

DAM

PROGRAMACIÓN



UD3-3-POO

CONCEPTOS BÁSICOS II

Sobrecarga de métodos (Overloading)

- Los métodos sobrecargados **se llaman igual**, pero **tiene** que **cambiar el número de parámetros y/o el tipo** de alguno de ellos dentro de una misma clase,
- Un método sobrecargado **puede cambiar el valor de retorno** y el modificador de acceso, **pero este cambio no lo convierte en sobrecargado**.
- Cuando se llama a un método que está sobrecargado se elige el método a ejecutar según los parámetros que se pasen al método.
- A continuación vemos un ejemplo de sobrecarga dentro de una misma clase.



```
public class Sobrecarga {
```

```
    void demoSobrec() {
```

```
        System.out.println("Sin parámetros\n");
```

```
    }
```

```
//Sobrecargando demoSobrec para un parámetro int
```

```
void demoSobrec(int a) {
```

```
    System.out.println("Un parámetro: " +a+"\n");
```

```
}
```

```
//Sobrecargando demoSobrec para dos parámetros int
```

```
int demoSobrec(int a, int b) {
```

```
    System.out.println("Dos parámetros: "+a+", "+b);
```

```
    return a+b;
```

```
}
```

```
//Sobrecargando demoSobrec para dos parámetros double
```

```
double demoSobrec(double a, double b) {
```

```
    System.out.println("Parámetros double: "+a+", "+b);
```

```
    return a+b;
```

```
}
```

```
}
```

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

```
    int sumaint;  
    double sumadouble;
```

```
    Sobre carga sc= new Sobre carga ();
```

```
    //Llamando todas las versiones de demoSobrec  
    sc.demoSobrec ();
```

```
    sc.demoSobrec (2);
```

```
    sumaint=sc.demoSobrec (4,6);
```

```
    System.out.println("Resultado de demoSobrec(4,6) es: "+sumaint+"\n");
```

```
    sumadouble=sc.demoSobrec (1.1,2.2);
```

```
    System.out.println("Resultado de demoSobrec(1.1,2.2) es: "+sumadouble);
```

```
}
```

Llamadas a los
métodos
sobrecargados

Sin parámetros

Un parámetro: 2

Dos parámetros: 4, 6

Resultado de demoSobrec(4,6) es: 10

Dos parámetros tipo double: 1.1, 2.2

Resultado de demoSobrec(1.1,2.2) es: 3.3000000000000003

Resultado de la
ejecución

Métodos abstractos

- **Un método abstracto es un método declarado pero no implementado.**
- En un método abstracto tan solo se indica:
 - **Nombre.**
 - **Parámetros.**
 - **Tipo a devolver.**
 - **Modificadores de acceso.**
- Un método abstracto **se escribe sin llaves ({}) y con un ; al final** de la declaración. Esta información del método se llama **firma del método**.

No pueden ser abstractos:

- Los constructores
- Los métodos estáticos
- Los métodos privados

```
[modificador] abstract tipoDevuelto nombreMetodo([parámetros]);
```

```
public abstract double getPerimetro();
```

- Un método se declara como abstracto porque en ese nivel de la jerarquía de clases no se tiene aún la información de cómo se va a implementar.

Clases abstractas

- Todas las **clases** que tengan algún **método abstracto** debe ser declarada también como **abstracta**.
- **Una clase abstracta no tiene porque tener métodos abstractos.**
- Las clases abstractas **no permite crear objetos a partir de ella aunque sí pueden tener constructores.**
- *Los constructores de una clase abstracta son heredados por sus clases derivadas y se ejecutan cuando se ejecuta el constructor de las mismas.*
- Una clase abstracta es una clase que **no tiene instancias**. Su utilidad consiste en proveer una estructura y comportamiento común a todas las subclases que heredan de ella.

```
public abstract class Figura {  
    public abstract double getArea();  
    public abstract double getPerimetro();  
}
```

Usa una clase abstracta cuando:

- quieras definir una **plantilla** para un grupo de subclases, y tengas algún código de implementación que todas las clases puedan usar.
- quieras garantizar que nadie va a hacer objetos de esa clase.

Ejemplo 1

```
public abstract class FiguraGeometrica {  
    public abstract double area();  
}
```

Método abstracto que será implementado en las clases hijas. En el método abstracto solo se indica su firma

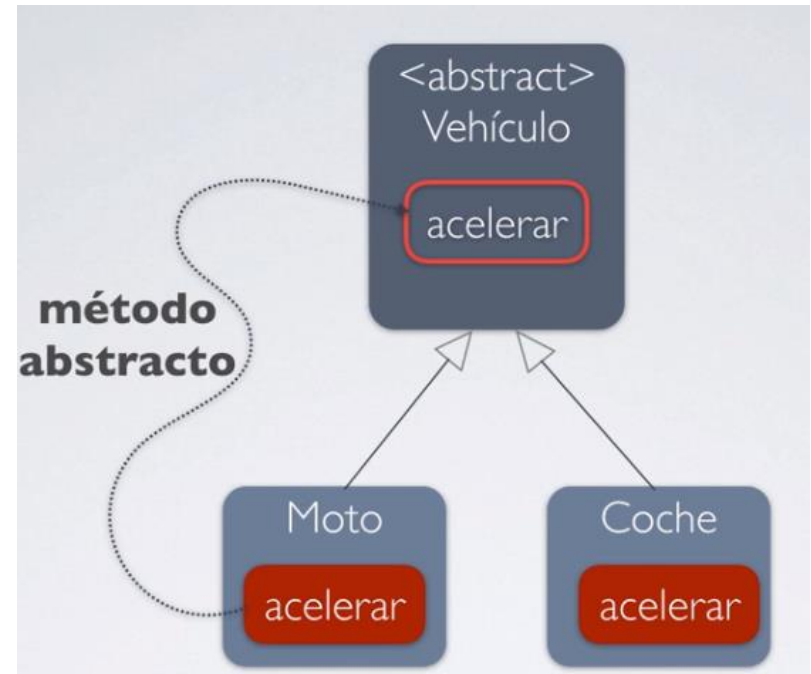
```
public class Rectangulo extends FiguraGeometrica {  
  
    double base;  
    double altura;  
  
    public Rectangulo(double base, double altura) {  
        this.base = base;  
        this.altura = altura;  
    }  
  
    @Override  
    public double area() {  
        return base * altura;  
    }  
}
```




Declaración del método abstracto de la clase padre

Ejemplo 2




Declararemos el método ***acelerar*** como método abstracto en la clase **Vehículo**, este método no tendrá código, solo cabecera. y todas las clases hijas lo deben implementar, salvo que ellas sean también abstractas

```
public abstract class Vehiculo {  
    private String marca;  
    public Vehiculo(String marca) {  
        super();  
        this.marca = marca;  
    }  
    public String getMarca() {  
        return marca;  
    }  
    public void setMarca(String marca) {  
        this.marca = marca;  
    }  
  
    public abstract void acelerar(); //Método abstracto  
}
```





```
public class Moto extends Vehiculo {  
  
    public Moto(String marca) {  
        super(marca);  
    }  
  
    public void acelerar() {  
        System.out.println("la moto acelera");  
    }  
}
```



```
public class Coche extends Vehiculo {  
  
    public Coche(String marca) {  
        super(marca);  
    }  
  
    public void acelerar() {  
        System.out.println("el coche acelera");  
    }  
}
```

Ahora se puede usar polimorfismo al construir un método en el programa principal de la siguiente forma:

```
public class Principal {  
  
    public static void main(String[] args) {  
  
        Vehiculo m=new Moto ("ducati");  
        Vehiculo c= new Coche("toyota");  
  
        acelerarVehiculo(m);  
        acelerarVehiculo(c);  
  
        //Las clases abstractas no se pueden instanciar (Vehiculo  
        //Vehiculo v= new Vehiculo("mercedes");  
    }  
  
    /*Método que usa polimorfismo: El programador que desarrolla este código no necesita  
    conocer la jerarquía de clases*/  
    public static void acelerarVehiculo(Vehiculo v) {  
        v.acelerar();  
    }  
}
```

Practica 1

Dada las siguientes clases:

1. Crear un método que permita introducir el tiempo de alquiler antes de emitir la factura.
2. Implementar un método *main* que solicite por teclado el valor del tiempo y el tipo de vehículo (cliente o abonado) y en función de esos datos muestre el precio a pagar de la factura. Se debe utilizar polimorfismo.

```
abstract public class Vehiculo {  
    protected String id;  
    protected int tiempo;  
  
    public Vehiculo ( String id ) {  
        this.id = id;  
        tiempo = 0;  
    }  
  
    abstract public double factura ();  
}
```

3. Almacenar la información de las distintas facturas en un ArrayList.

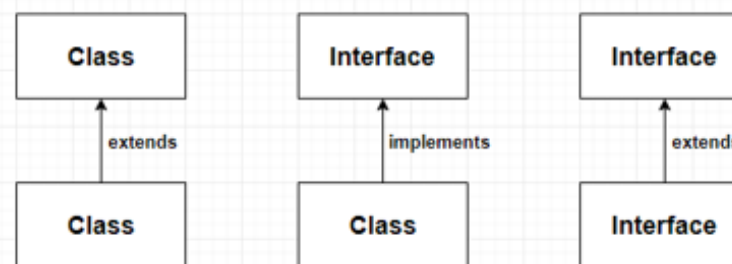
```
public class VehiculoCliente extends Vehiculo {  
    ...  
    public double factura () {  
        return (tiempo / 24) * 12.0 + (tiempo % 24) * 0.6;  
    }  
}
```

```
public class VehiculoAbonado extends Vehiculo {  
    ...  
    public double factura () {  
        return tiempo * 200.0;  
    }  
}
```

- Modificar el proyecto para incluir un nuevo método en las clases **VehiculoAbonado**.
 - Este método será un método sobrecargado de **public double factura()** que permitirá un parámetro de tipo String llamado **codigo** . Este método descontará 3% de la factura total si el código coincide con uno de los datos.
 - Los objetos de la clase **VehiculoAbonado** deben almacenar información de si utilizaron un código de descuento o no.
 - Explicar todos los cambios que se deben realizar para poder usar este método en **main**.
- Agregar la opción al menú para **mostrar todas las facturas e indicando, para cada una de ellas, si se ha aplicado el descuento o no.**
- Se debe tener en cuenta que el descuento sólo se aplica a VehiculoAbonado y que puede haber facturas de VehiculoAbonado a las que no se le ha aplicado ningún descuento.

Interfaces

- Una interface define una forma estándar y pública de especificar el comportamiento de otras clases.
- Todos los **métodos** de una interface son **métodos abstractos y públicos** (no es necesario especificarlo).
- Todos los **atributos** son de tipo **final y public** (no es necesario especificarlo), es decir, constantes y públicos. Hay que darles un valor inicial.
- Las clases no heredan, sino implementan (**implements**) las interfaces.
- **Las clase que implementan las interfaces están obligadas a implementar todos sus métodos.**
- Las clases pueden “implementar” los métodos de una interfaz con métodos abstractos.
- Las interfaces permiten la implementación de clases con comportamientos comunes, sin importar su ubicación en la jerarquía de clases.
- **Las clases pueden implementar más de una interface a la vez.**
- Una interfaz puede heredar (**extends**) de otra interfaz.



Definición de Interfaces

- La sintaxis de una interface es:

```
public interface [NombreInterface] {  
    //Firmas de métodos  
}
```

- **Ejemplo:** Interface que define relaciones de dimensión entre dos objetos, el de la propia clase y el que se pasa por parámetro.

```
public interface RelacionDimension {  
  
    public boolean esMasGrande (Object b) ;  
  
    public boolean esMasPequeno (Object b) ;  
  
    public boolean esIgual (Object b) ;  
}
```

Implementación de interfaces

- Para crear una clase concreta que implementa una interface se utiliza la palabra clave **implements**.
- Una clase puede implementar varias interfaces.
- **Ejemplo:**

```
public class Linea implements RelacionDimension {

    //Propiedades
    private double x1;
    private double x2;
    private double y1;
    private double y2;

    //Constructor
    public Linea(double x1, double x2, double y1, double y2) {
        this.x1 = x1;
        this.x2 = x2;
        this.y1 = y1;
        this.y2 = y2;
    }

    //Métodos públicos
    public double getLongitud() {
        double length = Math.sqrt((x2 - x1) * (x2 - x1)
                                   + (y2 - y1) * (y2 - y1));
        return length;
    }
}
```

```






//Implementación de la interface
@Override
public boolean esMasGrande(Object b) {
    double aLen = this.getLongitud();
    //Casting de b que pertenece a Object
    //(Linea)b lo convierte en un objeto de la clase Linea
    double bLen = ((Linea) b).getLongitud();
    return (aLen > bLen);
}

@Override
public boolean esMasPequeno(Object b) {
    double aLen = this.getLongitud();
    double bLen = ((Linea) b).getLongitud();
    return (aLen < bLen);
}

@Override
public boolean esIgual(Object b) {
    double aLen = this.getLongitud();
    double bLen = ((Linea) b).getLongitud();
    return (aLen == bLen);
}
}

```

Al hacer casting a un objeto de una superclase donde hemos almacenado el objeto de una subclase, se puede acceder a los métodos definidos en el objeto de la subclase.



```
public static void main(String[] args) {  
  
    Linea l1=new Linea(2,3,6,7);  
    Linea l2=new Linea(2,3,3,3);  
    if (l1.esIgual( l2)){  
        System.out.println("Las dos líneas son iguales");  
    }else{  
        if (l1.esMasGrande(l2)) {  
            System.out.println("La línea1 es más grande que la línea2 ");  
        }else{  
            System.out.println("La línea1 es más pequeña que la línea2");  
        }  
    }  
}
```

Práctica:

- Consulta la siguiente [Web](#) y crea dos clases para medir las dimensiones de los perímetros de rombos y rectángulos.
- En el método **main**, pide los datos necesarios de dos rombos y dos rectángulos para que el programa devuelva la información de la relación dimensional de los pares de figuras.

Diferencias entre clase abstracta e interfaz

- **Una clase no puede heredar de varias clases**, aunque sean abstractas (herencia múltiple). Sin embargo **sí puede implementar una o varias interfaces** y además seguir heredando de una clase.
- **Una interfaz no puede definir métodos** (no implementa su contenido), tan solo los declara o enumera.
- **Una interfaz puede hacer que dos clases tengan un mismo comportamiento independientemente de sus ubicaciones** en una determinada jerarquía de clases (no tienen que heredar las dos de una misma superclase, pues no siempre es posible según la naturaleza y propiedades de cada clase).
- **Una interfaz permite establecer un comportamiento** de clase **sin** apenas **dar detalles**, pues esos detalles aún no son conocidos (dependerán del modo en que cada clase decida implementar la interfaz).
- **Las interfaces tienen su propia jerarquía**, diferente e independiente de la jerarquía de clases.
- **Una clase abstracta proporciona una interfaz** disponible sólo **a través** de la **herencia**. Sólo quien herede de esa clase abstracta dispondrá de esa interfaz.
- A partir de ahora podemos hablar de otra posible **relación entre clases**: la de **compartir un determinado comportamiento (interfaz)**. Tan solo cuando haya realmente una **relación** de tipo "**es un**" se producirá **herencia**.