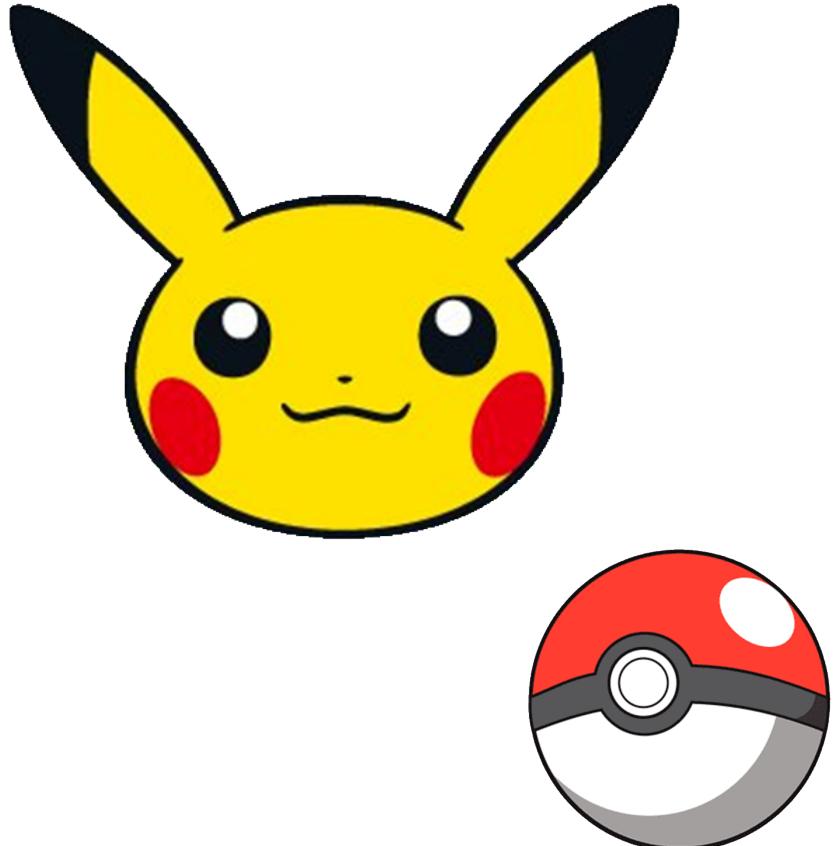
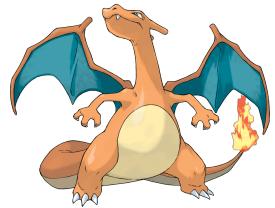


Módulo 09

Herencia y Poliformismo





Pokémon



ELECTRIC



GRASS



All Pikachu's are electric Pokémon.
All electric Pokémon are Pokémon.

Therefore,
All Pikachu's are Pokémon.

Nombra todos los atributos y métodos que puedes que pertenezcan a las siguientes clases:

Matrículas terminación 0, 1, 2 y 3

- Clase Pokemon



Matrículas terminación 4, 5 y 6

- Clase ElectricPokemon



Matrículas del 7 al 9

- Clase Pikachu



Atributos / Métodos	Java representation	All Pokemon	Electric Pokemon	Pikachu
Pokemon Number	int pokeDexNo;	Yes	Yes	Yes
Name	String name;	Yes	Yes	Yes
Height	double height	Yes	Yes	Yes
Can be healed	method heal()	Yes	Yes	Yes
Resistant to electricity	boolean resistantToElectricity	No	Yes	Yes
Can attack with thundershock	method thundershock()	No	Yes	Yes
Can surf	method surf()	No	No	Yes
Can evolve to Raichu	method evolveRaichu()	No	No	Yes

Herencia

La **herencia** es un concepto de la programación orientada a objetos que nos permite definir una clase particular basándonos en una clase más general.

La nueva clase definida **hereda** las propiedades (atributos y métodos) de la clase general.

**Sólo podemos hacer uso de la herencia
cuando nuestras clases cumplan con la
siguiente relación:**

{ *TODOS LOS _____ SON _____* }

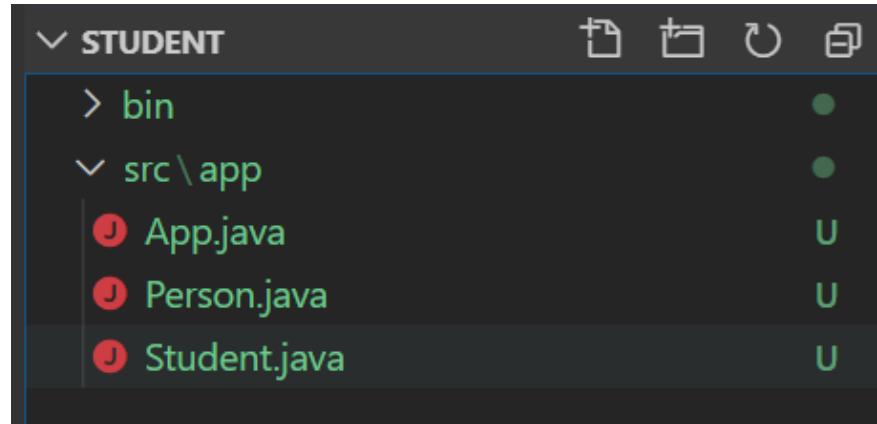
Herencia

Ejemplo:

- Todos los Perros son Animales.
- Todos los Pikachu son Pokémon.
- Todos los Alumnos son Personas.
- Todos los HEB son Tiendas.

Antes...

Asegúrate que las clases tengan visibilidad entre ellas.



```
public class Person {  
    private String name;  
  
    public Person(){  
        this.name = "No name yet";  
    }  
  
    public Person(String name){  
        this.name = this.name;  
    }  
  
    public void writeOutput(){  
        System.out.println("Name: " + this.name);  
    }  
  
    public boolean hasSameName(Person otherPerson){  
        return this.name.equalsIgnoreCase(otherPerson.name)  
    }  
}
```

```
public String getName() {  
    return name;  
}  
  
public void setName(String newName) {  
    this.name = name;  
}
```

```
public class Student extends Person{  
  
    private int studentNumber;  
  
    public Student(){  
        super();  
        studentNumber = 0;  
    }  
  
    public Student(String name, int studentNumber){  
        super();  
        this.setName(name);  
        this.studentNumber = studentNumber;  
    }  
  
    public void writeOutput(){  
        System.out.println("Name: " + this.getName());  
        System.out.println("Student Number: " +  
                           this.studentNumber);  
    }  
}
```

```
public int getStudentNumber() {  
    return studentNumber;  
}  
  
public void setStudentNumber(int studentNumber) {  
    this.studentNumber = studentNumber;  
}  
  
public boolean equals(Student other){  
    return this.hasSameName(other) &&  
           (this.studentNumber == other.studentNumber);  
}
```

Para indicar que una clase hereda a otra, utilizamos la palabra reservada **extends**.

Para indicar que una clase hereda a otra, utilizamos la palabra reservada **extends**.

Nomenclatura

Clase Base

- *Superclase / superclass*
- *Clase padre*
- *Clase Ancestro*

Clase Heredada

- *Subclase / subclass*
- *Clase hijo*
- *Clase derivada*

"Receta" para Herencia
Diseña las clases

"Receta" para Herencia
Especifica los modificadores de acceso en la clase base

"Receta" para Herencia
Define los constructores

"Receta" para Herencia
Agrega miembros y acciones

"Receta" para Herencia
Redefine y esconde los métodos necesarios

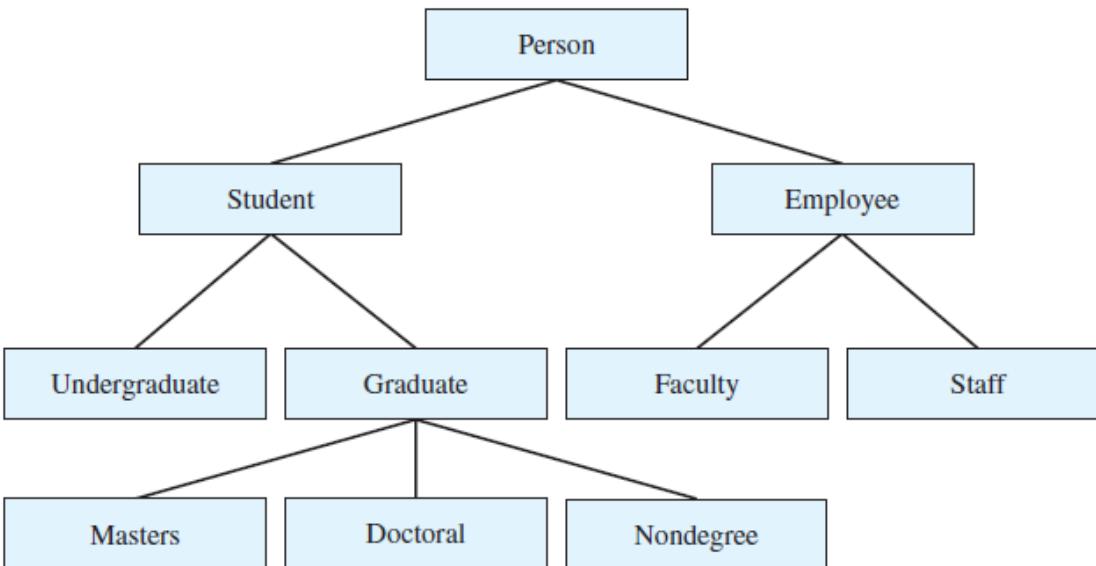
"Receta" para Herencia
Limita redefiniciones

“Receta” para Herencia

Diseña las clases

#1 Diseña las clases

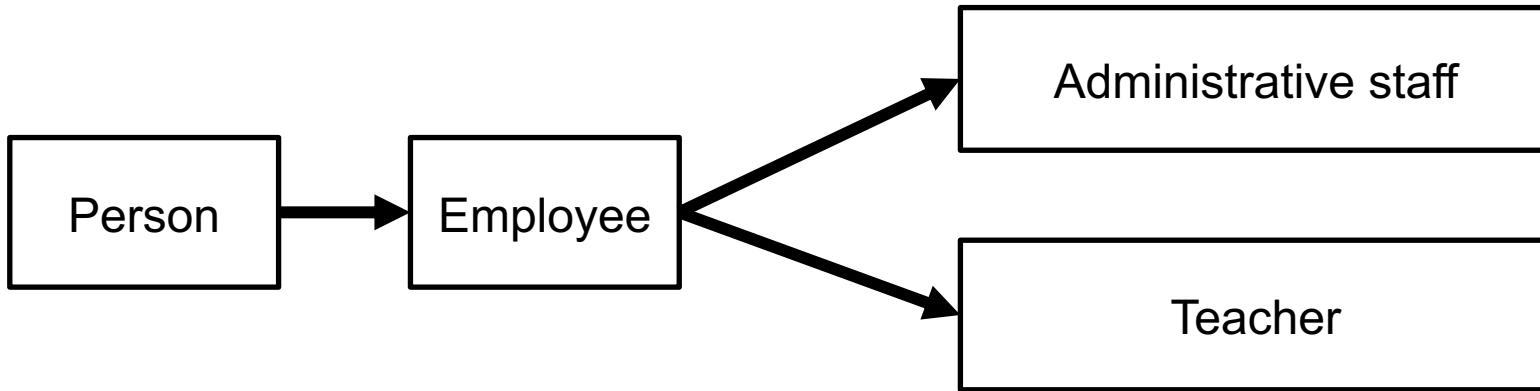
Diseña la clase base y la subclase. Identifica los componentes que pertenecen a cada clase.



Person es la **clase base** (o **superclase**) de Student.

Undergraduate es la **subclase** (o clase derivada) de Student.

#1 Diseña las clases

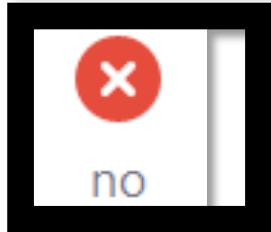


El objetivo principal de la herencia será definir funcionalidad general en cada clase para que puedan ser reutilizadas más adelante.

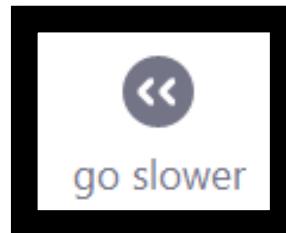
String[] cursosEnBlackboard



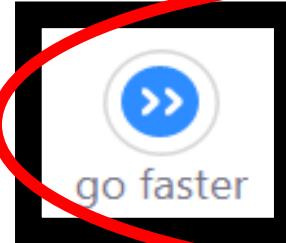
Person



Employee



Administrative Staff

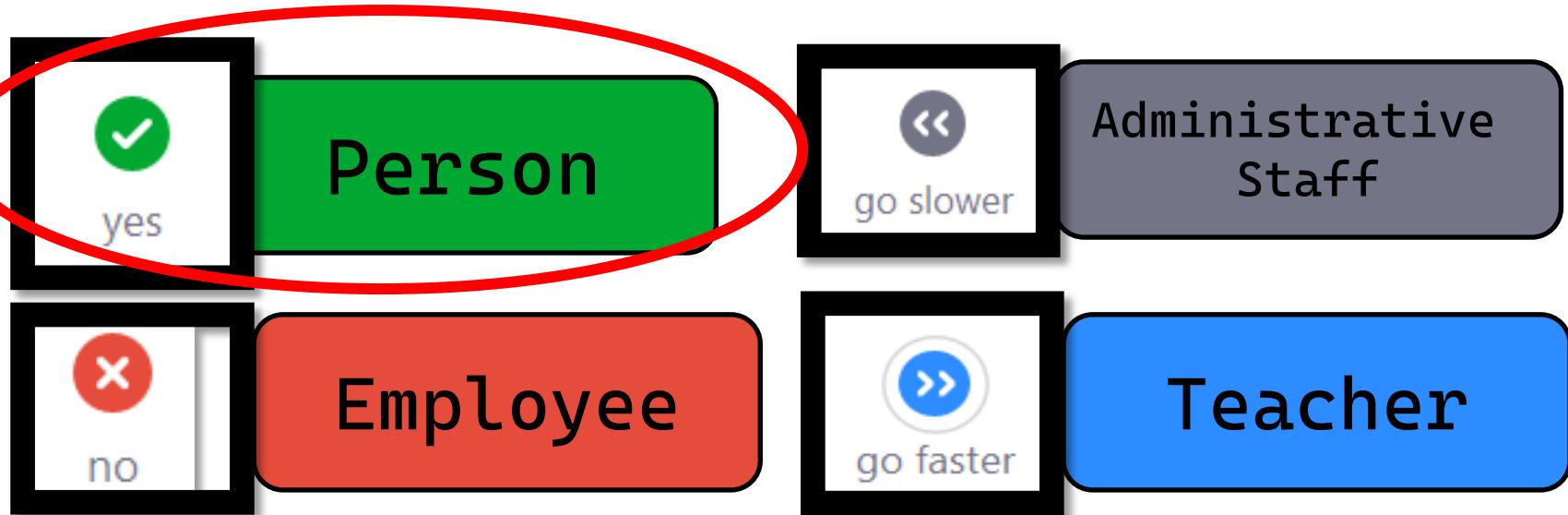


Teacher

int direccion



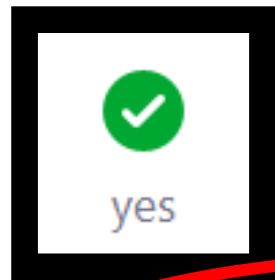
String email



printDocument()



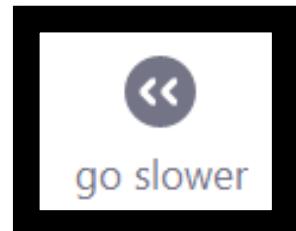
int numeroNomina



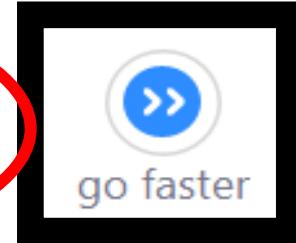
Person



Employee



Administrative
Staff

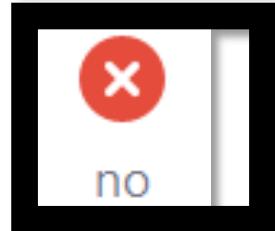


Teacher

inscribeAlumno()



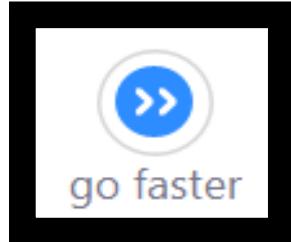
Person



Employee



Administrative
Staff

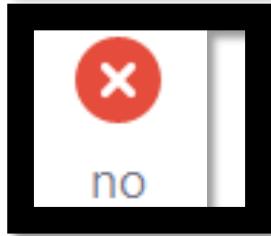


Teacher

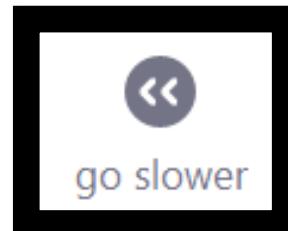
gradeExam()



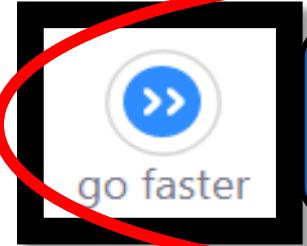
Person



Employee



Administrative Staff



Teacher

String cuentaCLABE



compareTo()



println()



“Receta” para Herencia

Especifica los
modificadores de acceso
en la clase base

#2 Modificadores de Acceso

Especifica los modificadores de acceso necesarios para cada variable y método. Limita las visibilidades a “lo mínimo necesario” para utilizar la clase.

	Class	Package	Subclass (same package)	Subclass (diff package)	Outside Class
public	Yes	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	Yes	No
default	Yes	Yes	Yes	No	No
private	Yes	No	No	No	No

```
public class Bicycle {  
    protected int cadence;  
    public int gear;  
    private int speed;  
  
    public Bicycle(int startCadence, int startSpeed, int startGear) {  
        gear = startGear;  
        cadence = startCadence;  
        speed = startSpeed;  
    }  
  
    protected void setCadence(int newValue) {  
        cadence = newValue;  
    }  
  
    private void setGear(int newValue) {  
        gear = newValue;  
    }  
}
```

```
public class MountainBicycle extends Bicycle {  
  
    // the MountainBike subclass adds one field  
    public int seatHeight;  
  
    // the MountainBike subclass has one constructor  
    public MountainBicycle(int startHeight, int startCadence,  
                           int startSpeed,  
                           int startGear) {  
        super(startCadence, startSpeed, startGear);  
        seatHeight = startHeight;  
    }  
  
    // the MountainBike subclass adds one method  
    public void setHeight(int newValue) {  
        seatHeight = newValue;  
    }  
}
```

<<Java Class>>

Bicycle

bicycles

- ◊ cadence: int
- gear: int
- speed: int

●^cBicycle(int,int,int)

- ◊ setCadence(int):void
- setGear(int):void
- applyBrake(int):void
- speedUp(int):void

<<Java Class>>

MountainBicycle

bicycles

- seatHeight: int

●^cMountainBicycle(int,int,int,in
● setHeight(int):void

#2 Modificadores de Acceso

Clase Bicycle

- Todos los métodos definidos en la clase `Bicycle`.
- Todos los atributos definidos en la clase `Bicycle`.

Clase MountainBicycle heredando Bicycle

- Todos los métodos public y protected definidos en la clase `Bicycle`.
- Todos los atributos public y protected definidos en la clase `Bicycle`.
- El constructor de la clase `Bicycle` utilizando la instrucción `super();`
- Todos los métodos definidos en la clase `MountainBicycle`.
- Todos los atributos definidos en la clase `MountainBicycle`.

“Receta” para Herencia

Define los constructores

#3 Define los constructores

Define los constructores de la clase base y la clase heredada. La primera instrucción del constructor de la clase heredada debe ser una llamada al constructor de la clase base utilizando super().

Clase Base

```
public class Person {  
    private String name;  
  
    public Person(){  
        this.name = "No name yet";  
    }  
    //...  
}
```

Clase Heredada

```
public class Student extends Person{  
    private int studentNumber;  
  
    public Student(){  
        super();  
        studentNumber = 0;  
    }  
    //...  
}
```

“Receta” para Herencia

**Agrega miembros y
acciones**

#4 Agrega miembros y acciones

¿Qué atributos adicionales a lo heredado requieres implementar en la subclase? Puedes agregar:

- Métodos con modificador de acceso privados, públicos o protegidos.
- Atributos con modificador de acceso privados, públicos o protegidos.

“Receta” para Herencia

**Redefine y esconde los
métodos necesarios**

#5 Redefine lo necesario

Observa los **métodos de instancia públicos o protegidos** de la clase base. ¿Requieren alguna modificación en su comportamiento al ser heredados?

Redefine (*overriding* en inglés) estos métodos en caso de ser necesario.

<<Java Class>>

Bicycle

bicycles

- ◊ cadence: int
- gear: int
- speed: int

●^cBicycle(int,int,int)

- ◊ setCadence(int):void
- setGear(int):void
- applyBrake(int):void
- speedUp(int):void

<<Java Class>>

MountainBicycle

bicycles

- seatHeight: int

●^cMountainBicycle(int,int,int,in
● setHeight(int):void

#5 Redefine lo necesario

```
public class Bicycle {  
    protected int cadence;  
    public int gear;  
    private int speed;  
  
    // the rest of the definition  
  
    public void reset() {  
        this.cadence = 0;  
        this.gear = 0;  
        this.speed = 0;  
    }  
}
```

```
public class MountainBicycle  
    extends Bicycle {  
    public int seatHeight;  
    // the rest of the definition  
  
    @Override  
    public void reset() {  
        super.reset();  
        this.seatHeight = 0;  
    }  
}
```

¿Hace sentido redefinir un método estático?



 yes

SI

 go slower

WHAT?

 no

NO

#5 Redefine y esconde lo necesario

```
public class Animal {  
    public static void testClassMethod() {  
        System.out.println("The static method in Animal");  
    }  
    public void testInstanceMethod() {  
        System.out.println("The instance method in Animal");  
    }  
}
```

```
public class Cat extends Animal {  
    public static void testClassMethod() {  
        System.out.println("The static method in Cat");  
    }  
    public void testInstanceMethod() {  
        System.out.println("The instance method in Cat");  
    }  
}
```

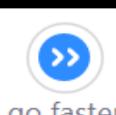
```
public static void main(String[] args) {  
    Cat myCat = new Cat();  
    Animal myAnimal = myCat;  
    Animal.testClassMethod();  
    myAnimal.testInstanceMethod();  
}
```

The static method in Cat
The instance method in Cat

The static method in Animal
The instance method in Animal

The static method in Animal
The instance method in Cat

The static method in Cat
The instance method in Animal



#5 Redefine y esconde lo necesario

Cuando declaramos un método estático en una subclase con la misma firma que un método heredado de su clase base, estamos **escondiendo (hiding)** el método de la clase base.

Los métodos estáticos pueden ser heredados pero nunca redefinidos por la forma en la que se llaman (nombre de la clase + nombre del método).

“Receta” para Herencia

Limita redefiniciones

#6 Limita redefiniciones

Existe algún método que no debamos permitir que sea redefinido? Piensa en algún método cuya implementación sea crítica para el correcto funcionamiento de la clase por cuestiones de integridad, seguridad, etc.

Utiliza el modificador **final** para evitar que estos métodos puedan ser redefinidos.

```
public class Website {  
  
    protected boolean authenticated;  
  
    public Website() {  
        this.authenticated = false;  
    }  
  
    public void authenticate(String password) {  
        if (password.equals("password")) {  
            this.authenticated = true;  
            System.out.println("Success: You have been authenticated");  
        } else {  
            this.authenticated = false;  
            System.out.println("Error: Try again");  
        }  
    }  
  
}  
  
public class Facebook extends Website {  
  
    public void authenticate(String password) {  
        this.authenticated = true;  
        System.out.println("Success: You have been authenticated");  
    }  
  
}
```

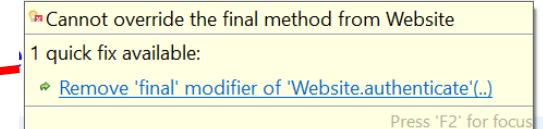
La clase Facebook hereda el método authenticate y lo redefine, eliminando funcionalidad crítica para la clase Website.

Esto sería un error porque estamos ignorando la verificación de seguridad que tenía authenticate en la clase Website.

```
public class Website {  
  
    protected boolean authenticated;  
  
    public Website() {  
        this.authenticated = false;  
    }  
  
    public final void authenticate(String password) {  
        if (password.equals("password")) {  
            this.authenticated = true;  
            System.out.println("Success: You have been authenticated");  
        } else {  
            this.authenticated = false;  
            System.out.println("Error: Try again");  
        }  
    }  
}
```

La solución es evitar la redefinición del método **authenticate()** agregando el modificador **final**.

```
public class Facebook extends Website {  
  
    public void authenticate(String password) {  
        this.authenticated = true;  
        System.out.println("Success: You have been authenticated");  
    }  
}
```



Esto evitará que las clases derivadas de Website redefinan el método **authenticate**.