Claro, aquí tienes un ejemplo simple de **sobrecarga de métodos** en Java:

java

Copiar

public class Calculadora {

// Método para sumar dos enteros

public int sumar(int a, int b) {

return a + b;

}

// Sobrecarga del método para sumar tres enteros

public int sumar(int a, int b, int c) {

return a + b + c;

}

// Sobrecarga del método para sumar dos números decimales (tipo double)

public double sumar(double a, double b) {

return a + b;

}

public static void main(String[] args) {

Calculadora calculadora = new Calculadora();

// Llamada al método con dos enteros

System.out.println("Suma de dos enteros: " + calculadora.sumar(5, 10));

// Llamada al método con tres enteros

System.out.println("Suma de tres enteros: " + calculadora.sumar(5, 10, 15));

// Llamada al método con dos números decimales

System.out.println("Suma de dos decimales: " + calculadora.sumar(5.5, 10.5));

}

}

**Explicación:**

* En este ejemplo, la clase Calculadora tiene tres métodos llamados sumar, pero con diferentes **firmas**:
  + Uno que suma dos enteros (int).
  + Otro que suma tres enteros.
  + Y otro que suma dos números decimales (double).
* La sobrecarga ocurre porque cada método tiene el mismo nombre, pero diferentes tipos o número de parámetros.

**Salida esperada:**

yaml

Copiar

Suma de dos enteros: 15

Suma de tres enteros: 30

Suma de dos decimales: 16.0

Este es un ejemplo sencillo de cómo se puede sobrecargar un método en Java, permitiendo que un mismo nombre de método pueda funcionar con diferentes tipos o cantidades de parámetros.

¡Por supuesto! Aquí tienes un ejemplo corto de **herencia** en Java:

java

Copiar

// Clase base (superclase)

class Animal {

public void hacerSonido() {

System.out.println("El animal hace un sonido.");

}

}

// Clase derivada (subclase) que hereda de Animal

class Perro extends Animal {

// Método sobrescrito

@Override

public void hacerSonido() {

System.out.println("El perro ladra.");

}

}

public class EjemploHerencia {

public static void main(String[] args) {

// Crear un objeto de la clase Perro

Perro perro = new Perro();

perro.hacerSonido(); // Llamará al método sobrescrito en Perro

// Crear un objeto de la clase Animal

Animal animal = new Animal();

animal.hacerSonido(); // Llamará al método de la clase Animal

}

}

### Explicación:

* **Animal** es la superclase (o clase base), y tiene un método hacerSonido().
* **Perro** es la subclase que hereda de Animal y **sobrescribe** el método hacerSonido() para proporcionar una implementación específica.
* En el método main, se crea un objeto de tipo Perro, y cuando se llama a hacerSonido(), se usa la versión sobrescrita del método en Perro.

### Salida esperada:

nginx

Copiar

El perro ladra.

El animal hace un sonido.

Este es un ejemplo básico de **herencia** y **sobrescritura** de métodos en Java.

¡Claro! Aquí tienes un ejemplo simple de **polimorfismo** en Java:

java

Copiar

// Clase base (superclase)

class Animal {

public void hacerSonido() {

System.out.println("El animal hace un sonido.");

}

}

// Clase derivada (subclase) que hereda de Animal

class Perro extends Animal {

@Override

public void hacerSonido() {

System.out.println("El perro ladra.");

}

}

// Otra clase derivada (subclase)

class Gato extends Animal {

@Override

public void hacerSonido() {

System.out.println("El gato maulla.");

}

}

public class EjemploPolimorfismo {

public static void main(String[] args) {

// Polimorfismo: se usa una referencia de tipo Animal

// para referirse a objetos de clases derivadas.

Animal miAnimal = new Animal();

Animal miPerro = new Perro();

Animal miGato = new Gato();

// Llamada al método hacerSonido() con distintos objetos

miAnimal.hacerSonido(); // Llama a la versión de Animal

miPerro.hacerSonido(); // Llama a la versión sobrescrita en Perro

miGato.hacerSonido(); // Llama a la versión sobrescrita en Gato

}

}

### Explicación:

* **Polimorfismo** en Java permite que una referencia de tipo **superclase** (en este caso Animal) se use para hacer referencia a objetos de clases derivadas (como Perro y Gato).
* Los objetos miPerro y miGato son de tipo Animal, pero al ejecutarse el programa, Java ejecutará los métodos sobrescritos específicos de Perro y Gato gracias al **polimorfismo**.

### Salida esperada:

nginx

Copiar

El animal hace un sonido.

El perro ladra.

El gato maulla.

El **polimorfismo** permite que un mismo método, como hacerSonido(), se comporte de manera diferente según el objeto en el que se llame, a pesar de que la referencia sea de tipo **superclase**.

¡Aquí tienes un ejemplo sencillo de **encapsulamiento** en Java!

El **encapsulamiento** es un principio de la programación orientada a objetos que restringe el acceso directo a los atributos de un objeto y permite su modificación solo a través de métodos específicos (getters y setters).

java

Copiar

// Clase con encapsulamiento

class Persona {

// Atributos privados

private String nombre;

private int edad;

// Getter para obtener el nombre

public String getNombre() {

return nombre;

}

// Setter para establecer el nombre

public void setNombre(String nombre) {

this.nombre = nombre;

}

// Getter para obtener la edad

public int getEdad() {

return edad;

}

// Setter para establecer la edad

public void setEdad(int edad) {

if (edad > 0) { // Verificación para evitar una edad inválida

this.edad = edad;

} else {

System.out.println("Edad no válida.");

}

}

}

public class EjemploEncapsulamiento {

public static void main(String[] args) {

// Crear un objeto de la clase Persona

Persona persona = new Persona();

// Usar los métodos setters para establecer valores

persona.setNombre("Juan");

persona.setEdad(25);

// Usar los métodos getters para obtener valores

System.out.println("Nombre: " + persona.getNombre());

System.out.println("Edad: " + persona.getEdad());

}

}

### Explicación:

* **Atributos privados**: Los atributos nombre y edad están marcados como private, lo que significa que no pueden ser accedidos directamente desde fuera de la clase.
* **Métodos públicos** (getters y setters): Se crean métodos public para acceder y modificar los atributos de manera controlada. Los métodos getNombre() y setNombre() permiten acceder y modificar el atributo nombre, y lo mismo ocurre con edad.
* **Control de acceso**: En el setter de edad, se agrega una validación para asegurarse de que solo se pueda establecer una edad positiva.

### Salida esperada:

makefile

Copiar

Nombre: Juan

Edad: 25

Este es un ejemplo básico de **encapsulamiento**, que permite proteger los datos internos de la clase y asegurarse de que solo se accedan de forma controlada mediante métodos públicos.

Solid

**Ejemplo sin aplicar SRP:**

java

Copiar

public class Empleado {

private String nombre;

private double salario;

public Empleado(String nombre, double salario) {

this.nombre = nombre;

this.salario = salario;

}

public double calcularSalario() {

return salario;

}

public void guardarEmpleadoEnBaseDeDatos() {

// Código para guardar el empleado en la base de datos

}

public void enviarCorreoBienvenida() {

// Código para enviar un correo de bienvenida

}

}

En este caso, la clase Empleado tiene varias responsabilidades:

* Calcular el salario.
* Guardar el empleado en la base de datos.
* Enviar un correo de bienvenida.

**Ejemplo aplicando SRP:**

java

Copiar

public class Empleado {

private String nombre;

private double salario;

public Empleado(String nombre, double salario) {

this.nombre = nombre;

this.salario = salario;

}

public double calcularSalario() {

return salario;

}

}

public class BaseDeDatos {

public void guardarEmpleado(Empleado empleado) {

// Código para guardar el empleado en la base de datos

}

}

public class Correo {

public void enviarCorreoBienvenida(Empleado empleado) {

// Código para enviar un correo de bienvenida

}

}

Ahora, cada clase tiene una única responsabilidad:

1. **Empleado**: Solo se encarga de la información del empleado y de calcular su salario.
2. **BaseDeDatos**: Se encarga de guardar al empleado en la base de datos.
3. **Correo**: Se encarga de enviar un correo de bienvenida al empleado.

**Beneficios de este enfoque:**

* Si necesitas cambiar la forma de almacenar los empleados en la base de datos, puedes modificar solo la clase BaseDeDatos, sin afectar a la lógica de Empleado ni a la clase Correo.
* Lo mismo ocurre si deseas cambiar la forma en que se envían los correos electrónicos, solo modificas la clase Correo.
* El código es más limpio y más fácil de mantener.

Este enfoque facilita el mantenimiento y la extensión de tu código a medida que el proyecto crece.

**1. Ejemplo de Interface**

Supongamos que estamos modelando vehículos, y queremos que diferentes tipos de vehículos tengan la capacidad de moverse. Usamos una **interfaz** para definir esta capacidad.

java

Copiar

// Definimos la interfaz

interface Vehiculo {

void mover(); // Método abstracto

}

// Implementamos la interfaz en clases concretas

class Coche implements Vehiculo {

@Override

public void mover() {

System.out.println("El coche se está moviendo");

}

}

class Bicicleta implements Vehiculo {

@Override

public void mover() {

System.out.println("La bicicleta se está moviendo");

}

}

public class Main {

public static void main(String[] args) {

Vehiculo coche = new Coche();

coche.mover(); // Imprime: El coche se está moviendo

Vehiculo bicicleta = new Bicicleta();

bicicleta.mover(); // Imprime: La bicicleta se está moviendo

}

}

**Características:**

* **Contrato**: La interfaz Vehiculo define el contrato de que cualquier clase que la implemente debe tener un método mover().
* **Implementación flexible**: Coche y Bicicleta implementan este contrato de manera diferente.

**2. Ejemplo de Clase Abstracta**

Ahora, supongamos que tenemos una jerarquía de **Empleados**, donde algunos empleados trabajan a tiempo completo y otros a tiempo parcial, pero todos comparten algunos comportamientos comunes. Usamos una **clase abstracta** para definir estos comportamientos comunes.

java

Copiar

// Clase abstracta

abstract class Empleado {

String nombre;

public Empleado(String nombre) {

this.nombre = nombre;

}

// Método común para todos los empleados

public void mostrarInformacion() {

System.out.println("Empleado: " + nombre);

}

// Método abstracto que debe ser implementado por las subclases

public abstract double calcularSalario();

}

// Subclase de Empleado

class EmpleadoTiempoCompleto extends Empleado {

double salarioFijo;

public EmpleadoTiempoCompleto(String nombre, double salarioFijo) {

super(nombre);

this.salarioFijo = salarioFijo;

}

@Override

public double calcularSalario() {

return salarioFijo;

}

}

// Subclase de Empleado

class EmpleadoPorHora extends Empleado {

double horasTrabajadas;

double tarifaPorHora;

public EmpleadoPorHora(String nombre, double horasTrabajadas, double tarifaPorHora) {

super(nombre);

this.horasTrabajadas = horasTrabajadas;

this.tarifaPorHora = tarifaPorHora;

}

@Override

public double calcularSalario() {

return horasTrabajadas \* tarifaPorHora;

}

}

public class Main {

public static void main(String[] args) {

Empleado emp1 = new EmpleadoTiempoCompleto("Juan", 3000);

Empleado emp2 = new EmpleadoPorHora("Ana", 40, 15);

emp1.mostrarInformacion(); // Imprime: Empleado: Juan

System.out.println("Salario: " + emp1.calcularSalario()); // Imprime: Salario: 3000

emp2.mostrarInformacion(); // Imprime: Empleado: Ana

System.out.println("Salario: " + emp2.calcularSalario()); // Imprime: Salario: 600

}

}

**Características:**

* **Comportamiento común**: La clase Empleado proporciona el método mostrarInformacion(), que es compartido por todas las subclases.
* **Método abstracto**: Cada tipo de empleado (tiempo completo o por hora) tiene su propia implementación del método calcularSalario().
* **Herencia**: Las subclases EmpleadoTiempoCompleto y EmpleadoPorHora heredan de Empleado y proporcionan su propia implementación del método abstracto calcularSalario().