Programación II
Desarrollo en Java

Clase N° 5:
Herencia y Polimorfismo

Profesores Carolina Archuby
y Daniel Díaz.

UTNMDP
Regional Mar del Plata

TEMAS A DESARROLLAR

- Herencia.
- Super
- Sobreescritura de métodos.
- Ejemplos de herencia con diagramas.
- Ejemplos de herencia con código.
- Polimorfismo.
- Subtipos y Sustitución.
- Tipos de polimorfismo: estático y dinámico.
- Ejemplos de polimorfismo con código.

HERENCIA


CONCEPTO
DE HERENCIA
Los hijos heredamos muchas características de los padres, abuelos, etc

HERENCIA

=> La herencia es una **forma de reutilización de software** en la que se crea una nueva clase (clase hija) **absorbiendo los miembros (atributos y métodos) de una clase existente (clase padre), pudiendo al mismo tiempo añadir atributos y métodos propios**.

CONCEPTO DE SUPERCLASE Y SUBCLASE

* El concepto de herencia conduce a una **estructura jerárquica** de clases o estructura de árbol.

* En la estructura jerárquica, la **clase padre** se llama **SUPERCLASE**.

* La clase hija de una superclase se llama **SUBCLASE**.

Superclases y Subclases

=> Las subclases no están limitadas al estado y comportamiento provisto por la superclase → **pueden agregar variables y métodos además de los que ya heredan.**

=> Las clases hijas también pueden **sobreescribir los métodos que heredan**

REGLAS DE LA HERENCIA

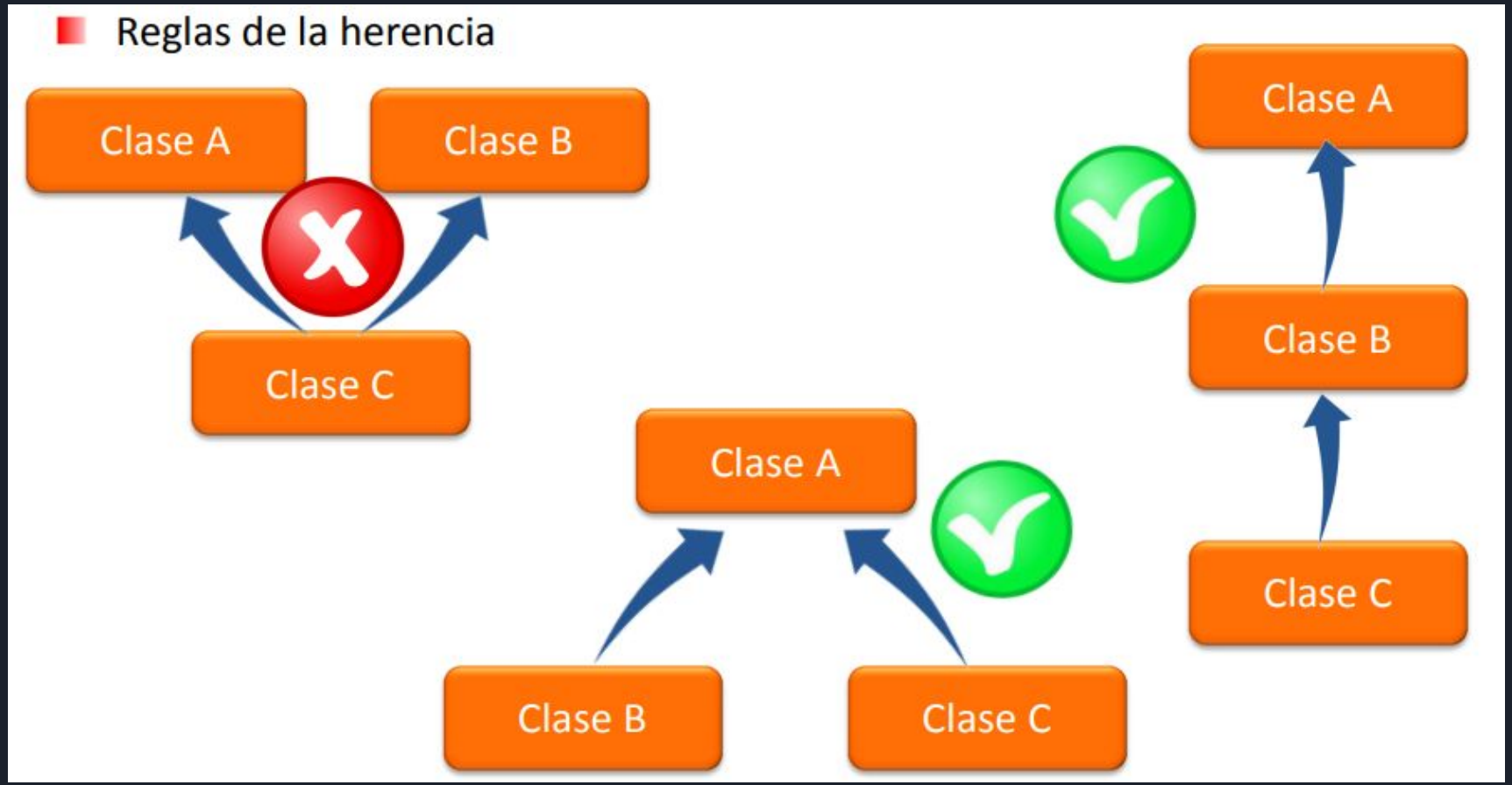
=> Una superclase puede tener cualquier número de subclases.

=> En Java, una subclase puede tener sólo una superclase (HERENCIA SIMPLE)

=> Si es posible la **herencia multinivel** (es decir, “A” puede ser heredada por “B”, y “C” puede heredar de “B”).

=> Por defecto, todas las clases derivan de java.lang.Object , a no ser que se especifique otra clase padre

REGLAS DE LA HERENCIA



Ventajas de la implementación de herencia en POO

=> **Reutilización de código**: la herencia permite a las clases hijas utilizar el código existente en la clase padre, lo que permite ahorrar tiempo y esfuerzo de programación, y además evitar errores, ya que se reutiliza código ya probado.

=> **Mantenibilidad y extensión:** la herencia puede facilitar el mantenimiento del código, la actualización y la extensión, ya que cualquier cambio en la clase padre se reflejará automáticamente en las clases hijas. Además, si tenemos una clase con una determinada funcionalidad y tenemos la necesidad de ampliar dicha funcionalidad, no necesitamos modificar la clase existente, sino que podemos crear una clase que herede de la primera, adquiriendo toda su funcionalidad y añadiendo sólo las nuevas

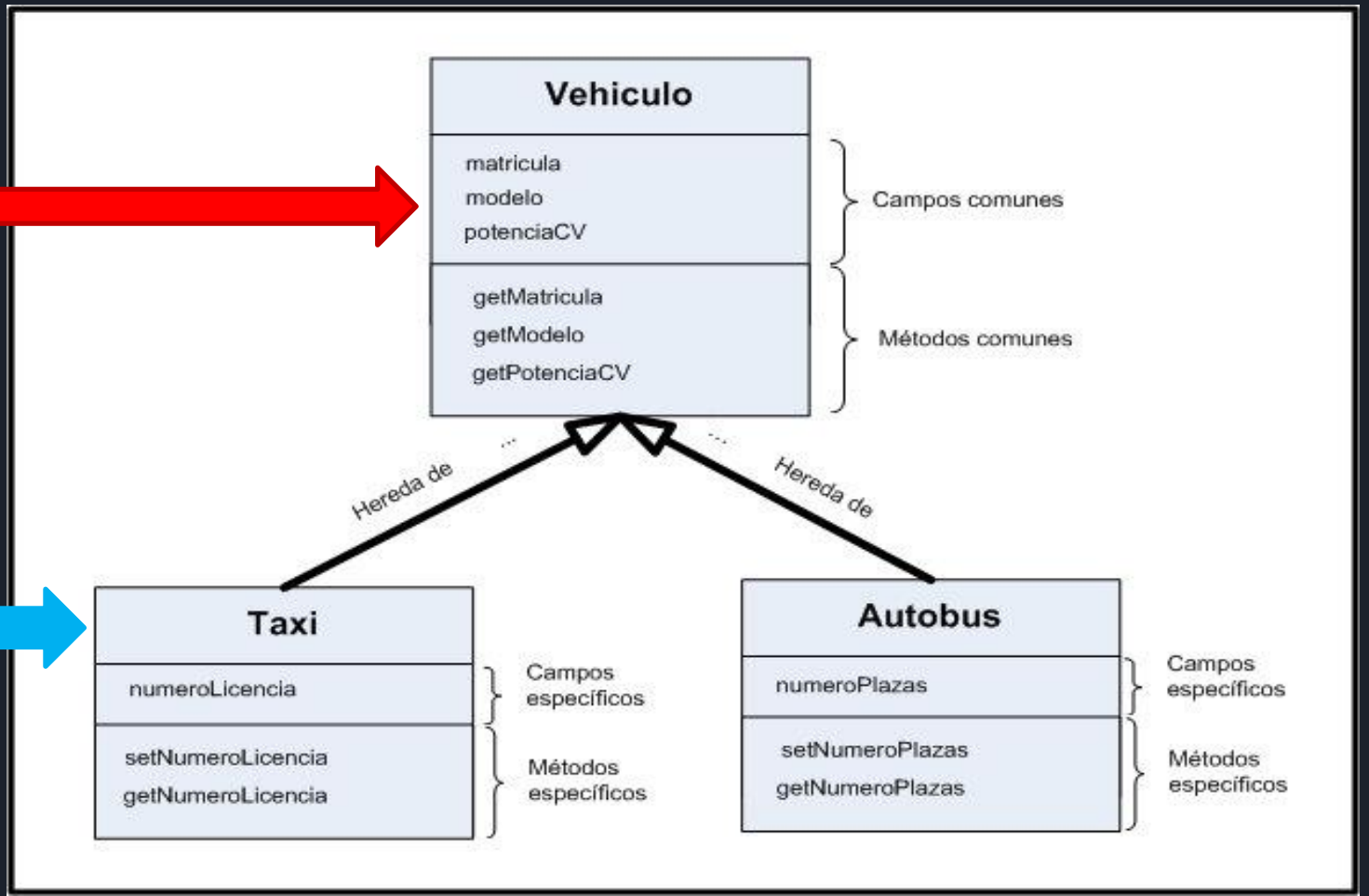
=> **Modularidad:** la herencia puede ayudar a organizar y estructurar el código de manera modular, lo que puede facilitar su comprensión y su mantenimiento.

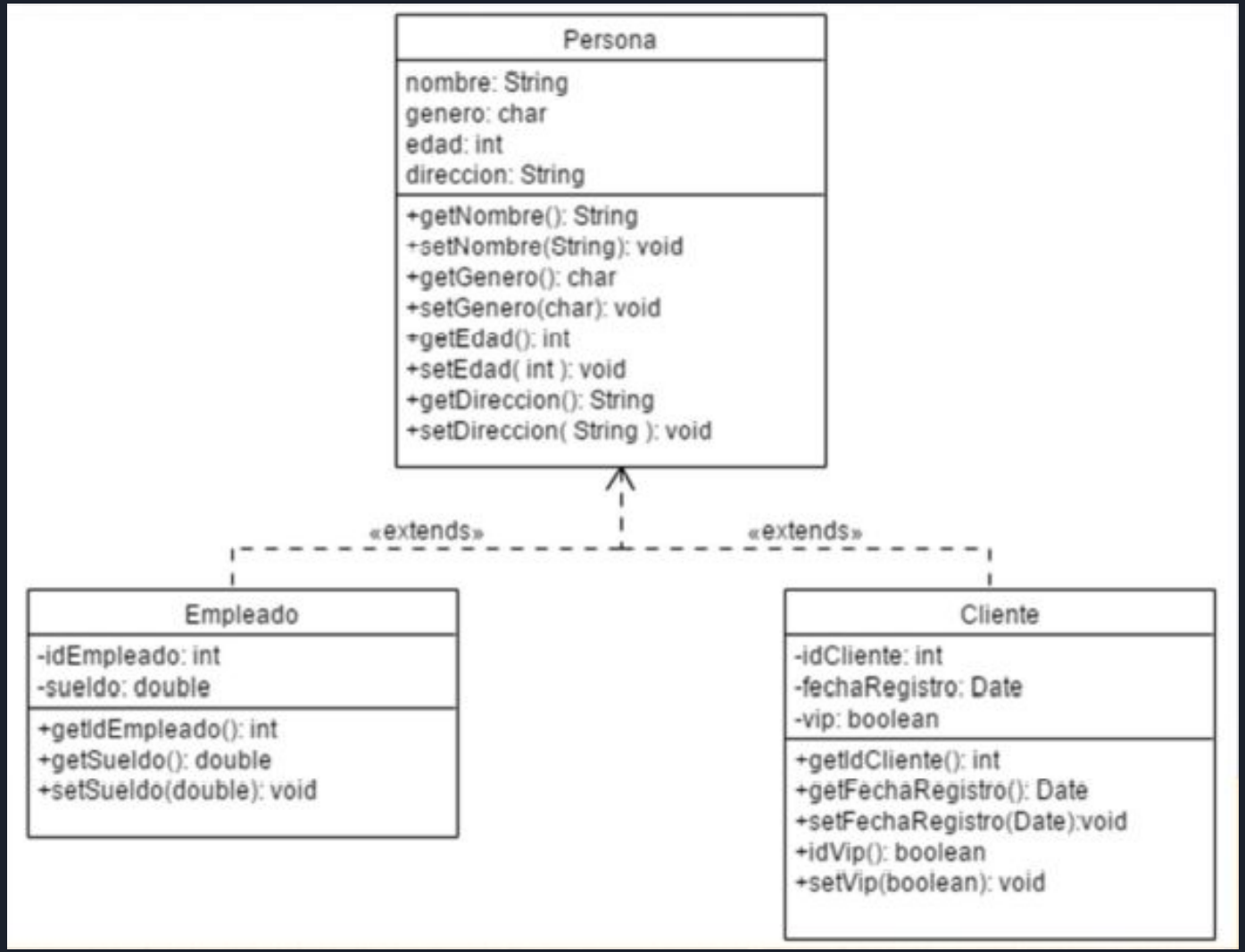
=> API: Java proporciona gran cantidad de clases (bibliotecas) al programador, en el API (Aplication Programming Interface) Java.

Ejemplos de herencia con diagramas UML

Superclase,
Clase Padre
o Clase Base

Subclases,
Clases Hijas
o Clases
Derivadas




Persona
nombre: String
genero: char
edad: int
direccion: String
+getNombre(): String
+setNombre(String): void
+getGenero(): char
+setGenero(char): void
+getEdad(): int
+setEdad(int): void
+getDireccion(): String
+setDireccion(String): void
«extends»
«extends»
Empleado
-idEmpleado: int
-sueldo: double
+getIdEmpleado(): int
+getSueldo(): double
+setSueldo(double): void
Cliente
-idCliente: int
-fechaRegistro: Date
-vip: boolean
+getIdCliente(): int
+getFechaRegistro(): Date
+setFechaRegistro(Date):void
+idVip(): boolean
+setVip(boolean): void

LA SOBREESCRITURA DE MÉTODOS:

=> Cuando una clase hereda de otra, el comportamiento de los métodos que hereda no siempre se ajusta a las necesidades de la nueva clase. En estos casos, la subclase puede optar por volver a reescribir el método heredado. Es lo que se conoce como sobrescritura de un método

=> Es un concepto que permite que una clase hija proporcione su propia implementación de un método que ya está definido en la clase padre.

=> Cuando se llama al método en un objeto de la clase hija, **la implementación de la clase hija se ejecutará en lugar de la implementación de la clase padre.**

Reglas de la sobreescritura de métodos:

=> La firma del método (su formato, que incluye el nombre del método, los tipos de parámetros y el tipo de retorno) **debe ser la misma en ambas clases**.

=> El método sobrescrito puede tener un modificador menos restrictivo que el de la superclase. Por ejemplo, el método de la superclase puede ser protected y la versión sobrescrita en la subclase puede ser public, pero nunca uno más restrictivo.

LA PALABRA RESERVADA SUPER

SE UTILIZA PARA INVOCAR MÉTODOS DE LA SUPERCLASE:

```
super.metodo();
```

La palabra reservada super y los constructores

=> La palabra clave ***super*** es una llamada al constructor de la superclase.

=> La **llamada al *super*** debe tener los **mismos parámetros** que tenga el constructor de la superclase. El constructor de la superclase inicializa los campos correspondientes y le pasa el control al constructor de la subclase

=> **El constructor de una subclase debe tener siempre como primera sentencia una invocación al constructor de su superclase.**

=> Si el constructor en la subclase no invoca explícitamente al constructor de la superclase, el compilador de Java inserta automáticamente una llamada al súper sin parámetros *(Esto implica que la superclase tendría que tener definido un constructor sin parámetros. Si sólo tuviera constructores con parámetros, entonces el compilador señalaría el error).*

Ejemplo de herencia y constructores:

```
public class Padre {
    public Padre() {
        System.out.println("Constructor de Padre");
    }
}
public class Hija extends Padre{
    public Hija() {
        super();
        System.out.println("Constructor de Hija");
    }
}
```

Invoca al constructor de la clase Padre

La palabra reservada super y el resto de los métodos

=> En contra de la regla de las llamadas a super en los constructores, **la llamada a super en los métodos puede ocurrir en cualquier lugar dentro de dicho método**, no tiene por qué ocurrir en su primer sentencia.

=> Al contrario que en las llamadas a super en los constructores, **no se genera automáticamente ninguna llamada a super y tampoco se requiere ninguna llamada a super**, es completamente opcional. Por lo tanto, el método de una subclase podría sobrescribir y ocultar por completo la versión de la superclase del mismo método

Primer ejemplo de herencia con código

Creacion de la clase padre *Animal*:

```
public class Animal {

      //Variable de instancia
      private String nombre;

      //Constructor por defecto
      public Animal() {
      }

      //Constructor con parámetros
      public Animal(String nombre) {
         this.nombre = nombre;
      }

      //Método
      public void comer() {
         System.out.println("Estoy comiendo");
      }
}
```

Nombre de la clase: primer letra mayúscula y en singular

Los constructores tienen el mismo nombre de la clase!!!

Método concreto (no es abstracto)

Creacion de la clase hija *Perro*:

```
public class Perro extends Animal {

    //Variable de instancia
    private int hambre;

    //Constructor por defecto
    public Perro() {
    }

    //Constructor con parámetros
    public Perro(String nombre, int hambre) {
        super(nombre);
        this.hambre = hambre;
    }

    //Método sobreescrito
    @Override
    public void comer() {
        super.comer();
        hambre --;
    }
}
```

Hay una herencia, por lo tanto Perro es una subclase de Animal

Hereda la variable de instancia “nombre” y agrega una nueva

Llama al constructor de la superclase Animal

Se sobreescribe el método de la superclase

Se llama al método comer() de la superclase Animal

Segundo ejemplo de herencia con código

Creacion de la clase padre ***Persona***:

```java
public class Persona {

    //Atributos comunes
    private String nombre;
    private int edad;
    private char genero;
    private boolean isMarried;

    //Constructor de la clase padre Persona
    public Persona(String nombre, int edad, char genero, boolean isMarried) {
        super(); // ??
        this.nombre = nombre;
        this.edad = edad;
        this.genero = genero;
        this.isMarried = isMarried;
    }

    //Metodos comunes
    public String saludar () {

        return "Hola";
    }
    //Getters y Setters...
```

(Todas las clases extienden de la clase Object por defecto)

Segundo ejemplo de herencia con código

Creacion de la clase hija *Empleado*

```java
public class Empleado extends Persona {

    //Atributos propios de la subclase
    private int idEmpleado;
    private double sueldoXHora;

    //Constructor de la subclase Empleado
    public Empleado(String nombre, int edad, char genero, boolean isMarried, int idEmpleado, double sueldoXHora)
        super(nombre, edad, genero, isMarried);//super hace referencia al constructor de la clase padre (Persona)
        this.idEmpleado = idEmpleado;
        this.sueldoXHora = sueldoXHora;
    }

    //Metodos propios de la subclase
    public double calcularSueldo(int horas) {
        return this.sueldoXHora * horas;
    }
    //Getters y Setters...
```

Creación de la clase hija *Cliente*

```java
public class Cliente extends Persona {

    // Atributos propios de la subclase
    private int idCliente;
    private Date fechaRegistro;
    private boolean isVip;

    // Constructor de la subclase Cliente
    public Cliente(String nombre, int edad, char genero, boolean isMarried, int idCliente, Date fechaRegistro,
            boolean isVip) {

        super(nombre, edad, genero, isMarried);//super hace referencia al constructor de la clase padre (Persona)
        this.idCliente = idCliente;
        this.fechaRegistro = fechaRegistro;
        this.isVip = isVip;
    }

    // Metodos propios de la subclase
    public String realizarCompra(String productos[]) {

        return "Compra realizada";
    }
```

HERENCIA Y DERECHOS DE ACCESO (modificadores de acceso)

=> Una subclase **puede** invocar a cualquier método **público** de su superclase como si fuera propio, no necesita ninguna variable

=> Una subclase **NO puede** acceder a los miembros **privados** de la superclase

Si un método de una subclase necesita acceder a un campo privado de su superclase, la superclase necesitará ofrecer los métodos apropiados (por ejemplo, los setters y getters).

Ejemplo de herencia y atributos privados. La clase padre:

```
public class Animal {

    //Variable de instancia
    private String nombre;

    //Constructor por defecto
    public Animal() {
    }

    //Constructor con parámetros
    public Animal(String nombre) {
       this.nombre = nombre;
    }

    //Método
    public String getNombre() {
       return nombre;
    }
}
```

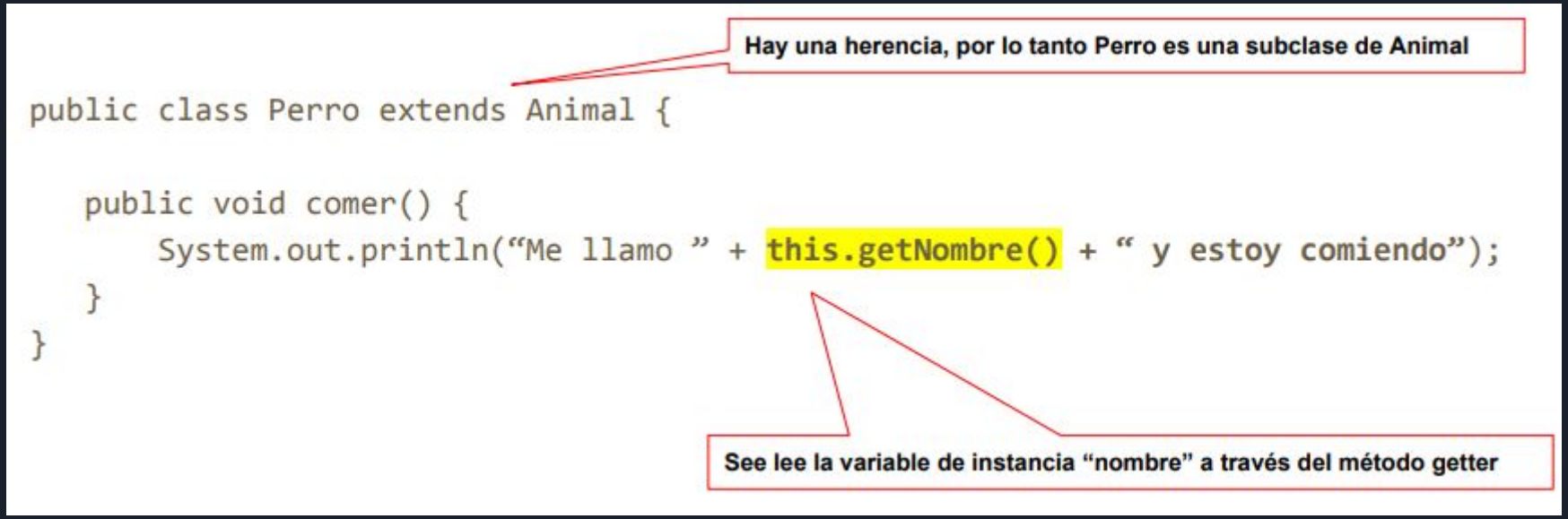
Nombre de la clase: primer letra mayúscula y en singular

El modificador de acceso "private" indica que la variable de instancia "nombre" puede ser leída desde la clase Animal. El resto de las clases que quieran leerla deben acceder a través del getter.

Los constructores tienen el mismo nombre de la clase!!!

Método "getter" para leer desde otro objeto el nombre de la variable "nombre"

Ejemplo de herencia y atributos privados. La clase hija:



Ejemplo de herencia y atributos protected. La clase padre:

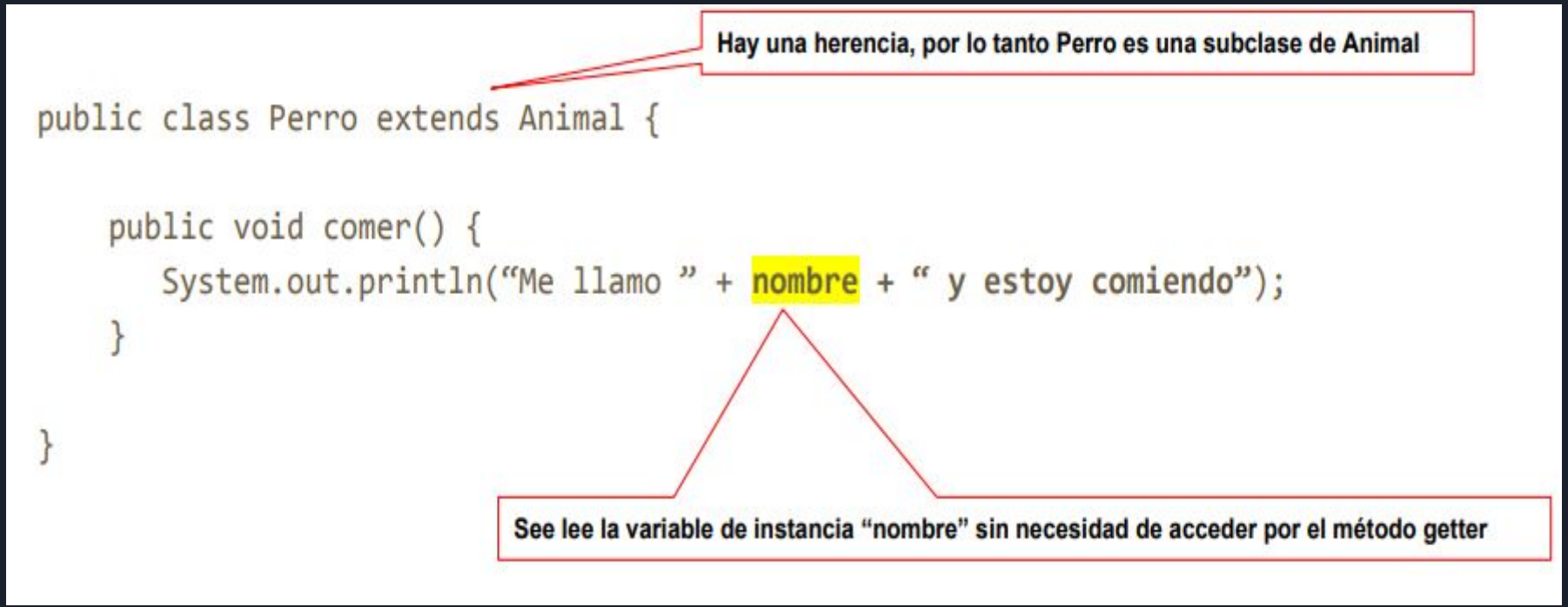
```
public class Animal {

    //Variable de instancia
    protected String nombre;

    //Constructor por defecto
    public Animal() {
    }

    //Constructor con parámetros
    public Animal(String nombre) {
        this.nombre = nombre;
    }

    //Método
    public String getNombre() {
        return nombre;
    }

}
```

Nombre de la clase: primer letra mayúscula y en singular

El modificador de acceso "protected" indica que la variable de instancia "nombre" puede ser leída desde las subclases.

Los constructores tienen el mismo nombre de la clase!!!

Ejemplo de herencia y atributos protected. La clase hija:



HERENCIA Y CLASES FINAL

Si queremos evitar que una clase sea heredada por otra, deberá ser declarada con el modificador final delante de superclase.

```
public final class ClaseA {
    //Código de la clase
     :
}
```

Si otra clase intenta heredar de una clase final, se producirá un error de compilación.

```
public class ClaseB extends ClaseA {
    //Esta clase no compilará y va a generar un error
     :
}
```

SUBTIPOS, SUSTITUCIÓN y VARIABLES POLIMORFICAS:

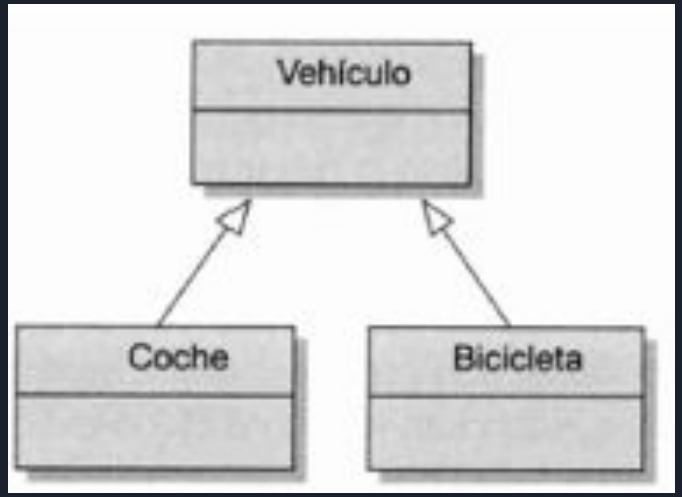
=> Por analogía con la jerarquía de clases, los tipos forman una jerarquía de tipos. El tipo que se define mediante la clase padre es el supertipo, y el que se define mediante la clase hija es el subtipo.

=> Se pueden **usar objetos de subtipos en cualquier lugar donde se espera un objeto del supertipo.** Esto se llama **SUSTITUCION**.

=> **Las variables pueden contener objetos del tipo declarado o de cualquier subtipo del tipo declarado**.

=> Las variables que contienen objetos son **variables polimórficas**, ya que **una misma variable puede contener objetos de distintos tipos**.

Ejemplo de sustitución y variables polimórficas:



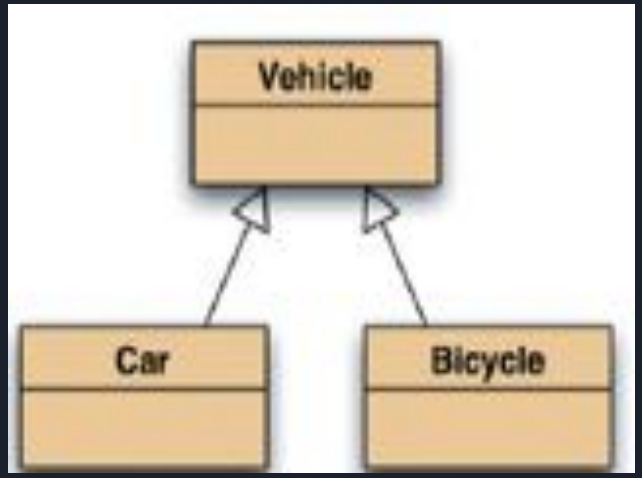
Las siguientes sentencias son válidas:

```
Vehiculo v1 = new Vehiculo();
Vehiculo v2 = new Coche();
Vehiculo v3 = new Bicicleta();
```

Las siguientes sentencias NO son válidas:

```
Coche a1 = new Vehiculo();      // ¡Es un error!
Coche a2 = new Bicicleta();     // ¡Es un error!
```

CASTING (ENMASCARAMIENTO DE TIPOS)

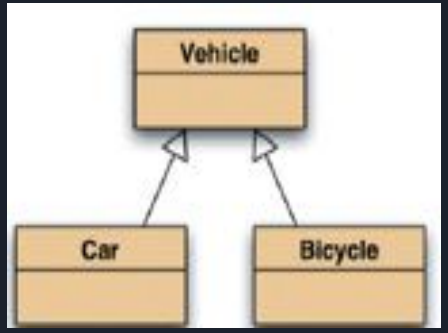


Un ejemplo:

```
Vehiculo v;

Coche a = new Coche();

v = a; // es correcta

a = v; // es un error

a = (Coche) v; // correcto
```

Enmascaramiento o casteo: al poner el tipo de dato Coche entre paréntesis por delante de la variable v, el compilador toma a la variable v como un coche y no arroja error **(¡OJO!!! ¡Reparar en que v EFECTIVAMENTE GUARDABA UN COCHE, NO UN VEHICULO!!)**

CASTING (ENMASCARAMIENTO DE TIPOS)



Otro ejemplo:

```
Vehiculo v;
   Coche a;
Bicicleta b;
a = new Coche();
   v = a;          // correcta
   b = (Bicicleta) a;      // ¡error en tiempo de compilación!
   b = (Bicicleta) v;      // ¡error en tiempo de ejecución!
```

Las últimas dos asignaciones fallarán. El intento de asignar la variable `a` en la variable `b` (aun estando enmascarada) dará por resultado un error en tiempo de compilación. El compilador se dará cuenta de que `Coche` y `Bicicleta` no constituyen una relación subtipo/supertipo, y por este motivo la variable `a` nunca puede contener un objeto `Bicicleta`; esta asignación no funcionará nunca.

CASTING o ENMASCARAMIENTO DE TIPOS)

=> Se especifica indicando el tipo de objeto entre paréntesis

=> El objeto no cambia en nada, simplemente se permite usar la referencia adecuadamente.

=> **En tiempo de ejecución se comprueba que el objeto es realmente de ese tipo**. **ClassCastException** si no lo es.

=> En Java esto se puede comprobar:

if (v instanceof A) // si el objeto v pertenece a la clase A o una de //sus subclases

Para el ejemplo anterior: *if (v instanceof Car)*

c = (Car)v;

=> Debe usarse con moderación

POLIMORFISMO

=> El polimorfismo es la **CAPACIDAD DE UNA VARIABLE, MÉTODO O OBJETO PARA TOMAR MÚLTIPLES FORMAS.**

=> Ventaja del polimorfismo: aumenta la flexibilidad y la capacidad de generalización del código, permitiendo escribir programas más genéricos, reutilizables y fáciles de extender.

=> Existen dos tipos de polimorfismo:

* Estático (sobrecarga de métodos)
* Dinámico (enlace dinámico)

TIPOS DE POLIMORFISMO

A) POLIMORFISMO ESTÁTICO O SOBRECARGA DE MÉTODOS:

=> Se produce cuando **una clase** tiene **varios métodos con el mismo nombre pero con diferentes parámetros o argumentos de entrada.** (esto significa que pueden realizar acciones similares, pero con diferentes tipos de datos o número de argumentos)

=> **En la sobrecarga no interviene el tipo de retorno**.

=> El compilador de Java decide cuál de los métodos sobrecargados se debe invocar **en tiempo de compilación**, basándose en los tipos de argumentos que se le pasan.

Ejemplo de polimorfismo estático:

```java
public class PolimorfismoEstatico {
    public int suma(int x, int y) {
        return x + y;
    }
    public double suma(double x, double y) {
        return x + y;
    }
    public static void main(String[] args) {
        PolimorfismoEstatico obj = new PolimorfismoEstatico();
        System.out.println(obj.suma(3, 4)); // llama a la versión de int
        System.out.println(obj.suma(2.5, 3.2)); // llama a la versión de double
    }
}
```

Otro ejemplo de polimorfismo estático:

```java
public class Impresora {
    public void imprimir(String texto) {
        System.out.println(texto);
    }

    public void imprimir(String texto, int copias) {
        for (int i = 0; i < copias; i++) {
            System.out.println(texto);
        }
    }
}
```

SOBRECARGA DE MÉTODOS Y HERENCIA:

=> La sobrecarga también podría darse en las clases hijas: una clase hija redefine un método de la clase padre, con el mismo nombre pero distinta lista de parámetros.

Ejemplo de sobrecarga de métodos y herencia:

```
class Ventana

{

    public void copiar(Ventana w) {...}

    public void copiar(String p, int x, int y)
    {...}

}
```

```
class VentanaEspecial extends Ventana

{

    public void copiar(char c,int veces,int x,int
    y) {...}

}
```

Implementación:

```
VentanaEspecial mv = new VentanaEspecial();

Ventana w = new Ventana();

mv.copiar("***",10,15);

mv.copiar(w);

mv.copiar('.',5,10,10);
```

TIPOS DE POLIMORFISMO

B) POLIMORFISMO DINÁMICO O ENLACE DINÁMICO:

=> Está **ligado a la herencia y a los** SUBTIPOS y la SUSTITUCION (el tema antes visto de asignar un objeto de una clase a una variable de su superclase):

* Cuando encontramos un comportamiento que muchas clases derivadas van a realizar, la tendencia es enviar ese método a la superclase para que todas sus hijas realicen ese comportamiento.
* ¿La contra? Se pierde la especialización de ese método.

=> ¿La solución? **Sobreescritura del método de la clase padre en las clases hijas. El método tiene la misma firma en la clase padre y en las hijas, y al invocarlo, se determinará en tiempo de ejecución cuál de ellos se ejecutará.**

Diferencias entre Polimorfismo Dinámico y Polimorfismo Estático:

=> TIEMPO: El estático se da en tiempo de COMPILACIÓN y el dinámico en tiempo de EJECUCIÓN.

=> En el ESTATICO los métodos tienen el MISMO NOMBRE pero DIFERENTES PARAMETROS, y los métodos se encuentran en la MISMA CLASE o en SUBCLASES.

=> En el DINAMICO los métodos tienen la MISMA FIRMA, y se encuentran en DISTINTAS CLASES.

Polimorfismo Dinámico: Ejecución dinámica de métodos:

=> **Pueden invocarse aquellos métodos del objeto que también estén definidos o declarados en la superclase**, pero no aquellos que sólo existan en la clase a la que pertenece el objeto

=> **El método que se ejecutará se determina en tiempo de ejecución y está determinado por el tipo dinámico, no por el tipo estático** (de ahí el nombre de polimorfismo dinámico).

¿CÓMO?... VEAMOS...

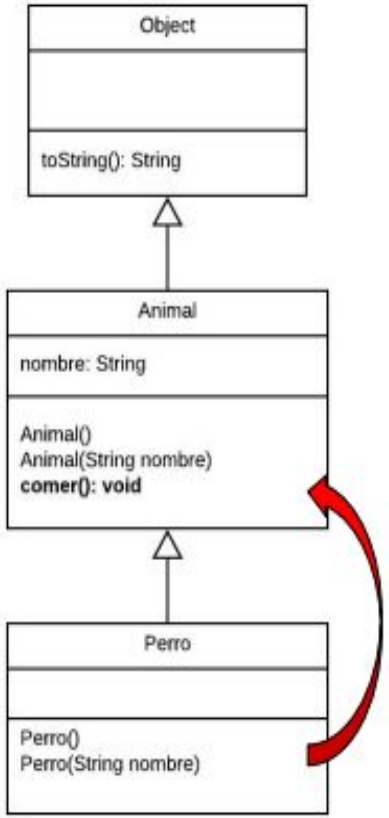
Cómo se determina el método a ejecutar:

=> Los métodos sobrescritos de las subclases tienen precedencia sobre los métodos de las superclases.

=> La búsqueda del método comienza al final de la jerarquía de herencia (comienza en la clase dinámica de la instancia), entonces **la última redefinición de un método** (el método que se encuentra primero) **es la que se ejecuta primero**.

=> Cuando un método está sobrescrito, sólo se ejecuta la última versión (la más baja en la jerarquía de herencia). Las versiones del mismo método en cualquier superclase no se ejecutan automáticamente

Primer ejemplo de ejecución dinámica de métodos:



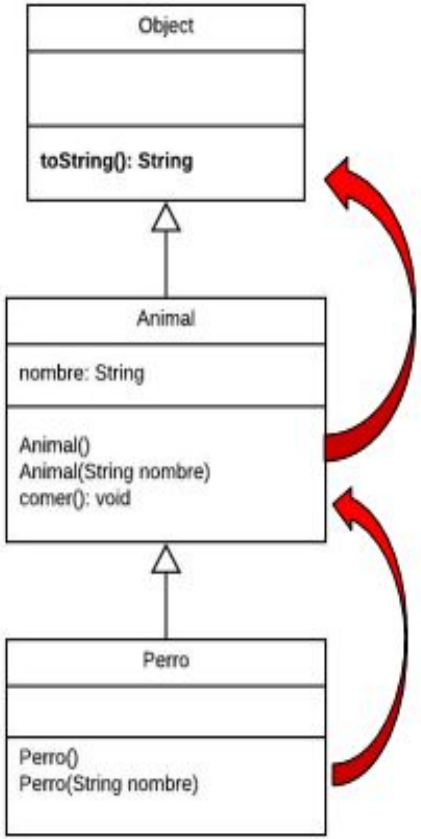
Ejemplo 1: Se quiere ejecutar el método comer() de la clase Perro.

```
public static void main(String [] args) {
    Perro perro = new Perro("Ayudante de Santa");
    perro.comer();
}
```

Para acceder a los métodos de la clase Perro, es necesario crear una instancia.

1) Se busca el método **comer()** en la subclase Perro.
2) Como no se encuentra la sobreescritura, se sube en la jerarquía hasta que se encuentra y se ejecuta.

Segundo ejemplo de ejecución dinámica de métodos:



Ejemplo 2: Se quiere ejecutar el método toString() de la clase Perro.

```
public static void main(String [] args) {
    Perro perro = new Perro("Ayudante de Santa");
    System.out.println(perro.toString());
}
```

Para acceder a los métodos de la clase Perro, es necesario crear una instancia.

1) Se busca el método **toString()** en la subclase Perro.
2) Como no se encuentra la sobreescritura, se sube en la jerarquía hasta que se encuentra y se ejecuta.

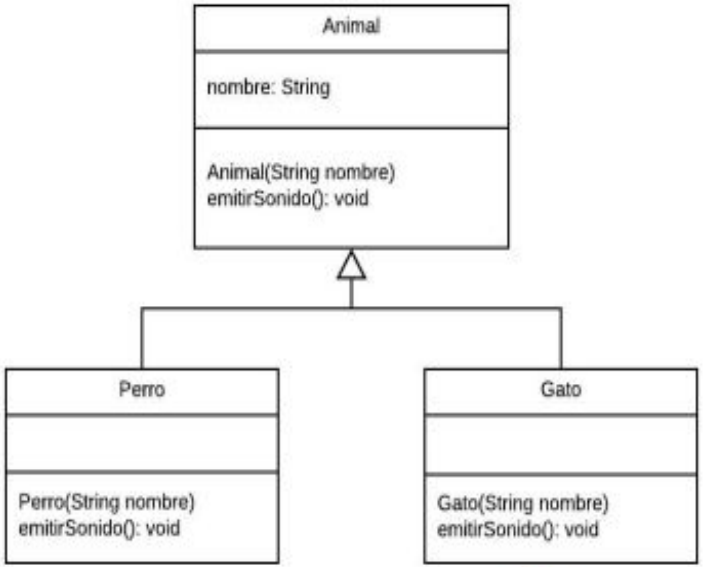
Tercer ejemplo de polimorfismo dinámico: Tenemos las siguientes clases:

```
public abstract class Animal {

    //Variable de instancia
    protected String nombre;

    //Constructor con parámetros
    public Animal(String nombre) {
       this.nombre = nombre;
    }

    public abstract void emitirSonido();

}
```

```
public class Perro extends Animal {
    @Override
    public void emitirSonido() {
        System.out.println("Guau!");
    }
}
```

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

Tercer ejemplo de polimorfismo dinámico: Invocación del método polimórfico



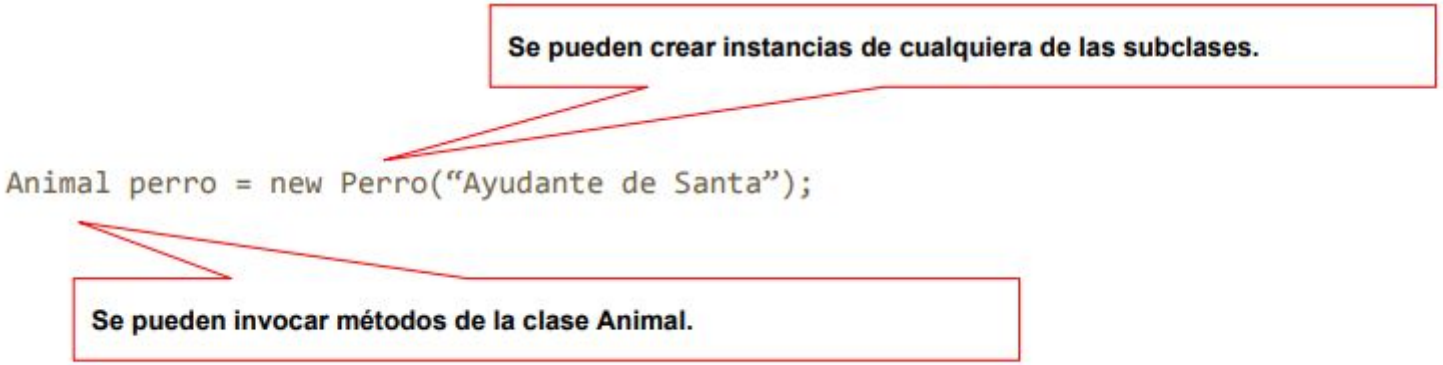

```
public static void main(String [] args) {

    Animal [] animales = new Animal[2];

    Perro perro = new Perro("Ayudante de Santa");
    Gato gato = new Gato("Bola de nieve I");

    animales[0] = perro;
    animales[1] = gato;

    for (int i = 0; i < animales.lenght; i++) {
        System.out.println(animales[i].emitirSonido());
    }
}
```

Se invoca al método de la subclase Gato cuando animales[1].emitirSonido();

Se invoca al método de la subclase Perro cuando animales[0].emitirSonido();

Polimorfismo: warnings

- Los métodos que se pueden invocar son los que contiene la clase del tipo declarado.



- Si hay métodos declarados en la subclase (Perro) que no pertenecen a la superclase (Animal), no se pueden invocar.

Ejemplo de uso incorrecto del polimorfismo dinámico:

```java
public abstract class Animal {

    //Variable de instancia
    protected String nombre;

    //Constructor con parámetros
    public Animal(String nombre) {
        this.nombre = nombre;
    }

    public abstract void emitirSonido();
}
```

```java
public class Perro extends Animal {

    @Override
    public void emitirSonido() {
        System.out.println("Guau!");
    }

    public void agarrarPelota() {
        System.out.println("Agarré la pelota");
    }
}
```

comportamiento definido para la clase hija, que no existe en la clase padre

Ejemplo de implementación incorrecta en el Main:

```java
Animal perro = new Perro("Ayudante de Santa");

perro.agarrarPelota();
```

El método no está definido en la clase animal

CONCLUSION:

=> El polimorfismo se beneficia directamente de la abstracción y la herencia al permitir que el mismo método se comporte de manera diferente dependiendo del objeto que lo invoque, basado en una estructura común definida por la abstracción (clase base) y las especializaciones proporcionadas por la herencia (clases derivadas).

=> En el ejemplo de los animales, esto facilita la gestión unificada de diferentes tipos animales, manteniendo al mismo tiempo la capacidad de tratarlos según sus características y comportamientos específicos