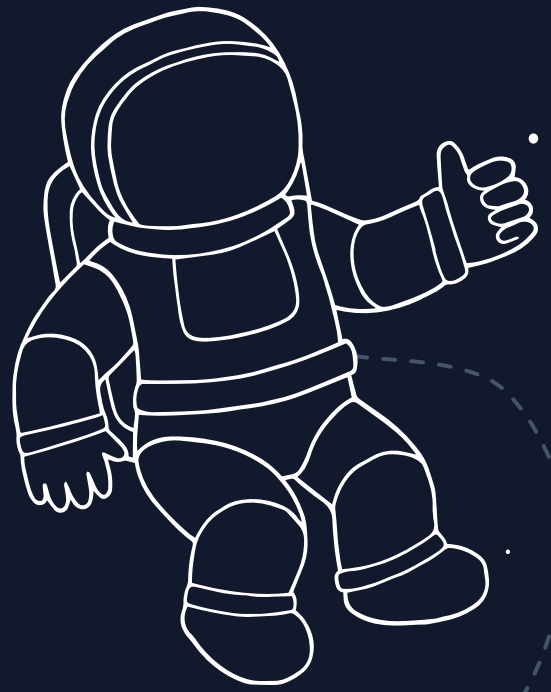
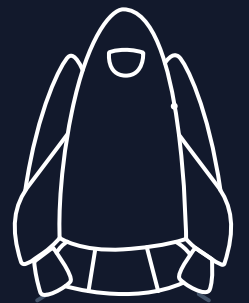


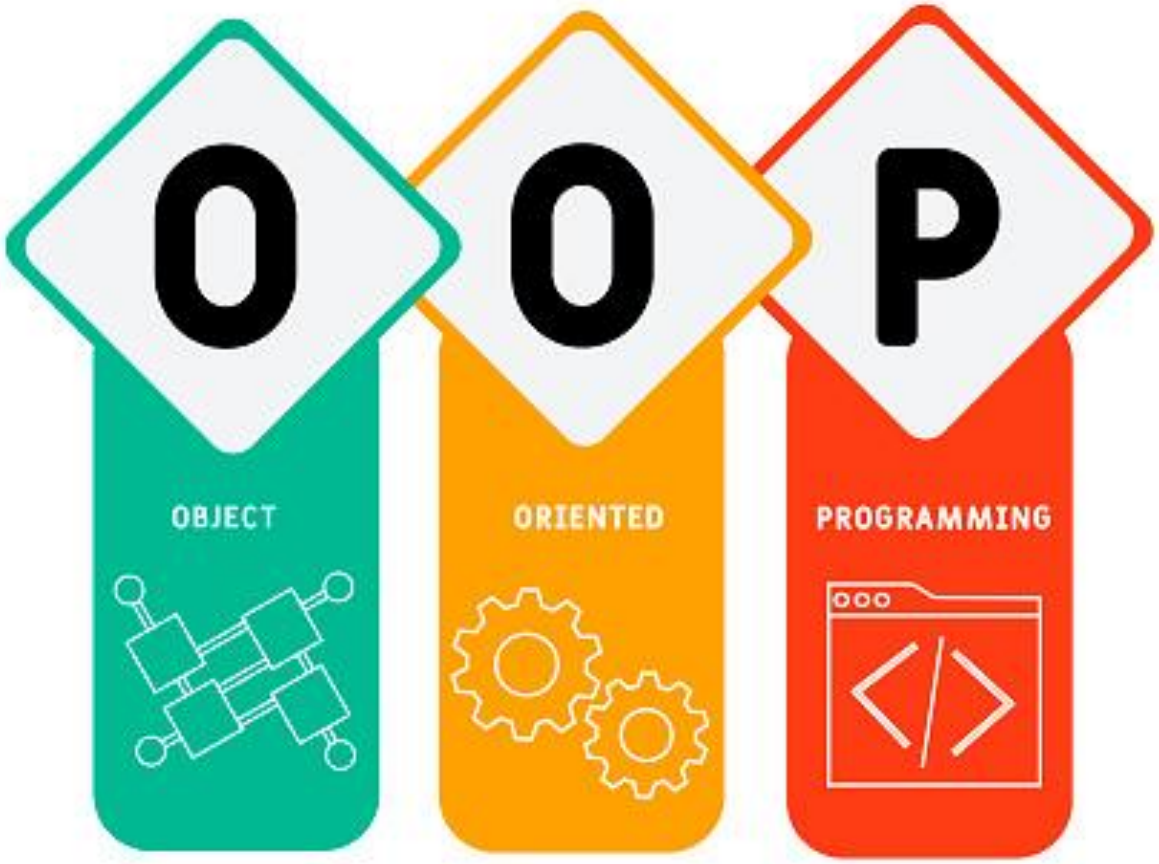
Programa académico CAMPUS

MODULO JAVA
Sesión 6
Herencia y Polimorfismo

Trainer Carlos H. Rueda C.

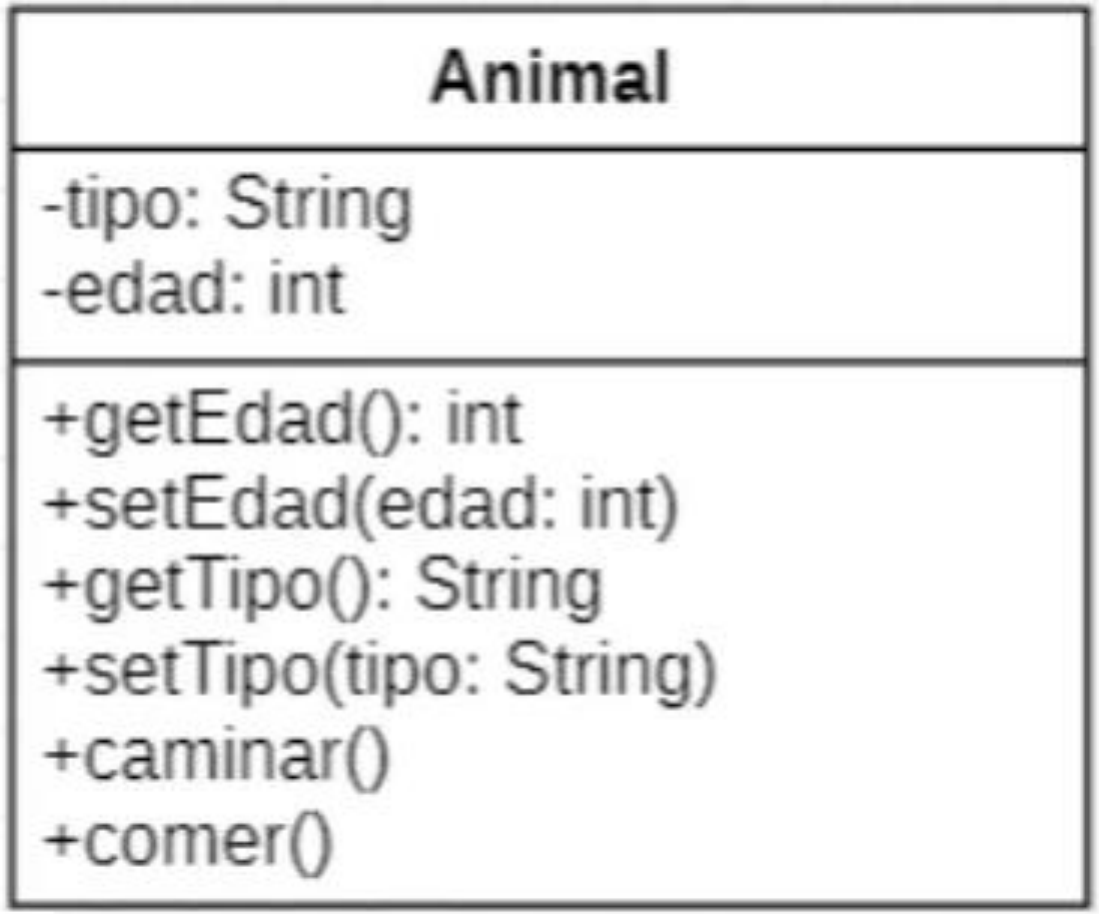


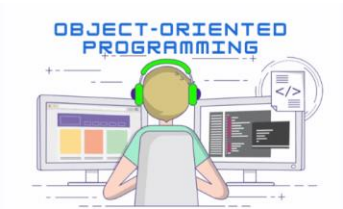

Java™


O
O
P
OBJECT
ORIENTED
PROGRAMMING

LOS 4 PILARES DE LA POO

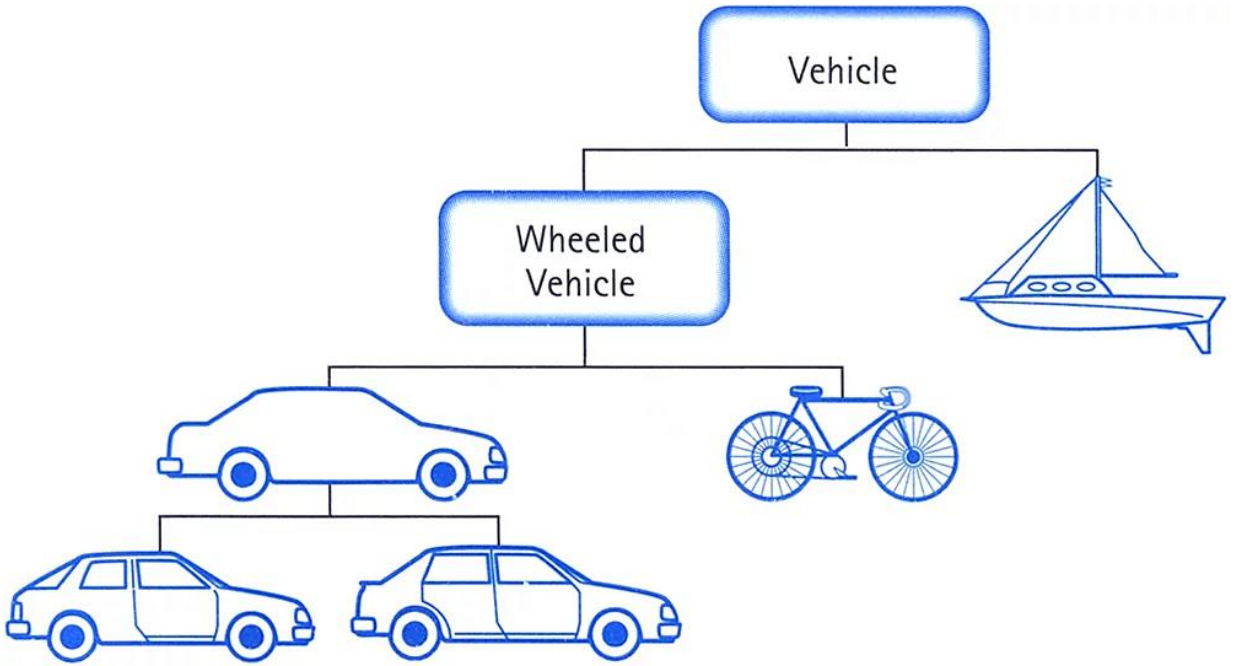
Diagrama de clases





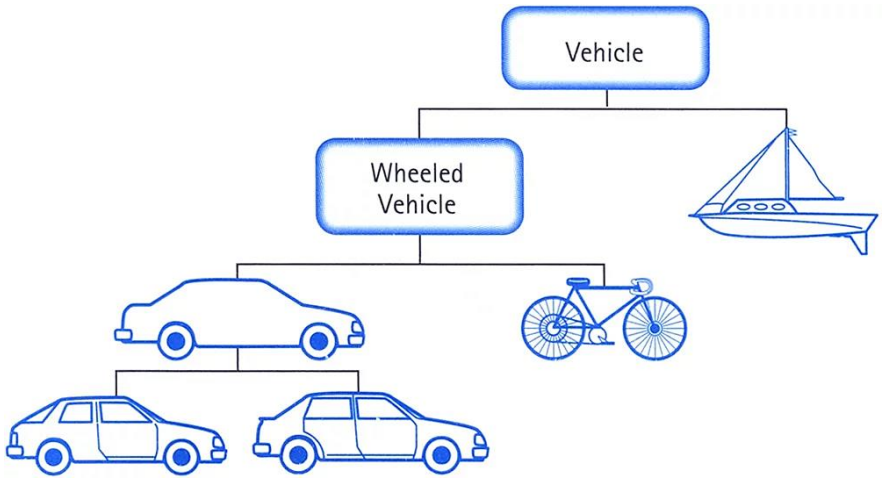
Herencia

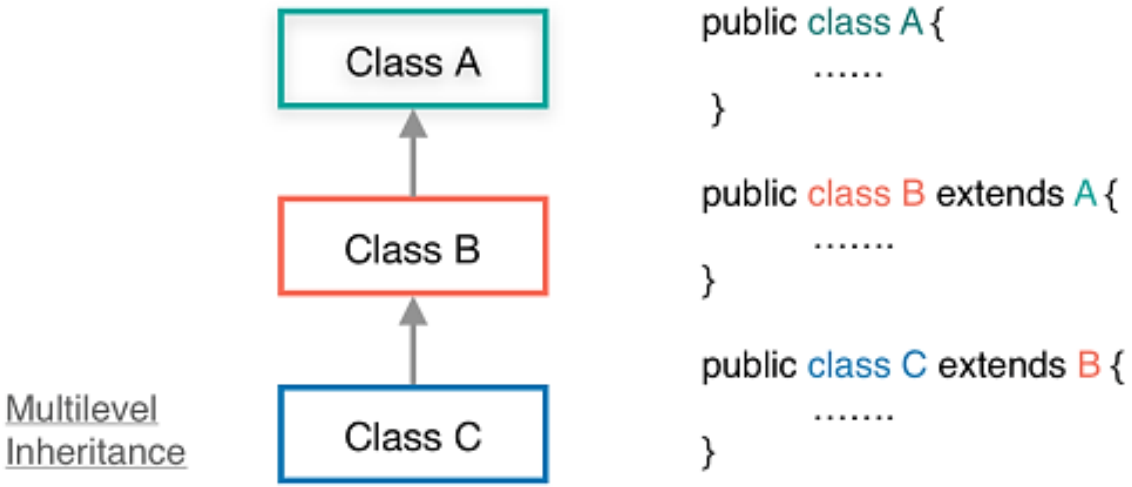
La herencia en programación orientada a objetos (POO) es un mecanismo fundamental que permite a una clase heredar las características (atributos y métodos) de otra clase. No se refiere a la herencia de bienes materiales, sino a la capacidad de una clase de aprovechar y extender el código existente. La herencia en POO permite la creación de nuevas clases basadas en clases existentes para reutilizar el código y establecer una jerarquía de clases en una aplicación. Una clase hija hereda los atributos y métodos de su clase padre, y además puede agregar nuevos atributos, métodos o redefinir los heredados.



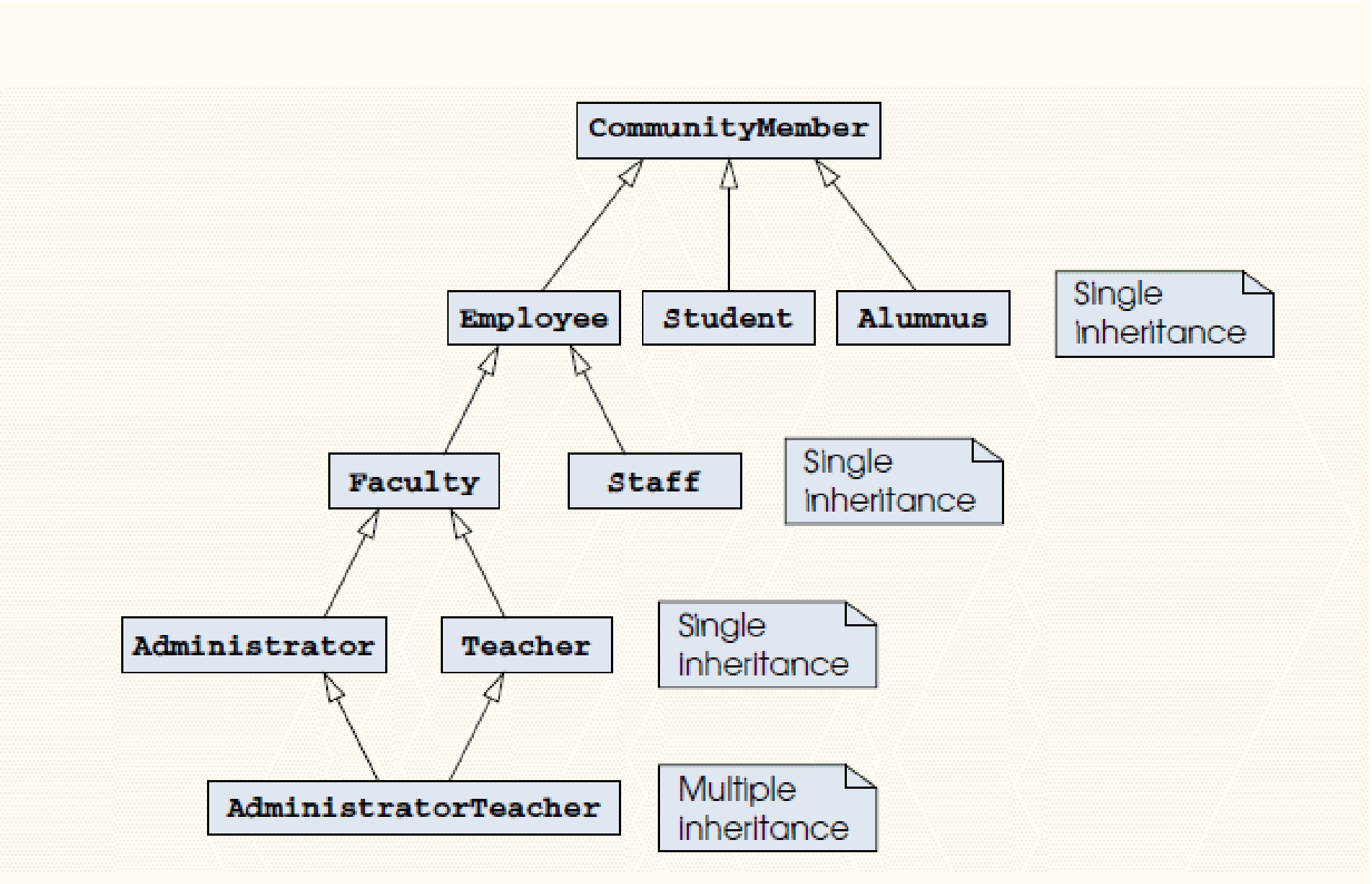
Herencia

En Java, la herencia se logra mediante la extensión de una superclase, también conocida como clase padre, con una subclase o clase secundaria. Esto se hace utilizando la palabra clave `extends`. Si no se declara una superclase específica, la clase se extiende implícitamente de la clase `Object`. La clase `Object` es la raíz de todas las jerarquías de herencia en Java y es la única clase que no se extiende de otra clase.



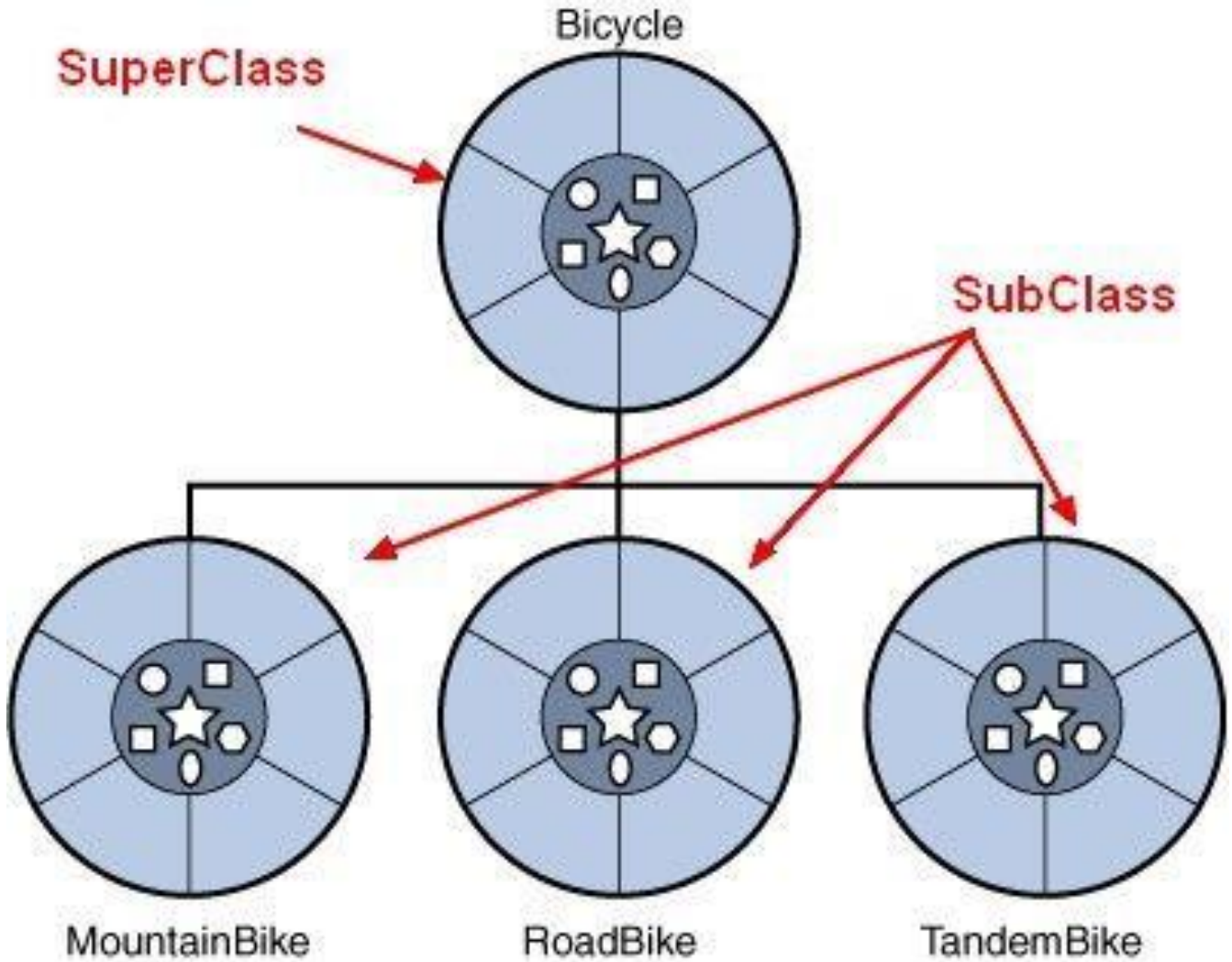


Herencia


CommunityMember
Employee
Student
Alumnus
Single inheritance
Faculty
Staff
Single inheritance
Administrator
Teacher
Single inheritance
AdministratorTeacher
Multiple inheritance

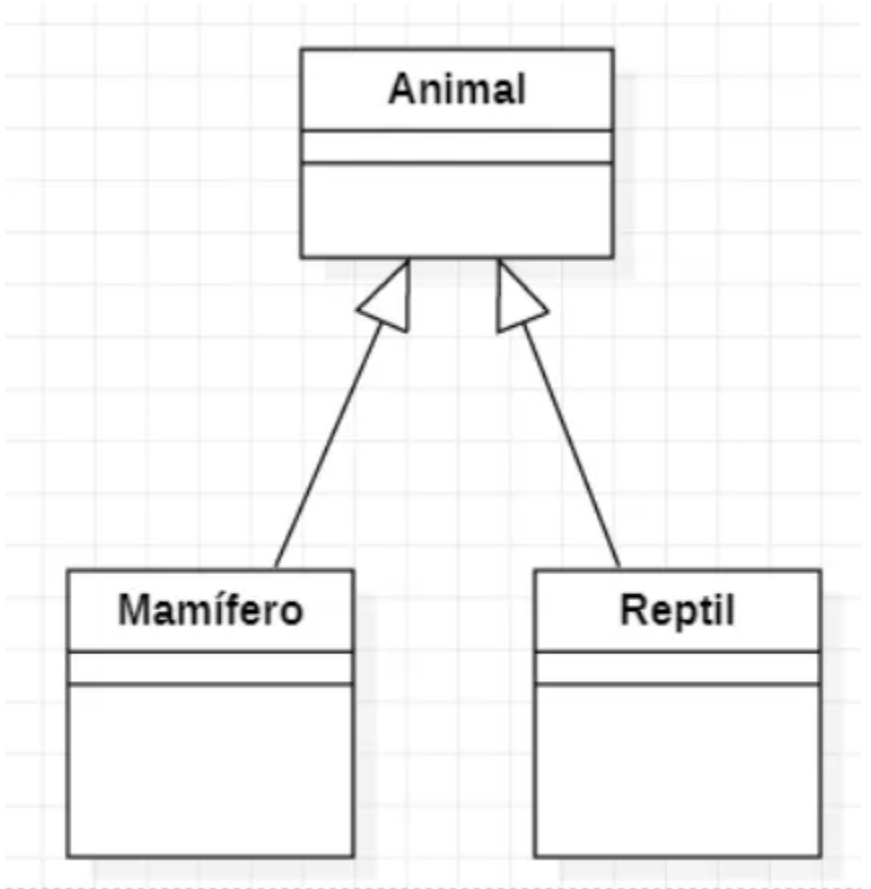

OBJECT-ORIENTED PROGRAMMING

Herencia

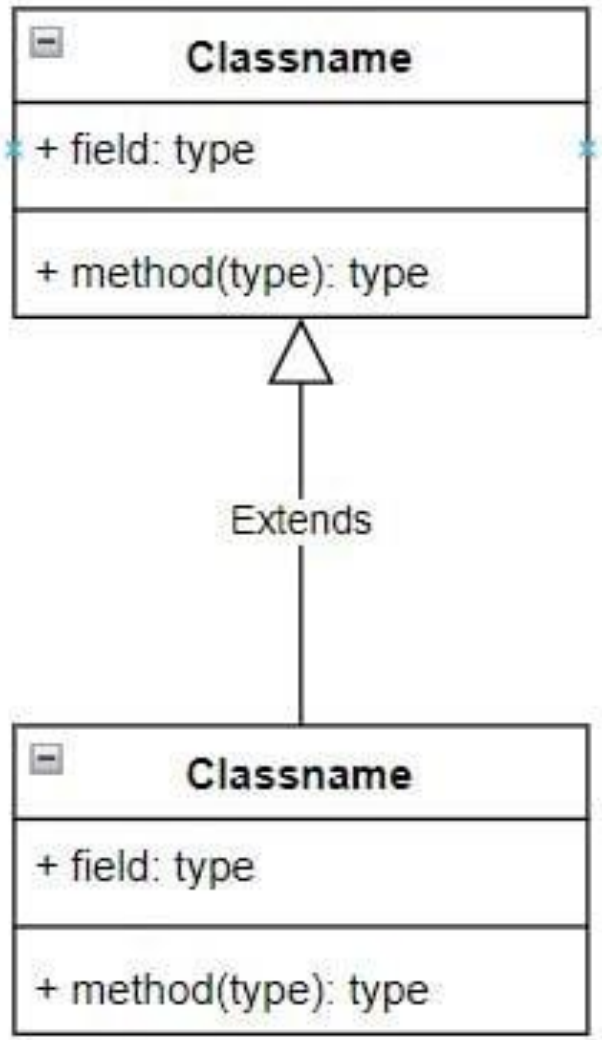

Bicycle
SuperClass
SubClass
MountainBike
RoadBike
TandemBike


OBJECT-ORIENTED
PROGRAMMING

Herencia

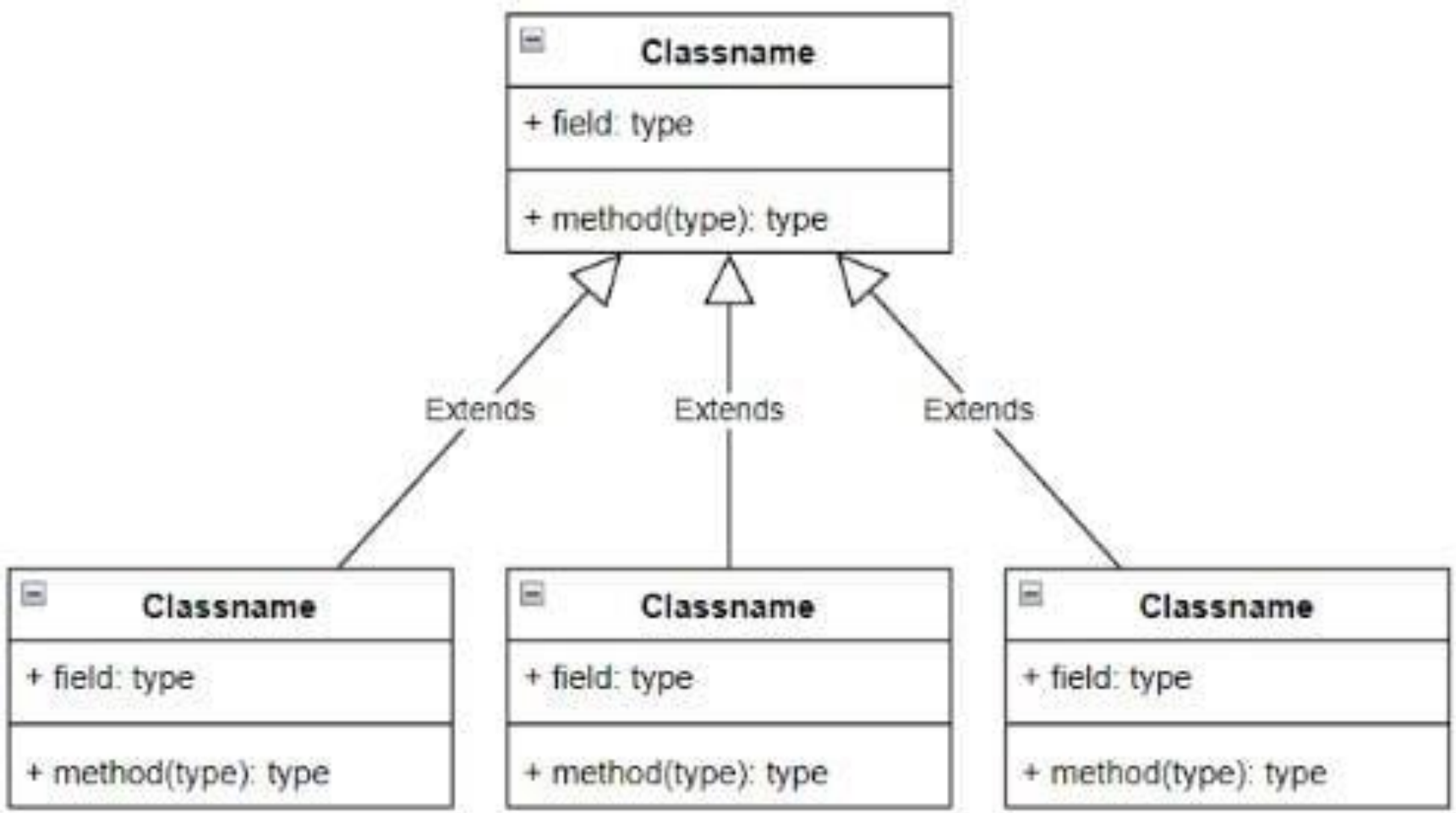


Herencia


Classname
+ field: type
+ method(type): type
Extends
Classname
+ field: type
+ method(type): type

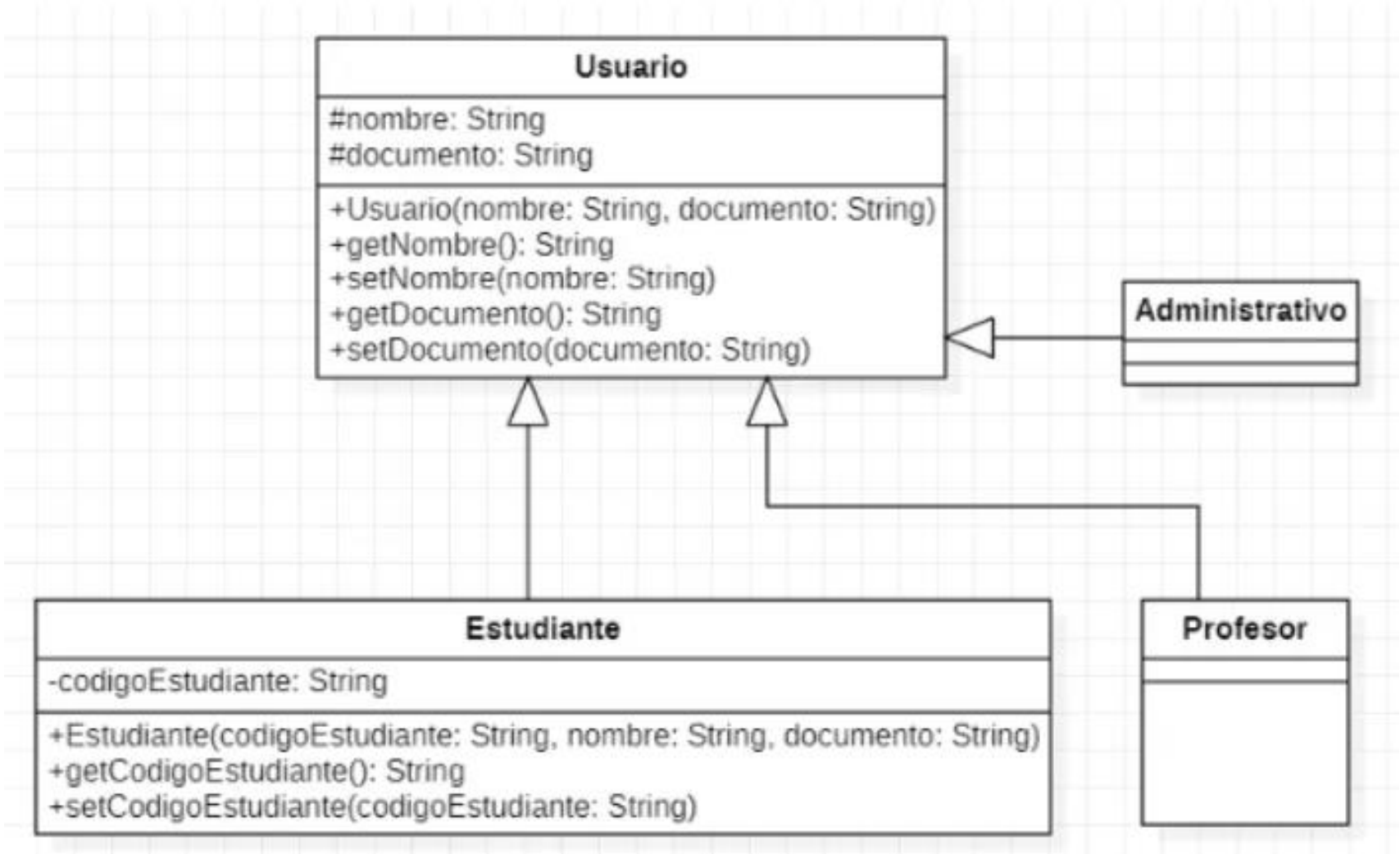

OBJECT-ORIENTED
PROGRAMMING

Herencia


Classname
+ field: type
+ method(type): type
Extends
Extends
Extends
Classname
+ field: type
+ method(type): type
Classname
+ field: type
+ method(type): type
Classname
+ field: type
+ method(type): type


OBJECT-ORIENTED
PROGRAMMING

Herencia


Usuario
#nombre: String
#documento: String
+Usuario(nombre: String, documento: String)
+getNombre(): String
+setNombre(nombre: String)
+getDocumento(): String
+setDocumento(documento: String)
Administrativo
Estudiante
-codigoEstudiante: String
+Estudiante(codigoEstudiante: String, nombre: String, documento: String)
+getCodigoEstudiante(): String
+setCodigoEstudiante(codigoEstudiante: String)
Profesor


OBJECT-ORIENTED
PROGRAMMING

Herencia

```java
public class Usuario {
    protected String nombre;
    protected String documento;

    public Usuario (String nombre, String documento) {
        this.nombre = nombre;
        this.documento = documento;
    }

    public String getNombre() {
        return nombre;
    }

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

    public String getDocumento() {
        return documento;
    }

    public void setDocumento(String documento) {
        this.documento = documento;
    }
}
```

Herencia

```java
public class Estudiante extends Usuario{
    private String codigoEstudiante;

    public Estudiante (String codigoEstudiante, String nombre, String documento) {
        super(nombre, documento);
        this.codigoEstudiante = codigoEstudiante;
    }

    public String getCodigoEstudiante() {
        return codigoEstudiante;
    }

    public void setCodigoEstudiante(String codigoEstudiante) {
        this.codigoEstudiante = codigoEstudiante;
    }
}
```

Herencia

```
public class MiPrimerProyecto {
    public static void main(String[] args) {
        Estudiante estudiante1 = new Estudiante(
            "123456",
            "Pepito Perez",
            "1098000000");
        estudiante1.getNombre();
    }
}
```

this y super

En Java, la palabra clave "`this`" se utiliza para hacer referencia al objeto actual en el que se está ejecutando un método o constructor. Tiene varios usos y funciones dentro de una clase:

```
public class MiClase {
    private int x; // variable de instancia

    public void setX(int x) {
        this.x = x; // "this" se refiere a la variable de instancia
    }

}
```

this y super

Invocar a otro constructor de la misma clase: Dentro de un constructor, "`this`" se puede utilizar para llamar a otro constructor de la misma clase. Esto se conoce como "sobrecarga de constructores" y es útil para evitar duplicación de código cuando hay múltiples constructores en una clase.

```
public class Persona {
    private String nombre;
    private int edad;

    public Persona(){
        this("Nombre por defecto", 0); //Llama al otro
        //constructor con valores por defecto
    }

    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

}
```

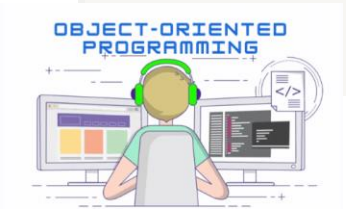
this y super

Pasar el objeto actual como argumento: A veces es necesario pasar el objeto actual a otro método. Usando "`this`", se puede hacer de manera explícita.

Java

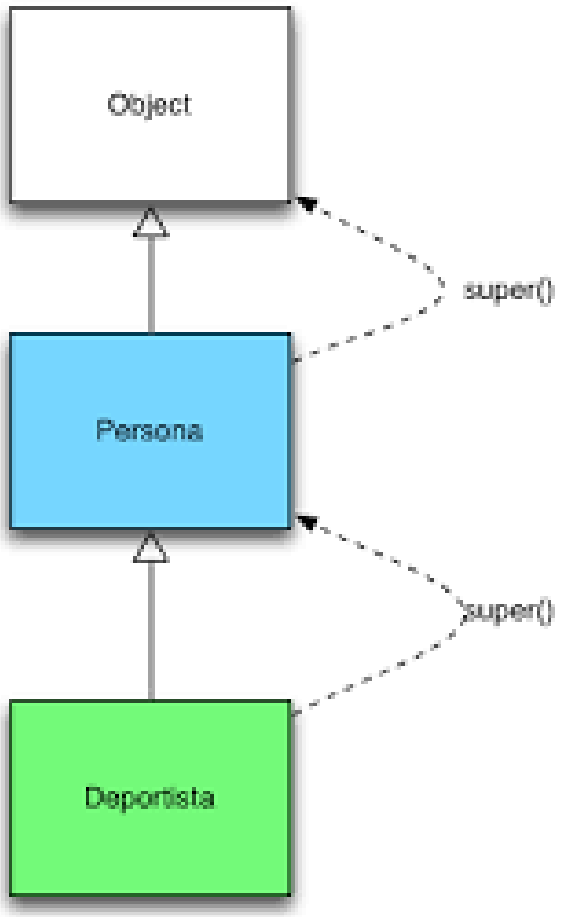
Copiar

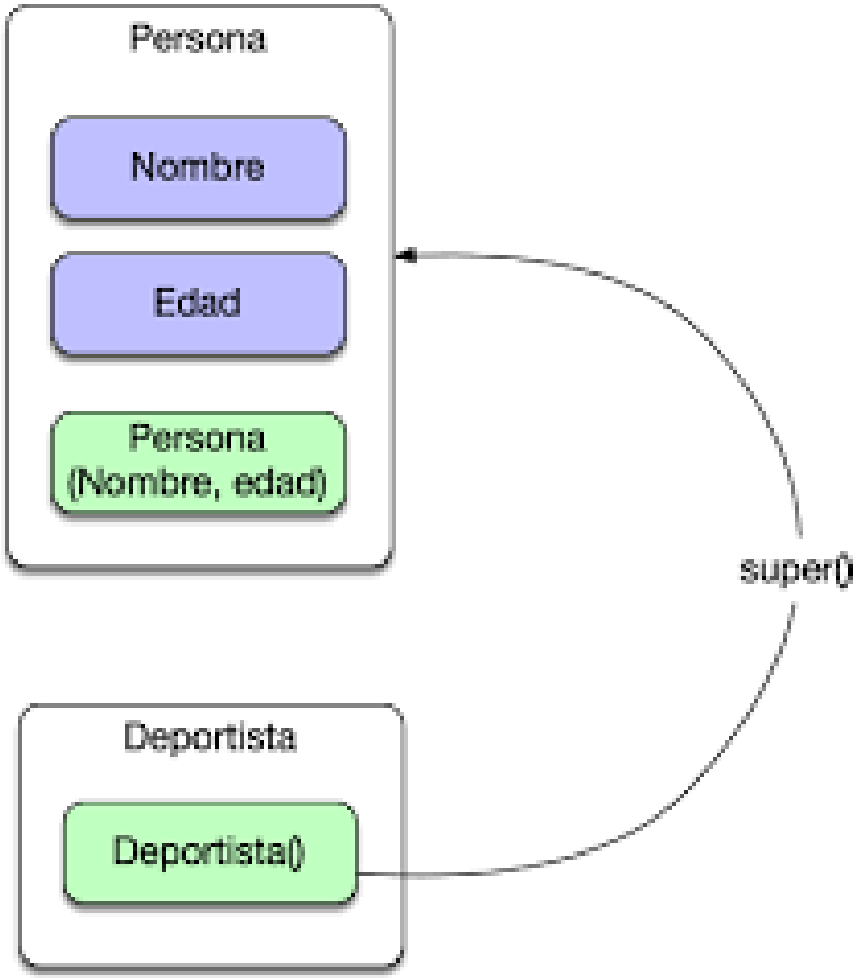
```java
public class MiClase {
    public void metodo1() {
        metodo2(this);
        //Pasando el objeto actual como argumento a metodo2
    }

    public void metodo2(MiClase objeto) {
        // Código para utilizar el objeto recibido como argumento
    }
```



this y super

Por otra parte, En Java, la palabra clave "`super`" se utiliza para hacer referencia a la clase padre o superclase desde una subclase. Tiene varios usos importantes dentro de una jerarquía de herencia:

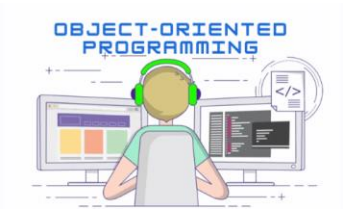




this y super

1. Invocar al constructor de la clase padre: Cuando se crea una subclase, el constructor de la superclase debe ser invocado antes de que se inicialicen los miembros específicos de la subclase. Para hacer esto, se utiliza "`super`" para llamar al constructor de la clase padre.

```
public class SubClase extends SuperClase {
    public SubClase() {
        super(); //Llama al constructor de la clase padre (superClase)
        // Código para inicializar miembros específicos de la subclase
    }

}
```



this y super

2 Acceder a miembros de la clase padre: "super" también se utiliza para acceder a los miembros (atributos o métodos) de la superclase, en caso de que tengan el mismo nombre que los miembros de la subclase

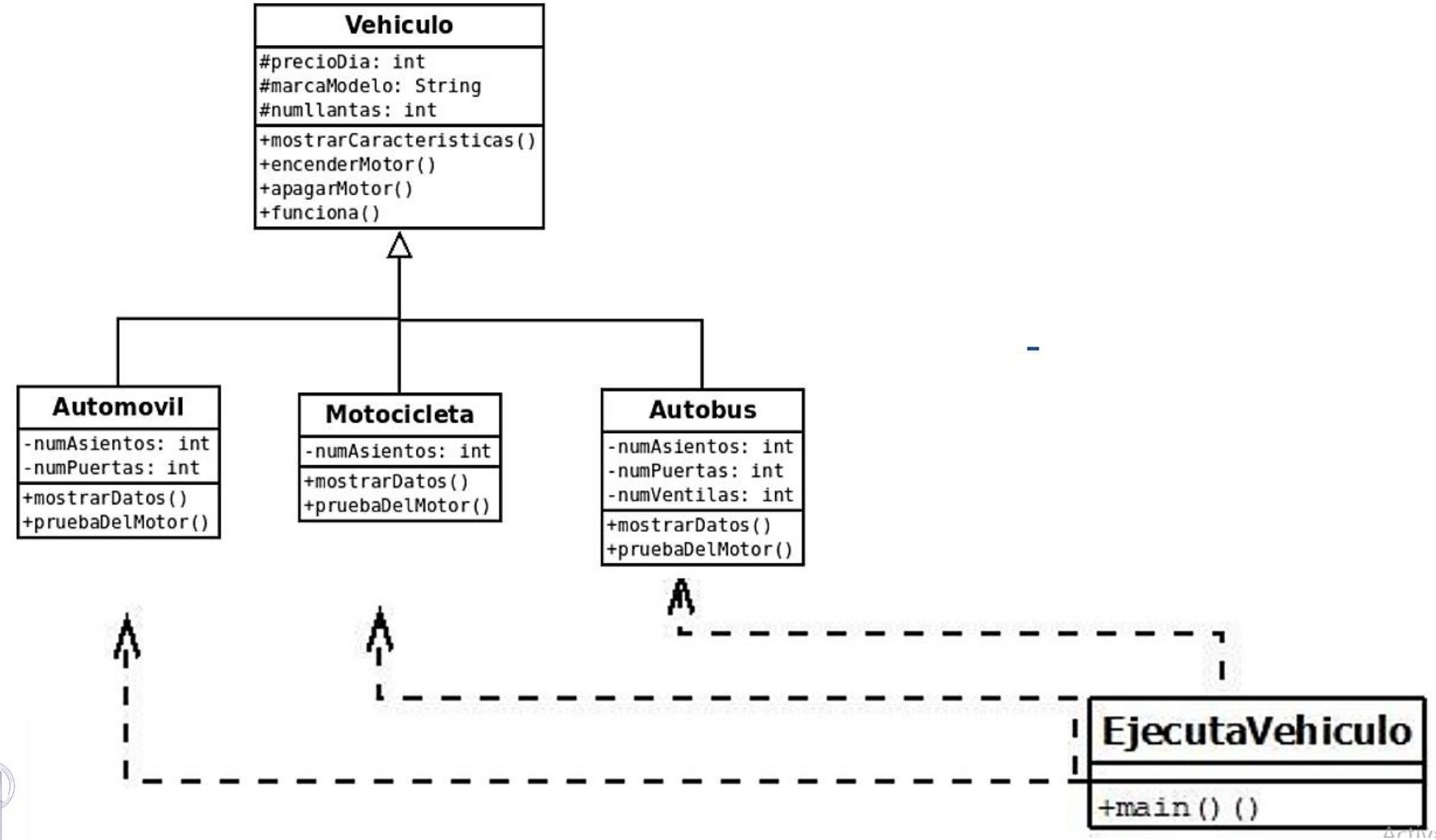
```
class SuperClase {
    public void metodo1() {
        System.out.println("Método 1 de la clase padre");
    }
}

public class SubClase extends SuperClase {
    public void metodo1(){
        super.metodo1(); //Llama al método de la superclase
        System.out.println("Método 1 de la subclase");
    }
```

this y super

3. Llamar a un constructor específico de la clase padre: Si la superclase tiene varios constructores, "`super`" se utiliza para llamar a un constructor específico de la superclase.

```
class SuperClase{
    public SuperClase() {
        // Constructor por defecto
    }

    public SuperClase(int valor) {
        // Constructor con parámetros
    }
}

public class SubClase extends SuperClase {
      public SubClase() {
        super(10);
        // Llama al constructor con parámetros de la superclase
    }
}
```

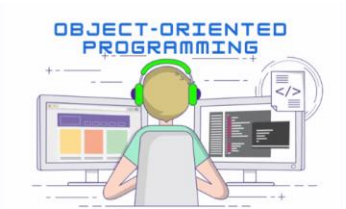
Herencia


Vehiculo
#precioDia: int
#marcaModelo: String
#numllantas: int
+mostrarCaracteristicas()
+encenderMotor()
+apagarMotor()
+funciona()
Automovil
-numAsientos: int
-numPuertas: int
+mostrarDatos()
+pruebaDelMotor()
Motocicleta
-numAsientos: int
+mostrarDatos()
+pruebaDelMotor()
Autobus
-numAsientos: int
-numPuertas: int
-numVentilas: int
+mostrarDatos()
+pruebaDelMotor()
EjecutaVehiculo
+main()()


OBJECT-ORIENTED
PROGRAMMING

Herencia

```
public class Vehiculo{
 // variables de instancia con visibilidad  protegidas
    protected int precioDia;
    protected String marcaModelo;
    protected int numllantas;

    // constructor
    public Vehiculo(String marcaModelo1, int precioDia1, int numllantas1) {
        marcaModelo = marcaModelo1;
        precioDia = precioDia1;
        numllantas = numllantas1;
        System.out.println("construyo el vehículo");
    }

    // metodos
    public void mostrarCaracteristicas(){
        System.out.println("Marca: " + marcaModelo);
        System.out.println("Precio: " + precioDia);
        System.out.println("Numero de llantas: " + numllantas);
    }
    public void encenderMotor(){
        System.out.println("el motor se encendio");
    }
    public void apagarMotor(){
        System.out.println("el motor se apago");
    }
    public void funciona(){
        System.out.println("el motor de " + marcaModelo + " funciona correctamente");
    }
}
```



Herencia

```
public class Automovil extends Vehiculo{
 // variables
 private int numAsientos = 5;
 private int numPuertas;

 public Automovil(String marcaModelo,int precioDia,int numllantas, int asientos, int puertas){
     super(marcaModelo,precioDia,numllantas);
     numAsientos = asientos;
     numPuertas = puertas;
     System.out.println("ha creado un auto");
 }
 public void mostrarDatos(){
     super.mostrarCaracteristicas();
     System.out.println( "Asientos: " + numAsientos);
     System.out.println( "Puertas: " + numPuertas);
 }
 public void pruebaDelMotor(){
     super.encenderMotor();
     super.apagarMotor();
     super.funciona();
 }
}
```

Herencia

```
public class Motocicleta extends Vehiculo{
        //variables
        private int numAsientos = 1;

        public Motocicleta(String marcaModelo, int precioDia, int numllantas, int asientos) {
                super(marcaModelo,precioDia,numllantas);
                numAsientos = asientos;
                System.out.println("Ha creado una motocicleta");
        }
        public void mostrarDatos() {
                super.mostrarCaracteristicas();
                System.out.println("Asientos: "+ numAsientos);

        }
        public void pruebaDelMotor() {
                super.encenderMotor();
                super.apagarMotor();
                super.funciona();
}
```

Herencia

```
public class Autobus extends Vehiculo{
        //variables de instancia privadas
        private int numAsientos = 41;
        private int numPuertas;
        private int numVentilas;
         // constructor con parámetros
        public Autobus(String marcaModelo, int precioDia, int numllantas, int asientos, int puertas,
int ventilas) {
                //con super estamos llamando a los atributos de la clase padre o superclase.
                super(marcaModelo,precioDia,numllantas);
                numAsientos = asientos;
                numPuertas = puertas;
                numVentilas = ventilas;
                System.out.println("Ha creado un autobús");
        }
        public void mostrarDatos() {
                super.mostrarCaracteristicas();
                System.out.println("Asientos: " + numAsientos);
                System.out.println("Puertas: " + numPuertas);
                System.out.println("Ventilas: "+ numVentilas);
        }
        public void pruebaDelMotor()
        {
            //con super estamos llamando los métodos de la clase padre o superclase.
              super.encenderMotor();
              super.apagarMotor();
              super.funciona();
        }
```

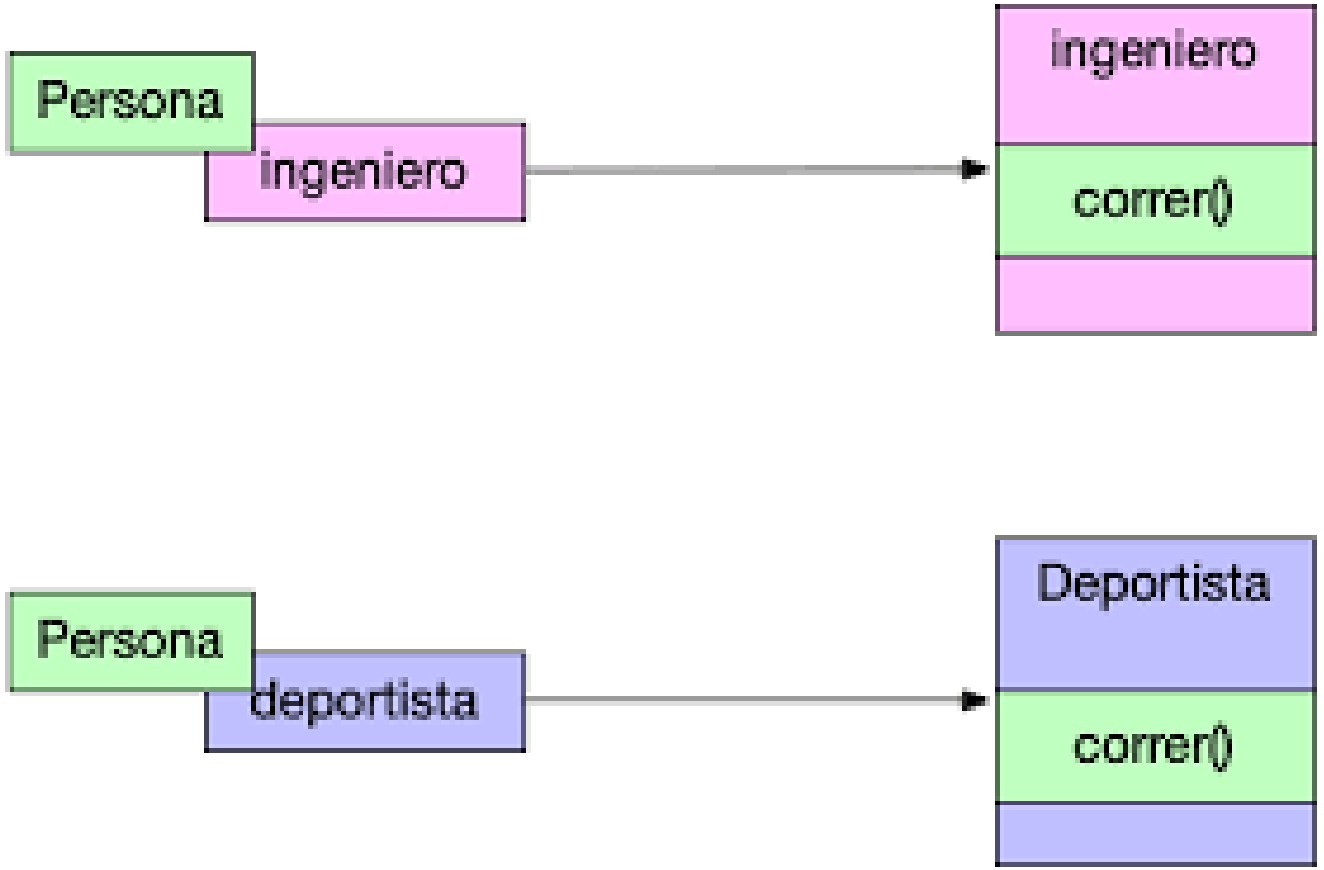
Herencia

```
public class EjecutaVehiculo {
    public static void main(String[] args){
        // Automovil
        Automovil v1 = new Automovil("volvo 550",120,4,5,4);
        v1.mostrarDatos();
        v1.pruebaDelMotor();

        // Moto
        Motocicleta m1 = new Motocicleta("Italika",120,2,2);
        m1.mostrarDatos();
        m1.pruebaDelMotor();

        //Autobús
        Autobus b1 = new Autobus("Mercedez",300,8,42,2,2);
        b1.mostrarDatos();
        b1.pruebaDelMotor();

        Autobus b2 = new Autobus("Mercedez smart",250,6,25,1,1);
        b2.mostrarDatos();
        b2.pruebaDelMotor();
    }
}
```

Polimorfismo

El polimorfismo es un concepto fundamental en la programación orientada a objetos (POO) que se refiere a la capacidad de una entidad, como una clase o un método, de asumir varias formas o comportamientos diferentes. En términos más generales, el polimorfismo permite que un objeto pueda comportarse como si fuera de un tipo diferente al suyo. Esto significa que, bajo ciertas condiciones, un objeto puede ser tratado como un objeto de una clase base, aunque en realidad sea una instancia de una clase derivada.

Polimorfismo



Polimorfismo

```java
class Animal {
    public void hacerSonido() {
        System.out.println("Sonido de animal desconocido");
    }
}

class Perro extends Animal {
    @Override
    public void hacerSonido() {
        System.out.println("Guau guau");
    }
}

class Gato extends Animal {
    @Override
    public void hacerSonido() {
        System.out.println("Miau miau");
    }
}
```

```java
public class MiPrimerProyecto {

    public static void main(String[] args) {
        Animal perro = new Perro();
        Animal gato = new Gato();

        perro.hacerSonido(); // imprimirá "Guau guau"
        gato.hacerSonido(); // Imprimirá "Miau miau"
    }
}
```

Polimorfismo

```java
class Animal {
    public void hacerSonido() {
        System.out.println("Sonido de animal desconocido");
    }
}

class Perro extends Animal {
    @Override
    public void hacerSonido() {
        System.out.println("Guau guau");
    }
}

class Gato extends Animal {
    @Override
    public void hacerSonido() {
        System.out.println("Miau miau");
    }
}
```

```java
public class MiPrimerProyecto {

    public static void main(String[] args) {
        Animal perro = new Perro();
        Animal gato = new Gato();

        perro.hacerSonido(); // imprimirá "Guau guau"
        gato.hacerSonido(); // Imprimirá "Miau miau"
    }
}
```

Sobrecarga

La sobrecarga es la capacidad de definir múltiples métodos con el mismo nombre en una clase, pero con diferentes parámetros. Esto permite que un método realice diferentes acciones según el tipo o número de argumentos que recibe. Por ejemplo:

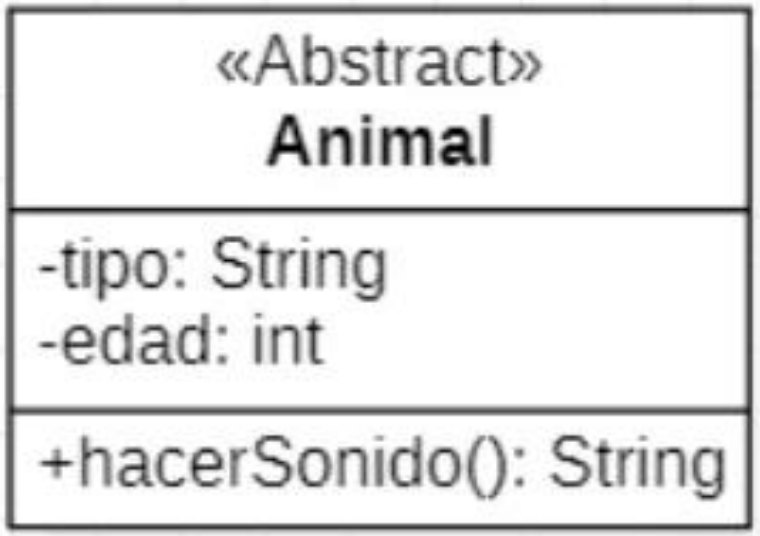
```java
public class Matematicas {
    public int sumar(int a, int b) {
        return a + b;
    }

    public double sumar(double a, double b) {
        return a + b;
    }

    public int sumar(int a, int b, int c) {
        return a + b + c;
    }
}
```

Sobre-escritura

La sobre-escritura es la capacidad de una clase derivada (subclase) de proporcionar una implementación diferente para un método que ya está definido en la clase base (superclase). La subclase redefine el método con la misma firma (nombre y parámetros) que el método en la clase base. Esto puede verse en el ejemplo de Perro y Gato en la sección de polimorfismo.

Clases abstractas

Una clase abstracta es una clase que no puede ser instanciada directamente, es decir, no se pueden crear objetos directamente a partir de ella. La clase abstracta sirve como una plantilla o modelo para otras clases que la extienden (heredan) y deben proporcionar una implementación para sus métodos abstractos. Una clase abstracta puede contener tanto métodos abstractos como métodos concretos.

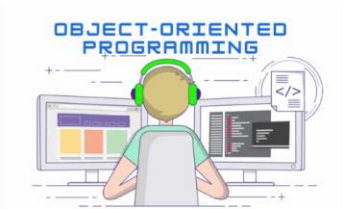


Clases abstractas

```
abstract class MiClaseAbstracta {
    // Métodos abstractos (sin implementación)
    abstract void metodoAbstracto();

    // Métodos concretos (con implementación)
    void metodoConcreto() {
        // Código del método concreto
    }
}
```

Clases abstractas

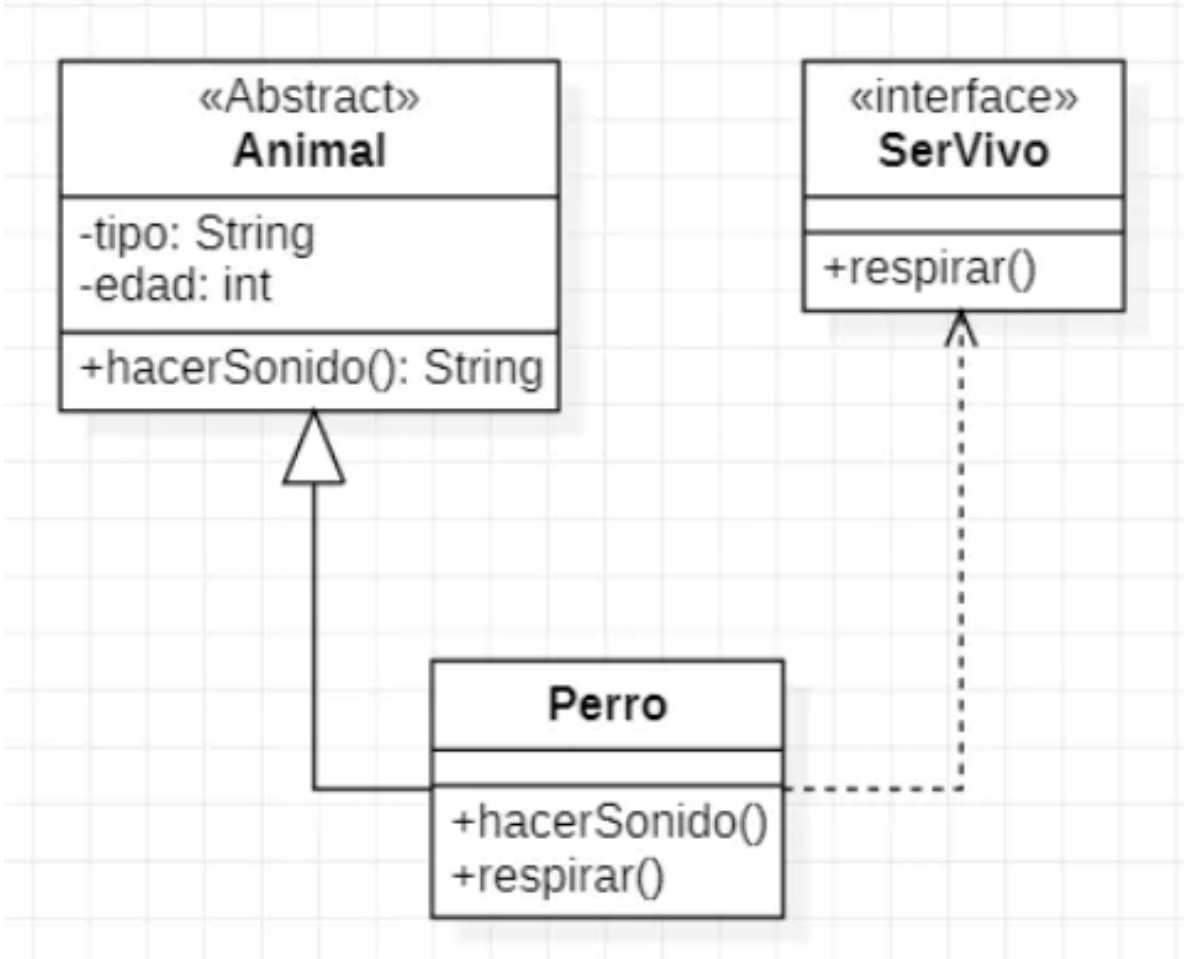
```java
abstract class Animal {
    abstract void hacerSonido();
}

class Perro extends Animal {
    @Override
    void hacerSonido() {
        System.out.println("Guau guau");
    }
}

class Gato extends Animal {
    @Override
    void hacerSonido() {
        System.out.println("Miau miau");
    }
}
```



Interfaces

```java
interface MiInterfaz {
    // Métodos abstractos
    void metodoAbstracto();

    // Constantes
    int CONSTANTE = 42;

    // Métodos con implementación (a partir de Java 8)
    default void metodoConImplementacion(){
        // Código del método con implementación
    }

    // Métodos estáticos (a partir de Java 8)
    static void metodoEstatico(){
        // Código del método estático
    }
}
```

Interfaces


«Abstract»
Animal
-tipo: String
-edad: int
+hacerSonido(): String
«interface»
SerVivo
+respirar()
Perro
+hacerSonido()
+respirar()


OBJECT-ORIENTED
PROGRAMMING

Interfaces

```java
public interface SerVivo {
    public abstract void respirar();
}

class Perro extends Animal implements SerVivo{
    @Override
    void hacerSonido() {
        System.out.println("Guau guau");
    }

    @Override
    public void respirar() {
        System.out.println("Respirando por la nariz");
    }
}
```

Clases Abstractas vs Interfaces

ABSTRACTA	INTERFACE
Tiene tanto métodos concretos como metodos abstractos abstractos	Solo puede tener metodos abstractos
Una clase solo puede extender de una única clase abstracta	Una clase puede implementar cualquier cantidad de interfaces
Puede tener un método constructor	No puede tener un método constructor
Los atributos pueden tener cualquier tipo de visibilidad(public, protected o private)	Los atributos unicamente pueden ser públicos
Pueden tener variables de instancia	No puede tener variables de instancia

Abstractas vs Interfaces

Para aprender más sobre sus diferencias, utilidades y cuando usarlas, ve los siguientes videos:

- Clase abstracta vs interfaz. ¿Cuál elegir? - # 1 Dilemas de un desarrollador
- INTERFAZ o Clase ABSTRACTA ❓ Dilemas de la POO

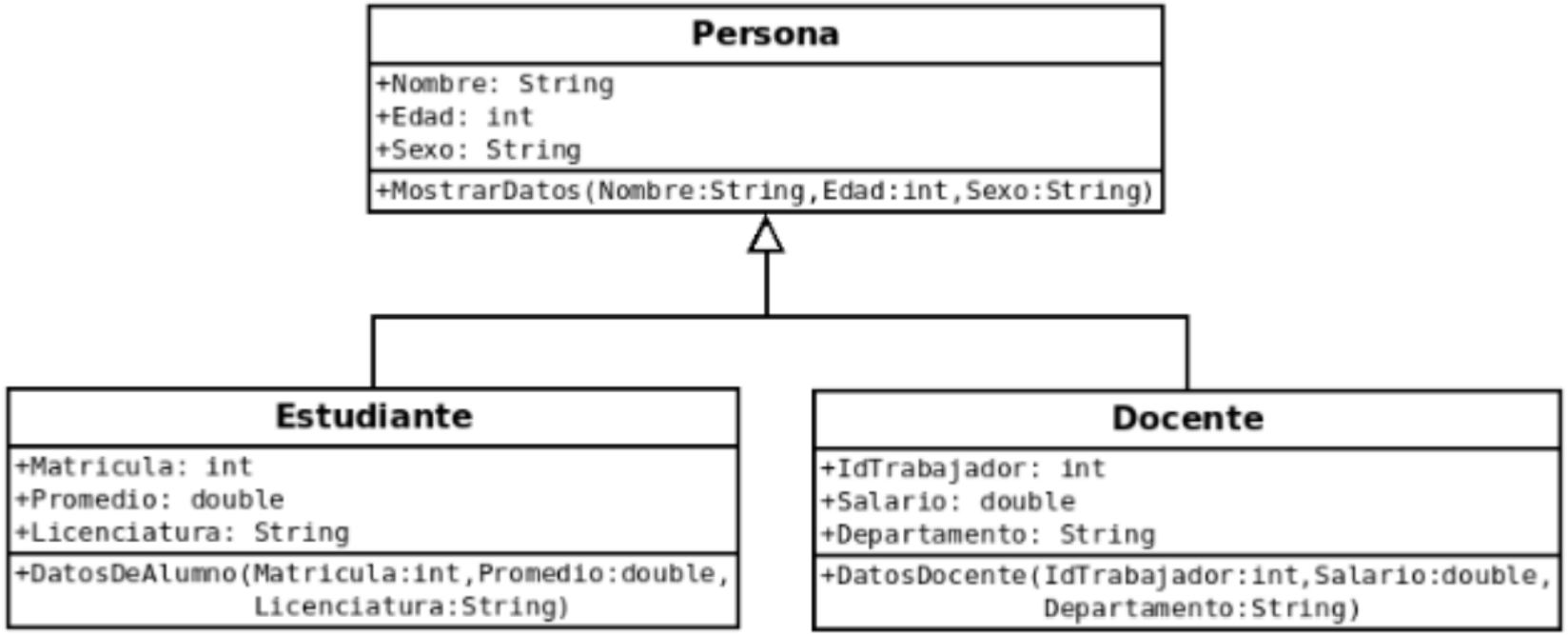
EJERCICIOS

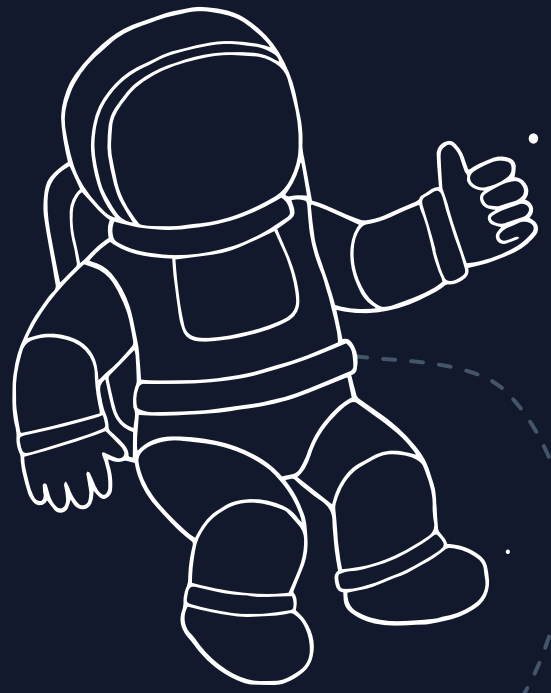
Programa de herencia Persona y tipos de personas

a) Diseña el diagrama de clases Personas y tres tipos de Personas junto con la relación de dependencia con la clase *EjecutaPersonas*.

b) Diseña la implementación y solución en Java

c) Genera tres objetos de cada clase en la clase *EjecutaPersonas*





Programa académico CAMPUS

Módulo JAVA