INSTITUTO UNIVERSITARIO DEL SURESTE

“Conceptos de orientación a objetos”

NOMBRE DEL ALUMNO

Heber Isaac Chan Domínguez

MATERIA

Análisis y diseño de sistemas

NOMBRE DE EL PROFESOR

Nicolas Navarrete

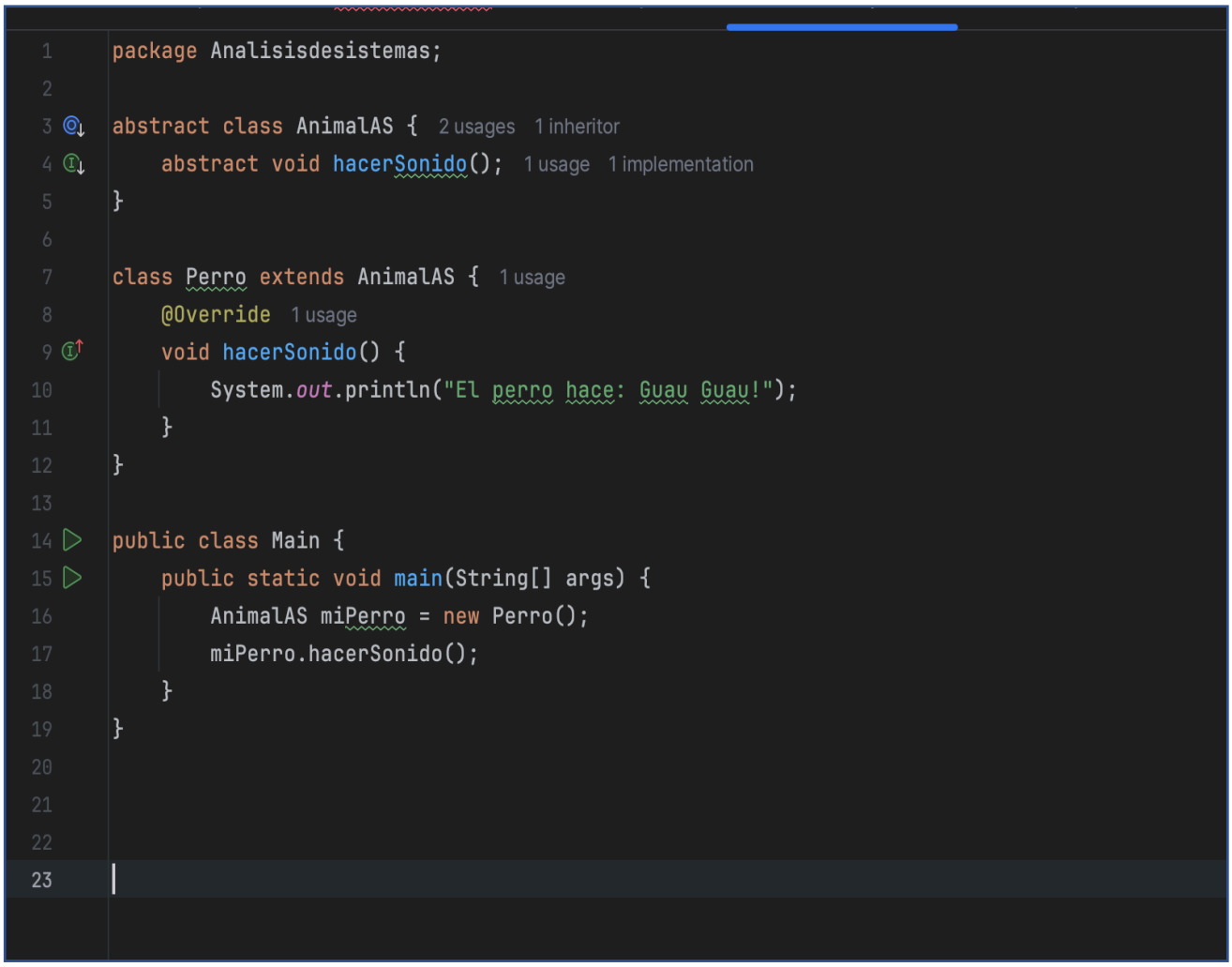
2 semestre ITIC, grupo único 27 de feb. de 25

IUNIS
INSTITUTO UNIVERSITARIO DEL SURESTE

ABSTRACCIÓN

La abstracción es el principio que permite modelar entidades del mundo real enfocándose solo en los atributos y comportamientos esenciales, ocultando los detalles de implementación. En la POO, se logra mediante clases abstractas e interfaces, que definen métodos sin especificar su implementación, dejando que las clases derivadas proporcionen su propia versión de dichos métodos.

Este concepto permite reducir la complejidad y mejorar la modularidad del código, ya que los usuarios de una clase solo necesitan conocer qué hace un objeto sin preocuparse por cómo lo hace.

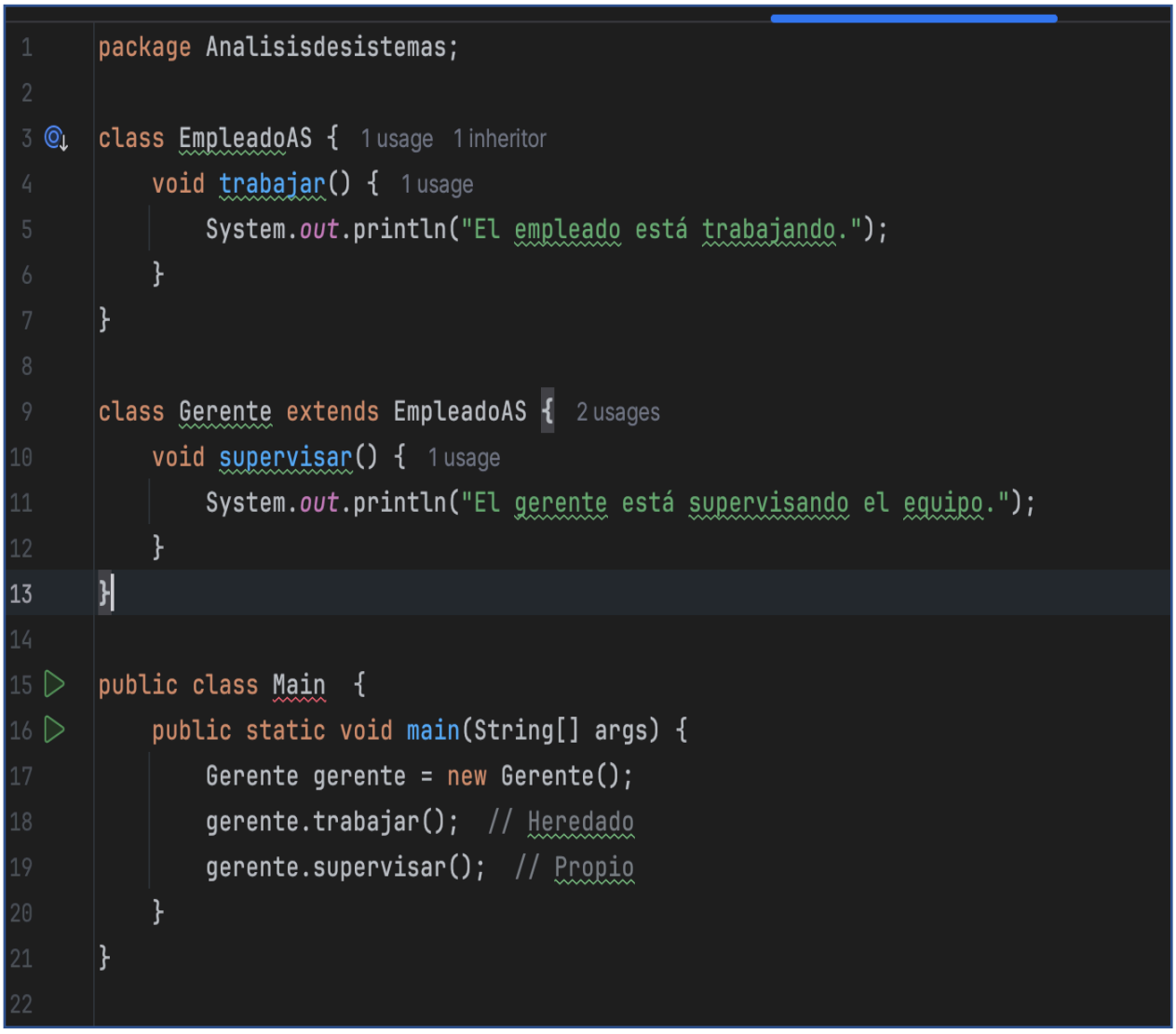


```java
package Analisisdesistemas;

abstract class AnimalAS {  2 usages  1 inheritor
    abstract void hacerSonido();  1 usage  1 implementation
}

class Perro extends AnimalAS {  1 usage
    @Override  1 usage
    void hacerSonido() {
        System.out.println("El perro hace: Guau Guau!");
    }
}

public class Main {
    public static void main(String[] args) {
        AnimalAS miPerro = new Perro();
        miPerro.hacerSonido();
    }
}
```

HERENCIA

La herencia es un mecanismo que permite crear nuevas clases a partir de clases ya existentes, reutilizando código y estableciendo relaciones jerárquicas. La clase base (o superclase) proporciona atributos y métodos que las clases derivadas (o subclases) heredan, pudiendo además agregar o modificar comportamientos específicos.

La herencia promueve la reutilización del código, reduce la redundancia y facilita la escalabilidad del software. También permite la especialización de clases, donde una subclase extiende o modifica la funcionalidad de la clase base sin necesidad de reescribir el código desde cero.

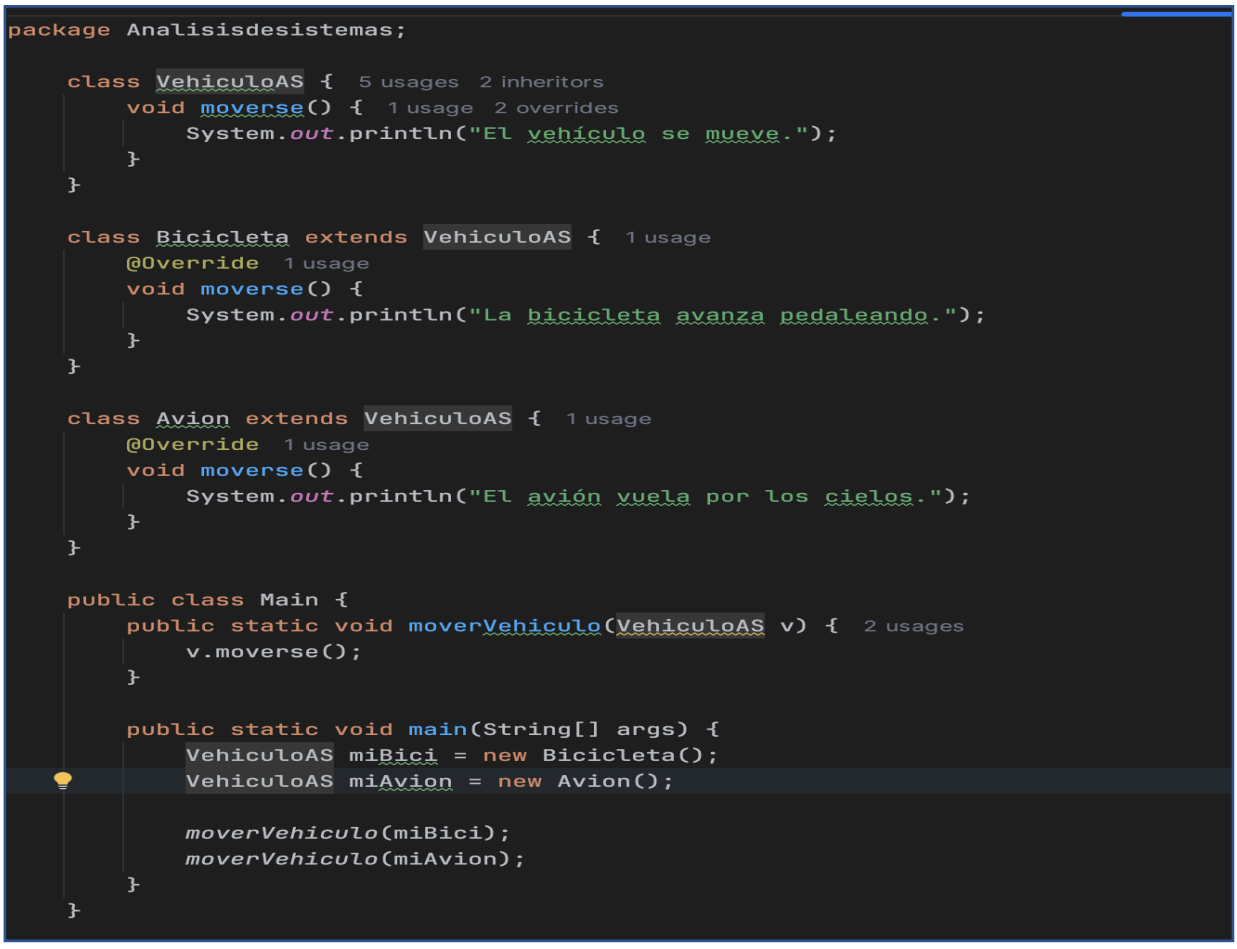


```java
package Analisisdesistemas;

class EmpleadoAS {  1 usage  1 inheritor
    void trabajar() {  1 usage
        System.out.println("El empleado está trabajando.");
    }
}

class Gerente extends EmpleadoAS {  2 usages
    void supervisar() {  1 usage
        System.out.println("El gerente está supervisando el equipo.");
    }
}

public class Main  {
    public static void main(String[] args) {
        Gerente gerente = new Gerente();
        gerente.trabajar();  // Heredado
        gerente.supervisar();  // Propio
    }
}
```

POLIMORFISMO

El polimorfismo es la capacidad de un objeto de tomar múltiples formas, permitiendo que una misma interfaz pueda ser utilizada para diferentes tipos de datos. Esto significa que diferentes clases pueden implementar métodos con el mismo nombre, pero con comportamientos específicos según la clase que los defina.

El polimorfismo facilita la flexibilidad y extensibilidad del código, ya que permite escribir funciones o métodos que pueden operar sobre objetos de distintas clases sin necesidad de conocer su implementación exacta. Se puede lograr a través de la sobrecarga de métodos (cuando varias funciones tienen el mismo nombre pero diferentes parámetros) y la sobrescritura de métodos (cuando una subclase redefine un método de su superclase para adaptarlo a su propia lógica).



```java
package Analisisdesistemas;

    class VehiculoAS {  5 usages  2 inheritors
        void moverse() {  1 usage  2 overrides
            System.out.println("El vehículo se mueve.");
        }
    }

    class Bicicleta extends VehiculoAS {  1 usage
        @Override  1 usage
        void moverse() {
            System.out.println("La bicicleta avanza pedaleando.");
        }
    }

    class Avion extends VehiculoAS {  1 usage
        @Override  1 usage
        void moverse() {
            System.out.println("El avión vuela por los cielos.");
        }
    }

    public class Main {
        public static void moverVehiculo(VehiculoAS v) {  2 usages
            v.moverse();
        }

        public static void main(String[] args) {
            VehiculoAS miBici = new Bicicleta();
            VehiculoAS miAvion = new Avion();

            moverVehiculo(miBici);
            moverVehiculo(miAvion);
        }
    }
```