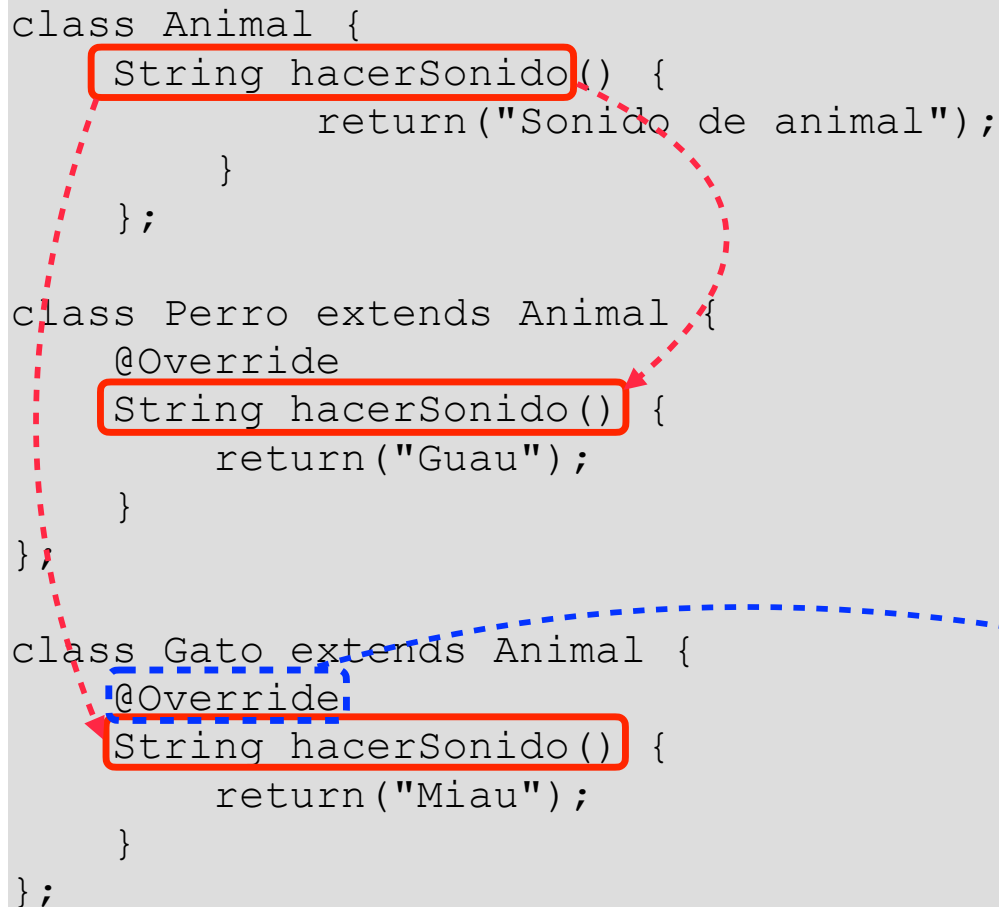
Salida

```
Guau
Miau
```

Con punteros (que es un tipo de referencia), C++ implementa correctamente el polimorfismo en tiempo de ejecución.

Métodos virtuales (java)

```
class Animal {  
    String hacerSonido() {  
        return("Sonido de animal");  
    }  
};  
  
class Perro extends Animal {  
    @Override  
    String hacerSonido() {  
        return("Guau");  
    }  
};  
  
class Gato extends Animal {  
    @Override  
    String hacerSonido() {  
        return("Miau");  
    }  
};
```



En Java, los métodos en la clases son virtuales por omisión. Un objeto es una referencia a su clase.

```
Animal a0 = new Perro();  
Animal a1 = new Gato();  
  
System.out.println(a0.hacerSonido());  
System.out.println(a1.hacerSonido());
```

Salida

```
Guau  
Miau
```

@Override es un "adorno". Es complementario al código.

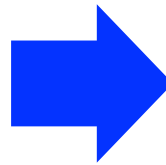
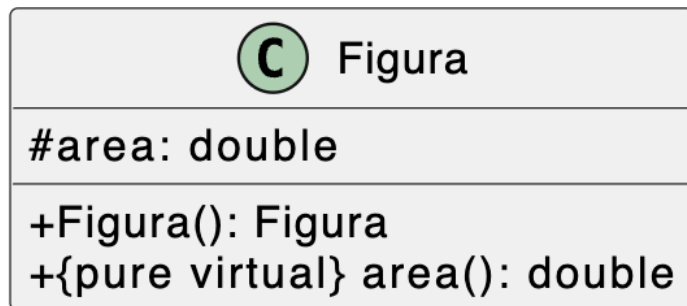
Métodos virtuales sin implementación

➔ Métodos virtuales puros

Métodos virtuales puros

Un método virtual puro es una función miembro de una clase base que **no tiene implementación** en la clase base y **debe** ser implementada en las clases derivadas.

Diagrama de clases



Pseudo-Código

```
CLASS Figura {  
protected:  
    area: double  
public:  
    INIT_CLASS(): Figura  
    DESTROY_CLASS():  
  
    PURE VIRTUAL double area()  
}
```

Métodos virtuales puros

```

CLASS Figura {
private:
    id: string
protected:
    area_: double
public:
    INIT_CLASS(): Figura
    INIT_CLASS(id: string): Figura

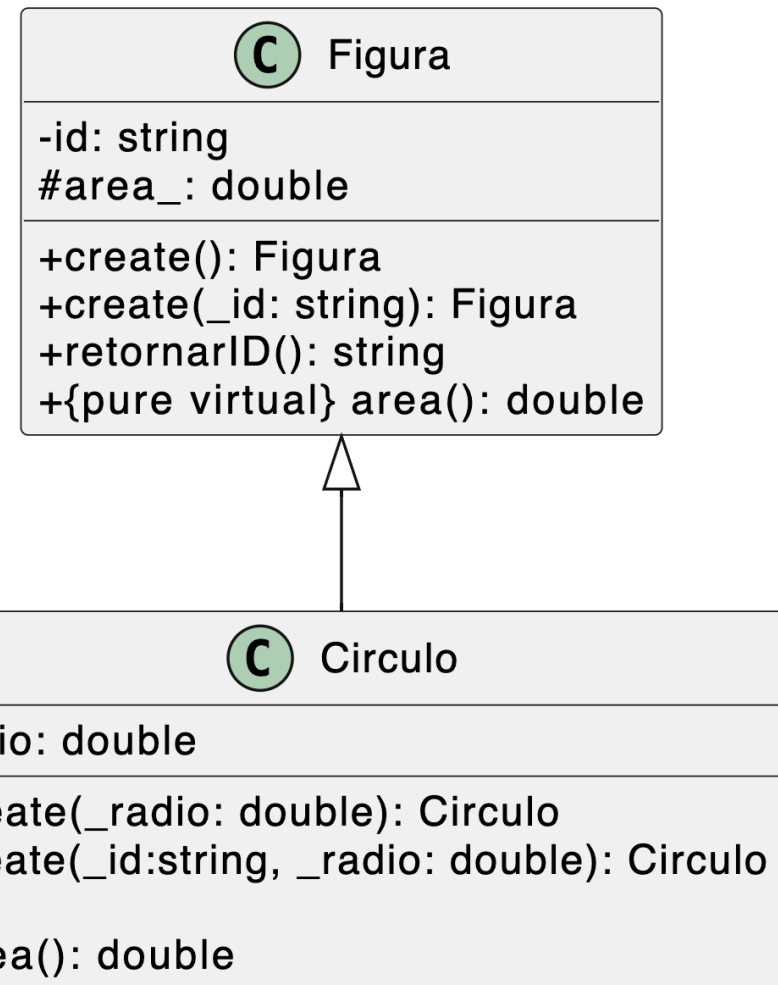
    retornarID(): string

    PURE VIRTUAL double area()
    ...métodos virtuales...
    ...otros métodos...
}
    
```

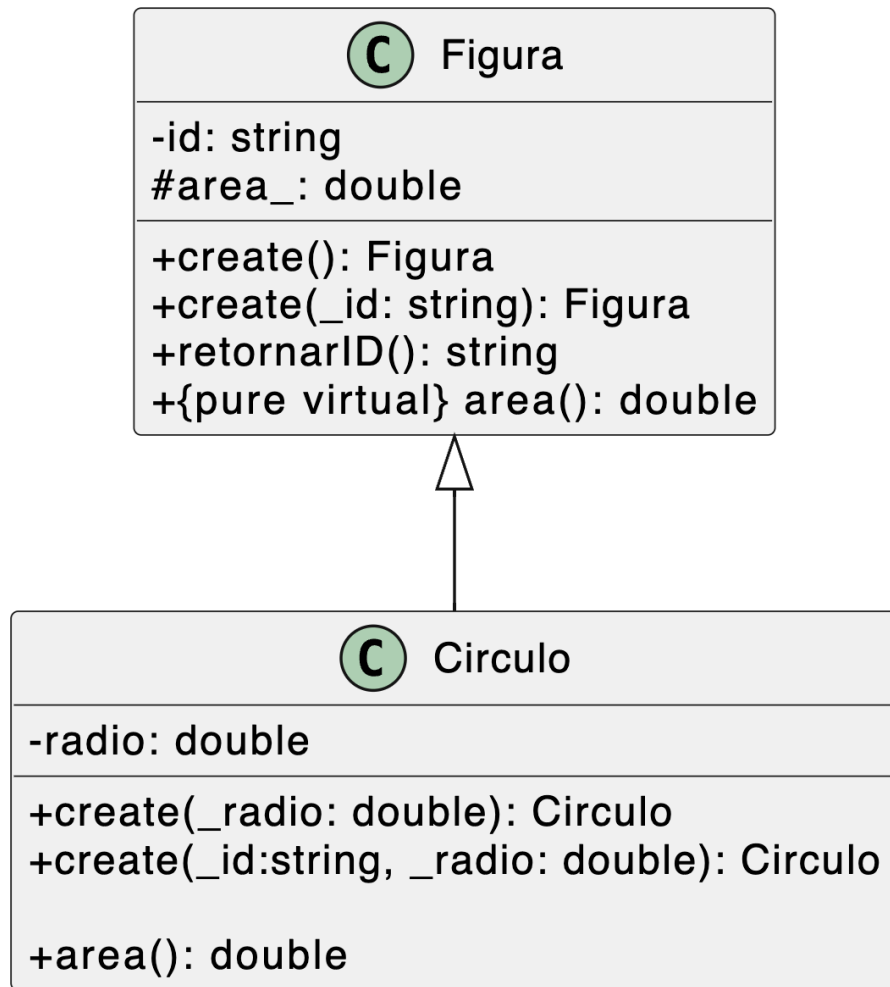
```

CLASS Circulo(Figura) {
private:
    radio: double
public:
    INIT_CLASS(_radio: double): Circulo
    INIT_CLASS(_id: string, _radio: double): Circulo

    OVERRIDE double area()
}
    
```



Métodos virtuales puros C++



```

class Figura {
private:
    std::string id;

protected:
    double area_;

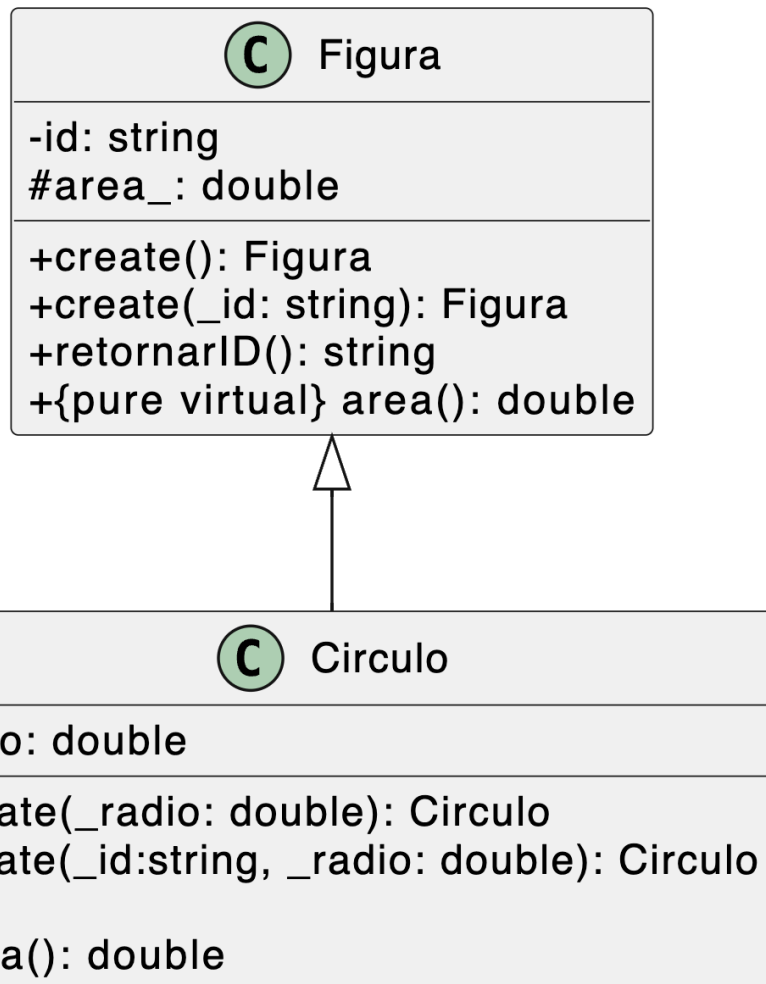
public:
    Figura() {
        id="";
    }

    Figura(std::string _id) {
        id = _id;
    }

    std::string retornarID() {
        return(_id);
    }

    virtual double area() = 0;
}
    
```

Métodos virtuales puros C++



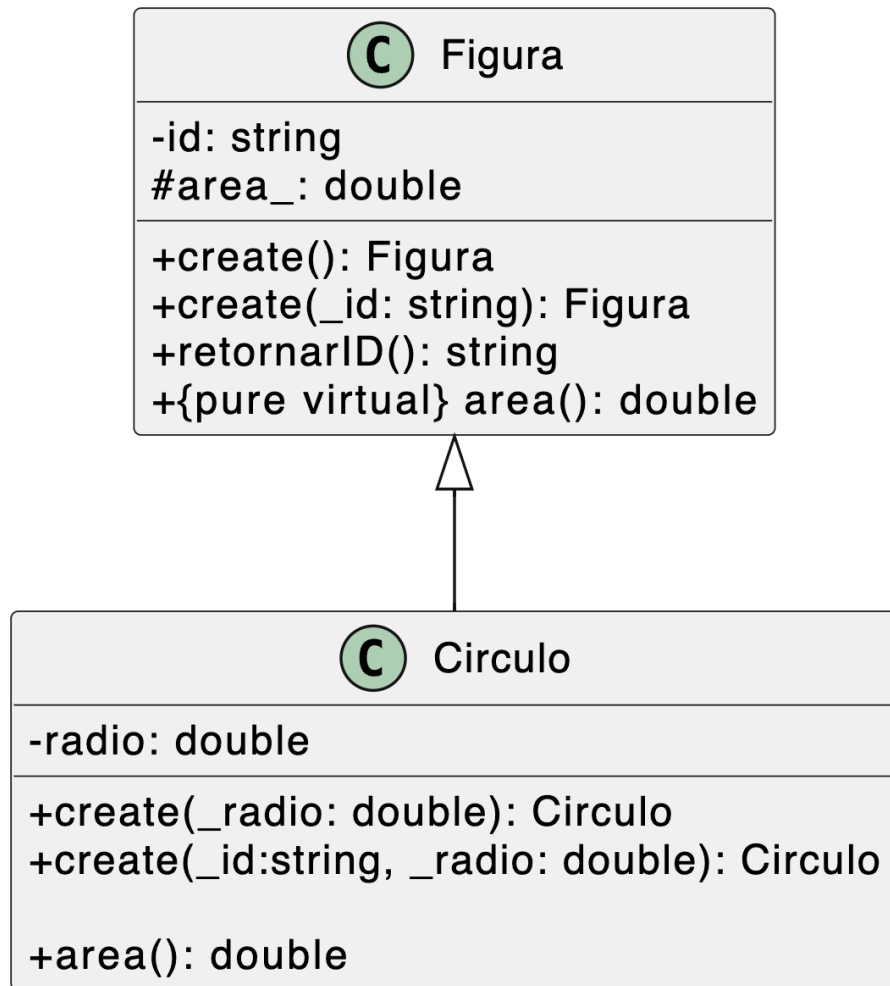
```

class Circulo : public Figura {
    double radio;
public:
    Circulo(double _radio){
        radio = _radio
        area = radio*radio*3.1415
    }

    Circulo(std::string _id, double _radio):Circulo(_id){
        radio = _radio
        area_ = radio*radio*3.1415
    }

    double area() override {
        return(area_);
    }
};
    
```

Métodos virtuales puros Java



Java no permite la existencia de clases que mezclen métodos virtuales puros con otros métodos. Java requiere que se ajuste el diseño de la jerarquía para poder implementarla.

Esto se verá en el capítulo de implementación de clases en Java.

