Ciclo de Desarrollo de Aplicaciones Multiplataforma

Grado Superior

U.T.4.

NOCIONES AVANZADAS DE PROGRAMACIÓN ORIENTADA A OBJETOS

Módulo de Programación

Departamento de Informática

IES Monte Naranco

1. HERENCIA

1.1.- FUNDAMENTOS DE LA HERENCIA

La herencia es la característica más importante de la POO. Con la herencia se pretende crear nuevas clases aprovechado las características (atributos y métodos) de otras ya creadas, evitando así tener que volver a definirlas; por tanto con la herencia conseguimos:

a) **Reutilización de código:** se consigue cuando se definen clases que especializan a otras previamente definidas, que además de sus propios miembros (atributos y métodos) incluirán otros definidos en otra clase.

b) **Mantener aplicaciones existentes:** una vez creada una clase con una determinada funcionalidad, si hay la necesidad de ampliarla no se necesitaría modificar la clase existente (puede seguir siendo utilizada por el programa para el que fue diseñada), bastaría con crear una nueva clase que herede de ella adquiriendo toda su funcionalidad y añadiendo la suya propia.

La herencia nos define un nuevo concepto conocido como jerarquía de clases. Es una representación gráfica que nos indica que clase hereda de otra.

En terminología de POO la clase de la cual se hereda se denomina superclase, clase padre, o clase base, y la clase que hereda se denomina subclase, clase hija o clase derivada. Los métodos y atributos comunes o generales se pondrán en la superclase, y en las subclase las características específicas (proceso de jerarquización o generalización).

Para que una clase herede de otra usaremos la palabra reservada **extends.**

La declaración de una clase que hereda de otra es la siguiente:

public class *nombre_subclase* extends *nombre_superclase* {
//cuerpo de la subclase
}

Comenzaremos con un ejemplo corto para explicar esta idea. El siguiente programa crea una superclase A y una subclase denominada B.

```
package pEjemplo1; //un ejemplo simple de herencia
//creación de la superclase
public class A {
    protected int i,j;
```

```
        public void mostrar() {
                System.out.println("i y j: "+ i+" "+j);
        }
}
package pEjemplo1;
//creacion de la subclase

public class B extends A {
        private int k;
        public int getK() {
                return k;
        }
        public void setK(int k) {

                this.k=k;
        }
        public void visualizar() {

                System.out.println("Subclase i y j: "+i+" "+j);
        }
        public void mostrark() {

                System.out.println("k: "+k);
        }
        public void sum() {

                System.out.println("i+j+k: "+(i+j+k));
        }
}

package pEjemplo1;
public class HerenciaSimple {
        public HerenciaSimple() {
                A superOb =new A();
                B subOb=new B();
                System.out.println("Contenido de subclase");
                subOb.visualizar();
                superOb.i=10;
                superOb.j=20;
                System.out.println("Contenido de superOb");
```

```
            superOb.mostrar();
            System.out.println("Contenido de subclase");
            subOb.visualizar();
            System.out.println();
            subOb.i=7;
            subOb.j=8;
            subOb.setK(9);
            System.out.println("Contenido de subOb");
            subOb.mostrar();
            subOb.mostrark();
            System.out.println("suma de i, j y k en subOb:");
            subOb.sum();
            System.out.println();
        }

        public static void main(String[] args) {
            HerenciaSimple interfaz=new HerenciaSimple();
        }
}
```

La salida es la siguiente:

Contenido de subclase

Subclase i y j: 0 0

Contenido de superOb

i y j: 10 20

Contenido de subclase

Subclase i y j: 0 0

Contenido de subOb

i y j: 7 8

k: 9

suma de i,j y k en subOb:

i+j+k: 24

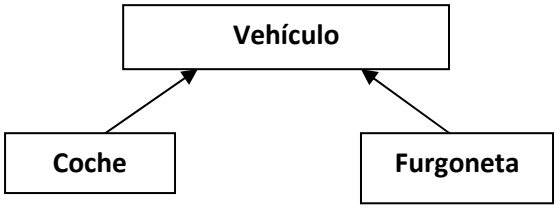
La subclase **B** incluye todos los miembros de su superclase **A** además de los suyos propios. Por esta razón mediante el puntero **subOb se** puede acceder a **i** y **j** y llamar a **mostrar ().**

Aunque **A** es una superclase para **B** es completamente independiente de **B**. El hecho de ser una superclase para una subclase no implica que no pueda ser utilizada por sí sola.

Regla para determinar la herencia de las clases

Se aplicará el término "Es un..." a nuestro diseño de clases. Preguntaremos si nuestra subclase es una superclase.

Ejm:



Así:

¿Un vehículo es un coche?. La respuesta sería No, puesto que un vehículo puede ser un coche, una furgoneta....

¿Un coche es un vehículo?. La respuesta sería Si. Significa que un coche es una subclase de vehículo y vehículo sería la superclase. La herencia está bien creada.

Es por ello que la relación de herencia se conoce también como relación is-a (es-un), ya que las instancias de las subclases son también de la superclase.

1.2.- ACCESO A LOS MIEMBROS HEREDADOS EN UNA SUBCLASE

Recordemos que los modos de acceso a los miembros de una clase son: private, public, protected y sin especificar.

1. Los miembros declarados como privados (private) en una superclase, no pueden ser accesibles directamente ni desde una subclase del mismo paquete, ni desde una subclase de otro paquete. Visibles solo en la clase donde se definen. Se accede a ellos a través de los métodos pertinentes si no están declarados como private.

2. Los miembros declarados como protegidos (protected) en una superclase, pueden ser accedidos directamente desde la propia clase donde están declarados, desde subclases del mismo o de otro paquete, y desde otra clase del mismo paquete.
3. Los miembros declarados como públicos (public) en una superclase, pueden ser accedidos desde la propia clase donde están declarados, desde una subclase del mismo u otro paquete y desde otra clase del mismo u otro paquete.
4. Los miembros en una superclase cuyo tipo de acceso está sin especificar (por defecto), pueden ser accedidos desde la propia clase donde están declarados, desde una subclase del mismo paquete y desde otra clase del mismo paquete, pero no pueden ser accedidos ni desde una subclase, ni desde otra clase de otro paquete diferente.

	Acceso desde una subclase del mismo paquete	**Acceso desde una subclase de otro paquete**	**Acceso desde una clase del mismo paquete**	**Acceso desde una clase de otro paquete**
private	NO	NO	NO	NO
public	SI	SI	SI	SI
sin especificar	SI	NO	SI	NO
protected	SI	SI	SI	NO

Durante la herencia cuando se crea un objeto de la subclase se reserva memoria en el mismo para todos los métodos y atributos de la superclase, incluidos los declarados como privados, aunque no se pueda acceder a ellos directamente desde la subclase (el acceso solo va a ser posible si la superclase tiene definidos métodos públicos o protegidos que lo permitan).

Existe una controversia por parte de los programadores, pues hay algunos que consideran que todos los miembros de la superclase son heredados por las clases derivadas o subclases al formar parte de sus objetos, aunque no se tenga acceso directo a los mismos, sin embargo otros opinan que una subclase solo hereda los miembros de la clase padre que están declarados como protegidos, públicos o no llevan ningún tipo de modificador de acceso (dependiendo si la clase hija está declarada en el mismo paquete o no que su clase padre), al poder ser accedidos de forma directa. Como excepción destacamos los constructores que no son heredables.

Veamos el siguiente ejemplo:

```
package pEjemplo2;
public class A {
        protected int i;
        private int j;
        public void setJ (int y) {
                j=y;
        }
        public void setI (int x) {
                i=x;
        }
}
```

```
package pEjemplo2;
public class B extends A {
        private int total;
        public void sum () {
                total=i +j; //ERROR aquí no se puede acceder a j directamente
                System.out.println(total);
        }
}
```

```
package pEjemplo2;
public class Main {
        public Main() {
                B subOb=new B();
                subOb.setI(10);
                subOb.setJ(12);
                System.out.print("Total es igual a  ");
                subOb.sum();
        }
        public static void main(String[] args) {
                Main interfaz=new Main();
        }
}
```

Este programa no se compilará porque la referencia a **j** dentro del método **sum()** de **B,** dará lugar a una violación de acceso. El atributo **j** está declarado como **private.**

Conclusión:

No se puede acceder directamente a aquellos miembros de la superclase que se hayan declarado como privados. El acceso dependerá de los permisos de visibilidad que tenga.

Una solución sería:

```
package pEjemplo2;

public class A {
        protected int i;
        private int  j;
        public void setJ (int y) {
                j=y;
        }
        public void setI (int x) {
                i=x;
        }
        public int getJ() {
                return j;
        }
}
```

```
package pEjemplo2;
public class B extends A{
        private int total;
        public void sum() {
                total=i+getJ();
                System.out.println(total);
        }
}
```

```
package pEjemplo2;
public class Main {
	public Main() {
		B subOb=new B();
		subOb.setI(10);
		subOb.setJ(12);
		//El método getJ() al ser púbico es heredado por la subclase B
		System.out.print("Total es igual a  ");
		subOb.sum();
	}
	public static void main(String[] args) {
		Main interfaz=new Main();
	}
}
```

Salida por pantalla:

Total es igual a 22

En herencia por lo general en la superclase los atributos son privados o protegidos (permiten encapsular un poco nuestros atributos), y los métodos son públicos. En la subclase los atributos son privados y los métodos públicos.

1.3.- SUPER

Es una palabra reservada que nos permite referirnos a la superclase.

Super tiene dos funciones generales:

a) Llamar al constructor de la superclase.

b) Acceder a un miembro de la superclase que ha sido ocultado por uno de la subclase.

Super para llamar a constructores

De forma general los constructores inicializan las propiedades o atributos de las clases, es por ello que al no heredarse los constructores tanto la superclase como las subclases deben tener los suyos propios. En las subclases los constructores además de establecer el estado inicial de la propia subclase deben establecer primero el estado inicial de la superclase (cuando se crea un objeto de una subclase primero se debe ejecutar el constructor de

la superclase y luego el de la subclase). Para ello los constructores de las subclases tienen la posibilidad de invocar a los constructores de sus superclases mediante la palabra reservada **super**. Por tanto el responsable de construir un objeto de la subclase es por una parte el constructor de la superclase que inicializa la porción de la superclase del objeto y el constructor de la subclase que inicializa la parte de la subclase.

Sintaxis:

super ([parámetros])

Los parámetros pueden ser: cero, uno o más de uno.

La llamada super debe ser obligatoriamente la primera sentencia del constructor.

Ejm:

```
package pCaja;
    public class Caja {
        protected double ancho;
        protected double alto;
        protected double largo;
        public  Caja ( double w,double h, double d) {
            ancho=w;
            alto=h;
            largo=d;
        }
        public Caja( ) {
            this (-1,-1,-1);
        }
//Constructor que se utiliza para crear un cubo
        public Caja (double lon) {
            this (lon, lon, lon);
        }
//Cálculo y devolución del volumen
        public double volumen() {
            return ancho*alto*largo;
        }
}
```

```
package pCaja;
public class Cubo extends Caja {
    private double peso;
    //Constructor para Cubo
    public Cubo( double medida, double m ) {
        ancho=medida;
        alto=medida;
        largo=medida;
        peso=m;
    }
```

En el constructor inicializamos explícitamente los atributos de la clase padre. Aunque podemos hacerlo por estar declarados como protected no es eficiente, puesto que estamos duplicando el código de la superclase en la subclase. Además si la superclase tuviera atributos privados la subclase no podría acceder directamente a los mismos para **inicializarlos.**

La solución consiste en utilizar la palabra reservada super, siempre que una subclase necesite referirse a su superclase.

```
    public Cubo( double medida, double m) {
        super(medida);
        peso =m;
    }
}
package pCaja;
public class MainCaja {
    public MainCaja(){
        double vol;
        Cubo miCaja1=new Cubo(10,2.3);
        Cubo miCaja2=new Cubo(2,4.6);
        vol=miCaja1.volumen();
        System.out.println("El volumen de miCaja1 es "+ vol);
        vol=miCaja2.volumen();
        System.out.println("El volumen de miCaja2 es "+ vol);
    }
```

```
    public static void main(String[ ] args) {
        MainCaja interfaz =new MainCaja();
    }
}
```

Salida por pantalla:

El volumen de miCaja1 es 1000.0

El volumen de miCaja2 es 8.0

Cuando en un constructor de una subclase no se utiliza la sentencia super el compilador inserta automáticamente super() como primera instrucción.

Si en la superclase hemos definido algún constructor con parámetros y no hemos definido el constructor sin parámetros, el compilador no encontrará este constructor y por tanto nos dará un error de compilación.

Si en la superclase no hemos definido ningún constructor no existirán problemas, puesto que el constructor por defecto lo crea automáticamente el compilador cuando nosotros no definimos otro.

En cualquier caso, se recomienda poner explícitamente la llamada super([parámetros]).

Super para acceder a un miembro de la superclase

Sintaxis:

super.miembro

Donde miembro puede ser un método o un atributo.

Si en una subclase se definen propiedades o atributos con el mismo nombre que en la superclase, cuando referenciamos en la subclase directamente el nombre de estos atributos nos estaremos refiriendo al de la subclase (el de la superclase quedará oculto). En este sentido super actúa de una forma parecida a this, puesto que cuando la escribimos delante de un atributo nos estamos refiriendo al atributo de la superclase.

Ejm:

```
package ejemplosuper;
public class A {
    protected int i;
}
```

```
package ejemplosuper;
public class B extends A {
    private int i;    // Esta variable i oculta la variable i de A
    public B (int a, int b) {
        super.i=a;   // i de A
        i=b;  // i de B
    }
   public void mostrar() {
        System.out.println("En la superclase i: "+ super.i);
        System.out.println("En la sublase i: "+ i);
    }
}
```

```
package ejemplosuper;
public class Main {
   public Main(){
        B subOb= new B(1,2);
        subOb.mostrar();
   }
    public static void main(String[ ] args) {
        Main interfaz=new Main();
    }
}
```

La salida es la siguiente:

```
En la superclase i: 1
En la subclase i: 2
```

Los métodos tienen un mecanismo similar al de ocultación de atributos, se denomina "redefinición o sobrescritura ". Nos permite volver a definir el comportamiento de los métodos de la superclase en la subclase.

Para poder redefinir un método en la subclase debe tener la misma firma o signatura que en la superclase, es decir, debe tener el mismo nombre y parámetros (número, orden y tipo). El tipo de retorno debe ser el mismo o compatible con el tipo de retorno del original, y su modo de acceso debe ser el mismo o menos restrictivo. Es común que al sobrescribir en la subclase métodos de la clase padre, sea necesario invocar métodos y/o atributos originales de la clase padre, mediante la palabra super.

Redefinir un método en una subclase significa cambiar la implementación del método de la superclase para que se adapte a las necesidades de la subclase.

Una subclase puede redefinir un método de la clase padre por dos motivos:

a) **Reemplazo:** se sustituye completamente la implementación del método heredado manteniendo la semántica.

b) **Refinamiento:** se añade nueva funcionalidad al comportamiento heredado. En el refinamiento resulta útil invocar a la versión heredada del método.

Sin herencia no hay redefinición.

Los métodos static no pueden ser redefinidos en las subclases.

Ejm:

```
package ejemplosuper;
public class Empleado {
    protected String nombre;
    protected int edad;
    protected double sueldo=1000;
    public double obtenerSueldo() {
        return sueldo;
    }
```

```
package ejemplosuper;
public class Jefatura extends Empleado {
    private double incentivo;
    public Jefatura() {
        super();
        incentivo=100;
    }

    public double obtenerSueldo() {
        return super.getSueldo()+incentivo;
    }
}
```

En este caso tenemos un método llamado obtenerSueldo() en la superclase que no nos sirve para la subclase, ya que no tiene en cuenta el incentivo del jefe. Es la razón por la cual se ha redefinido. El método obtenerSueldo() redefinido reemplaza en la subclase el método obtenerSueldo() que hereda.

Si obtenerSueldo() no tuviera delante super nos estaríamos refiriendo al método obtenerSueldo() de la subclase. Sin embargo se quería hacer referencia al método de la superclase ya que se quería coger el sueldo del empleado.

Así pues, esta forma de super tiene una mayor aplicación en situaciones en las que los nombres de miembros de una subclase ocultan a los de la superclase. Se utiliza por tanto para delimitar que versión del método vamos a usar.

No debemos confundir redefinición o sobrescritura con sobrecarga. Existe sobrecarga cuando disponemos en una misma clase de varios métodos con el mismo nombre, pero diferenciables por el número, y/o orden, y/o tipo, de los parámetros que reciben. La redefinición o sobrescritura existe si hay herencia. En ella la cabecera de un método implementado en la clase padre y en la clase hija es la misma, cambia lo que hay entre las llaves, es decir, el cuerpo del método.

1.4.- CREACIÓN DE UNA JERARQUÍA MULTINIVEL

Una subclase puede ser una superclase para otra subclase, es decir, una clase puede ser superclase y subclase al mismo tiempo, lo que significa que las clases van tomando todas las características de sus clases ascendientes (no solo de su superclase o clase padre inmediata) a lo largo de todas la rama del árbol de la jerarquía de clases en la que se encuentre, aunque una subclase no puede acceder directamente a los miembros de los "abuelos". Es posible construir en Java jerarquías que contengan tantos niveles de herencia como se quiera, aunque solo se puede especificar una superclase para cada subclase creada (sin embargo una superclase puede tener cualquier número de subclases).

Por ejemplo, podemos tener una clase Vehículo. Si vamos a trabajar con vehículos que se desplacen por tierra, agua y aire, podríamos tener una clase llamada VehículoTerrestre que herede las características de Vehículo pero que incorpore otros atributos como: número de ruedas y altura de los bajos. Podemos tener también una clase que herede de VehículoTerrestre como la clase Coche, que incorpore nuevos atributos y métodos.

Proceso de inicialización (herencia de tres niveles: abuelo padre e hijo)

1. Se reserva espacio para los atributos del hijo (incluidos los heredados del padre y abuelo).

2. Se inicializan estos atributos a los valores por defecto (null, false,0....)

3.

3.1) Se invoca al constructor del hijo

a) Se invoca la constructor del padre

a.1) Se invoca al constructor del abuelo

a.2) Se ejecuta el constructor del abuelo (atributos de padre e hijo siguen inicializados a valores por defecto).

b) Se termina de ejecutar el constructor del padre (atributos del hijo siguen inicializados a los valores por defecto)

3.2) Se termina de ejecutar el constructor del hijo.

Ejm:

```
package pEjemplo3;
public class Vehículo {
   protected boolean aireAcondicionado;
    public Vehículo() {
        aireAcondicionado=true;
        System.out.println("Dentro del constructor de Vehículo. Aire: "+
        aireAcondicionado);
    }
}
```

```
package pEjemplo3;
public class VehículoTerrestre extends Vehículo{
    private int numRuedas;
     public VehículoTerrestre() {
        super();
         numRuedas=4;
          System.out.println("Dentro del constructor deVehículoTerrestre.
         Ruedas:"+ numRuedas);
     }
}
```

```
package pEjemplo3;
public class Coche extends VehículoTerrestre {
     private double peso;
     public Coche() {
         super();
         peso=7;
         System.out.println("Dentro del constructor de Coche. Peso:"+ peso);
     }
}
```

```
package pEjemplo3;
public class Main {
   public Main(){
        Coche obj=new Coche();
   }
    public static void main(String[ ] args) {
          Main interfaz=new Main();
    }
}
```

Salida:

Dentro del constructor de Vehiculo. Aire: true

Dentro del constructor de Vehículo Terrestre. Ruedas: 4

Dentro del constructor de Coche. Peso: 7.0

1.5.- UTILIZACIÓN FINAL CON LA HERENCIA

La palabra clave final tiene tres utilizaciones. En primer lugar sirve para crear el equivalente de una constante. Las otras dos se aplican a la herencia.

Utilización de final para impedir la herencia

Si queremos evitar que una clase sea heredada es necesario que el nombre de la clase vaya precedido de la palabra clave **fina**l, es decir, si utilizamos el modificador final al definir una clase evitamos que se puedan construir clases derivadas de ella.

Ejm:

```
public final class NombreClase { }

public class subclase extends NombreClase { }
```

Al declarar una clase con el modificador **final** se declara también implícitamente a todos sus métodos con **final**.

La utilización de este modificador asociado a las clases no es muy habitual, pues restringe la posibilidad de reutilizar las clases usando el mecanismo de herencia.

Utilización de final para impedir la sobrescritura

La sobrescritura o redefinición de métodos es una de las características más importantes de Java, pero pueden presentarse ocasiones en las que haya que evitarla. Para imposibilitar que un método sea sobrescrito en una subclase hay que especificar el modificador final en su declaración en la superclase. Los métodos que se declaran con final no pueden ser sobrescritos.

```
package pEjemploFinal;
public class A {
        public final void meth() { //Este método no se puede redefinir
           System.out.println("Este es un metodo final");
        }
}
```

```
package pEjemploFinal;
public class B extends A {
         public void meth() {   //Error no está permitida la sobresecritura
                  System.out.println("No es correcto");
        }
}
```

1.6.- LA CLASE OBJECT

En java todas las clases heredan de otra, excepto la clase Object.

- Si una clase hereda de otra específica (definida por nosotros), se indicará con extends en la definición de la clase.
- Si no se especifica nada en la definición de una clase, el compilador hace que herede de Object (raíz de la jerarquía de clases en java y la única que no hereda de nadie). Que una clase herede de Object implica que va a contar con sus atributos y métodos. Algunos de ellos son: clone(), toString(), equals(), etc. Como cualquier otro método de una superclase se pueden redefinir en sus subclases.

Es como si a una clase cualquiera, public class MiClase {----}, le añadiera automáticamente, public class MiClase extends Object {----}.

Aunque no se tenga definido el método toString() podemos invocarlo.

Ejm:

```
package pObject;
public class MiClase {
   private int numero;
 }
package pObject;
public class Main {
   public Main(){
      MiClase obj=new MiClase();
      System.out.println (obj.toString());
   }
   public static void main(String[ ] args) {
       Main interfaz=new Main();
   }
}
```

La salida por pantalla podría ser: pObject.MiClase@123f3567. Esto es debido a que al no haber redefinido en nuestra clase el método toString() se invoca el heredado de la clase Object, el cual nos devuelve el nombre del paquete + nombre de la clase + @ + un número entero que identifica al objeto. Nosotros redefinimos el método en nuestras clases para adaptarlo y que nos imprima los valores de los atributos del objeto.

2. POLIFORMISMO (muchas formas)

2.1.- DEFINICIÓN

El término se refiere a la capacidad que tiene un objeto (referencia al objeto) para poder comportarse de diferentes formas, es decir, las subclases de una clase pueden definir su propio comportamiento. Así por ejemplo un método puede tener múltiples implementaciones que se seleccionan en base al tipo de objeto que le referencia. Del mismo modo permite que clases de distintos tipos puedan ser referenciadas por una misma variable.

2.2.- PROPIEDADES DEL POLIMORFISMO

1. Una variable definida como tipo de la clase padre (enlace estático) puede hacer referencia a un objeto de cualquiera de sus subclases (enlace dinámico), siempre y cuando haya una relación de herencia entre las clases. A este tipo de polimorfismo se le denomina polimorfismo de asignación.

   Ejm:

   ```
   public class Mamifero {----}

   public class Gato extends Mamifero {-----}

   public class Perro extends Mamifero {-----}

   Mamifero cat = new Gato();

   Mamifero dog = new Perro();
   ```

   De esta forma permitimos que diferentes clases puedan ser referenciadas por variables del mismo tipo, lo cual es importante para escribir código genérico (un único código o método con diferentes interpretaciones en tiempo de ejecución).

2. En base al polimorfismo de asignación se pueden crear métodos capaces de trabajar con varios objetos diferentes en tiempo de ejecución. El tipo de objeto con el que se trabaja se selecciona en base a la clase sobre la que se ha creado el objeto, no al de la referencia. Este tipo de polimorfismo se denomina polimorfismo puro.

Por tanto para que exista polimorfismo se debe cumplir lo siguiente:

a) Los objetos deben ser manipulados usando referencias a la clase base o padre.

b) Los métodos deben estar declarados (métodos abstractos) o implementados en la clase base o clase padre.

c) Los métodos deben estar redefinidos en las clases derivadas o subclases, para que en tiempo de ejecución se determine que versión del método debe ejecutarse.

No puede haber polimorfismo sin redefinición, pero sí a la inversa.

Ejm:

```
package  polimorfismo;
public class Mamifero {
        protected String especie;

        public Mamifero (String especie) {
                this.especie=especie;
        }
        public String getEspecie() {
                return  especie;
        }
        public void mover () {
                System.out.println("Los animales se mueven");
        }
}
package polimorfismo;
public class Gato extends Mamifero {
        private int npatas;

        public Gato (String especie, int npatas) {
                super(especie);
                this.npatas=npatas;
        }
        public int getNpatas(){
                return npatas;
        }
        @Override
        void mover() {
                System.out.println ("El Gato se mueve sigilosamente");
        }
}
```

```
package polimorfismo;
public class Perro extends Mamifero {
        private String raza;
        public Perro (String especie, String raza) {
                super(especie);
                this.raza=raza;
        }

        public String getRaza() {
                return raza;
        }

        @Override
        public void mover() {

                System.out.println ("El perro de raza "+raza+" se mueve con
                rapidez");
        }
}

package polimorfismo;
public class Polimorfismo{
        public void  decirMovimiento (Mamifero animal) {
                animal.mover();
        }
}

package polimorfismo;
public class Principal {
        public Principal (){
                Mamifero cat=new Gato("gato",4);
                Mamifero dog=new Perro("perro","labrador");
                Polimorfismo enlace=new Polimorfismo();
                enlace.decirMovimiento(cat);
                enlace.decirMovimiento(dog);
        }
          public satic void main(String[ ] args) {
                Principal interfaz=new Principal();
        }
}
```

2.3.- VENTAJAS

- Facilita la definición de clases genéricas donde se definen todos los métodos genéricos que se precisen (métodos con diferentes interpretaciones en tiempo de ejecución), los cuales manipularán un tipo de datos real que se especificará a través de un parámetro. De esta forma podemos crear nuevas subclases sin necesidad de modificar las existentes.
- Da flexibilidad a los programas.
- Permite reutilizar código mejorando su mantenimiento.

3. CASTING

Si no tenemos un método sobrescrito (no figura en la superclase y en la subclase), cuando intentamos hacer una llamada a un método de la subclase utilizando una referencia de la superclase, se va a producir un error, puesto que desde una superclase no se tiene visibilidad a métodos de la subclase (al contrario no hay problema pues las subclases los heredan).

El tipo de la variable de referencia determinará los métodos que podrán ser llamadas sobre el objeto.

Ejm: Supongamos que en la clase Gato tenemos el siguiente método implementado:

```
public void verCaracteristicas(){
        System.out.println("Los gatos son flexibles y elásticos");
}
```

Con el main siguiente:

```
Mamifero animales =new Mamifero("animales");
Mamifero cat =new Gato("gato",4);
cat.varCaracteristicas(); /*Esta llamada daría un error (se producirá la excepción: ClassCastException), pues se intenta acceder a un método de la subclase (no sobreescrito), desde una referencia del tipo de la superclase (Mamifero)*/
```

Para acceder al método verCaracteristicas(), debemos hacer una conversión hacia abajo (hacia las subclases). Para ello hay que realizar un casting explícito y así transformar una variable de un tipo a otro.

Hay dos posibilidades:

1. Definiendo una variable del tipo de la subclase.

Ejm:

Gato instanciaGato = (Gato)cat;

instanciaGato.verCaracteristicas();

2. Utilizar el mecanismo de casting sin definir una variable de la subclase.

Ejm: ((Gato) cat).verCaracteristicas();

4. OPERADOR instanceof

Este operador permite conocer la clase a la que pertenece un objeto (un objeto de la clase hija también lo es de la clase padre y de sus antecesores).

Devuelve un valor booleano. Retorna true si el objeto pertenece a la clase que se está comprobando o a alguna sus subclases, y false en caso contrario.

Con este operador podemos evitar errores.

Sintaxis: puntero que apunta al objeto **instanceof** nombre de la clase

- if (alum1 instanceof Alumno) {-----} /* Devuelve true si alumn1 es una instancia de la clase que se está comprobando o de alguna de sus subclases*/
- if (! (obj instanceof Coordenada)) {---} /* Si obj no apunta a un objeto de la clase Coordenada o de alguna de sus subclases, retorna directamente false */

Ejemplo:

```
public class Vehiculos {
        protected String modelo;
        protected String color;
}

public class Coches extends Vehiculos {
        private int puertas;
}
```

```
public class Principal {
    public static void main (String [ ] args) {
        Vehiculos coche1 = new Coches();

        if (coche1 instanceof Coches) /* Devuelve true, puesto que coche1 apunta a un objeto de la clase Coches*/

        Coches coche2 = new Coches();

        if (coche2 instanceof Vehiculos) /* Devuelve true, puesto que todo objeto de una clase también es una instancia de su padre (un coche es un  vehículo siempre*/

        Vehiculos coche3 = new Vehiculos();

        if (coche3 instanceof Coches) /* Devuelve false, puesto que coche3 no es una instancia de Coches*/

  }
}
```

Errores de compilación

- No se puede usar **instanceof** para tratar de comprobar si un objeto pertenece a una clase diferente a la suya o a sus antecesoras. Solo es posible utilizar el operador con referencias que apuntan a su clase o a sus antecesoras.

  Ejm:

```
public class Animal {-------}
public class Gato  extends Animal {----}
public class Perro extends Animal {-----}
public class Principal {
    public static void main (String [ ] args) {
            Animal toby = new Perro();
            if (toby instanceof Gato)
                    System.out.println("Esto no se va a escribir");
    }
}
```

- Si se aplica **instanceof** a una referencia que se encuentre apuntando a null, el resultado siempre será false sin importar la clase con la que se valida.
- Solo se puede usar **instanceof** asociado a un condicional. No se puede usar en un System.out.println.

  Ejm: if (interino instanceof Profesor)

5. MÉTODOS Y CLASES ABSTRACTAS

5.1.-MÉTODOS ABSTRACTOS

A veces para hacer uso de la redefinición o sobrescritura nos vemos obligados a definir métodos en la superclase sin cuerpo. En estos casos se puede utilizar la versión abstracta del método.

Un método abstracto es un método que se declara pero no se define su contenido. Deben sobrescribirlo todas las clases que hereden de la clase que lo contiene. La sintaxis es la misma que la de un método no abstracto pero terminado con punto y coma, y sin poner las llaves de comienzo y final del método (no tiene cuerpo). Debe llevar en su cabecera la palabra reservada abstract (indica que se debe hacer, pero no como hacerlo).

Ejm: public abstract double perimetro();

Los métodos abstractos nunca deben ser definidos con el modificador final, puesto que impide que puedan ser redefinidos o sobrescritos en las subclases. Del mismo modo el modificador abstract no se puede usar en métodos de instancia privados ni en métodos de clase (static), pues ambos tipos de métodos no pueden ser sobrescritos en las subclases.

Si una clase tiene al menos un método abstracto, forzosamente tiene que ser abstracta.

5.2.-CLASES ABSTRACTAS

Hay ocasiones en las que se quiere definir de forma generalizada una superclase declarando su estructura sin implementar cada método, dejando que cada subclase complete los detalles necesarios. Su único objetivo es que sirva de superclase a las clases “reales” para mantener nuestra aplicación con una estructura más organizada y fácil de entender, indicando el modelo que deben seguir las subclases, de tal forma que definen su comportamiento.

Una clase abstracta es similar a una clase normal, ya que posee nombre, atributos, métodos, y aunque puede tener solo métodos generales con un comportamiento común para las subclases, lo normal es que al menos uno de ellos sea abstracto. De este tipo de clases no se tiene intención de crear objetos.

Sintaxis: public abstract class Persona {
    //Al menos un método abstracto
}

Utilizar clases abstractas tienen lógica solo con herencia, cuando existan varias clases con características comunes pero con distintos comportamientos (la herencia facilita a la máquina virtual de java encontrar los métodos definidos en las subclases de manera más rápida para ser ejecutados, al estar definidos en la clase padre, lo que conlleva la disminución del tiempo de ejecución). La finalidad por tanto de este tipo de clases es definir las cosas que se tienen que hacer, correspondiendo a cada subclase como hacerlo (la clase padre servirá de plantilla para las clases hijas). Así, si tenemos una clase que hereda de otra abstracta, estamos obligados a implementar todos los métodos abstractos de la clase padre, de igual manera, si tenemos otra clase que también hereda del mismo padre, se implementará el mismo método pero con un comportamiento distinto.

Veamos cómo se implementa en Java.

```
public abstract class SuperClase {

    //Método con implementación
    public void metodoConcreto() {
    .....
    }

    //Método abstracto sin implementación
    public abstract void metodoAbstracto();
}

public class subClase extends SuperClase {
    //Método con implementación
    public void metodoAbstracto() {
    .....
    }
}
```

5.3.-CARACTERÍSTICAS DE UNA CLASE ABSTRACTA

- Una clase abstracta **No** puede ser instanciada (no se pueden crear objetos directamente con new), solo puede ser heredada, aunque es posible declarar variables de una clase abstracta que puedan apuntar a objetos de cualquiera de sus subclases (se pueden crear arrays de objetos de una clase abstracta, puesto que lo que se crea al instanciar un array son referencias a objetos, no los objetos propiamente dichos).

  Al no poder instanciar una clase de este tipo nos aseguramos que sus atributos o propiedades solo estarán disponibles para sus clases hijas, por tanto se facilita el diseño y la programación orientada a objetos.

  Ejm:

  ClaseAbstracta variable;
  ~~variable =new ClaseAbstracta();~~

- Una subclase que hereda de una clase abstracta puede ser a su vez una clase abstracta. El atributo abstract se hereda hasta el momento en que se logra una implementación completa de todos los métodos abstractos de la superclase.

  Ejm:

```
public abstract class Abstracta {
    //Tiene al menos un método abstracto
}
 public abstract class Derivada extends Abstracta {
    /*Podemos implementar parte de los métodos abstractos, todos o ninguno. No estamos obligados a implementar los métodos abstractos*/
}
public class NoAbstracta extends Derivada {
    /*Tenemos que implementar todos los métodos abstractos que tenga Abstracta y Derivada*/
}

NoAbstracta instancia =new NoAbstracta();
```

- Si al menos un método de una clase es abstracto, esto obliga a que la clase completa sea definida como abstracta, sin embargo la clase puede tener el resto de métodos no abstractos.

- Todas las clases que heredan de una clase abstracta están obligadas a sobrescribir los métodos abstractos, aunque cuando se sobrescriben por la clase heredada ya no lo van a ser. Sin embargo no están obligadas a sobrescribir los métodos que no sean abstractos.

- Una clase abstracta puede heredar de otra que no lo es.

- Aunque no se pueda instanciar un objeto de una clase abstracta, cuando se crea de cualquiera de las subclases se invoca al constructor de la clase padre, por lo que una clase abstracta siempre tiene al menos un constructor (si no está definido por el propio usuario es el compilador el que lo proporciona por defecto), aunque no puede ser abstracto. El constructor se utiliza para inicializar los atributos de la clase abstracta.

Ejm:

```
public abstract class Instrumento {
    private String tipo;
    public Instrumento(String tipo) {
        this.tipo=tipo;
    }
    public abstract void tocar();
    public String toString() {
        return"Tipo: "+tipo;
   }
}

public class Guitarra extends Instrumento {
    public Guitarra(String tipo) {
        super(tipo);
     }
    public void tocar() {
        System.out.println("Se toca con los dedos");
    }
}
```

```
public class Violin extends Instrumento {
    public Violin(String tipo) {
        super(tipo);
     }
    public void tocar() {
        System.out.println("Se toca con el arco");
    }
}

public class InstrumentoGeneral {
      public void visualizar(Instrumento general) {
            general.tocar();
      }
}

public class Principal {
      public Principal() {
          InstrumentoGeneral obj=new InstrumentoGeneral();
          Instrumento miGuitarra= new Guitarra("Guitarra");
          Instrumento miViolin= new Violin("Violín");
          System.out.println(miGuitarra.toString());
          obj.visualizar(miGuitarra);
          System.out.println(miViolin.toString());
          obj.visualizar(miViolin);
      }
      public static void main(String arg [ ]){
            Principal interfaz=new Principal();
      }
}
```

En el ejemplo vemos una clase abstracta Instrumento la cual posee una propiedad: tipo, un método concreto: toString() y un método abstracto: tocar(). Las subclases Guitarra y Violín, heredarán la propiedad y los métodos de la clase Padre.

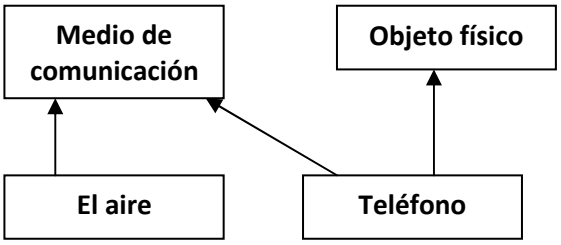
Tanto la guitarra como el violín se pueden tocar, aunque cada uno de estos instrumentos se toca de manera diferente. Es por ello que definimos el método tocar() en la clase padre como abstracto, y al heredarlo las subclases están obligadas a implementarlo y darle la funcionalidad que le corresponda.

En la clase Principal se tiene la lógica para ejecutar la aplicación. Se crearon los objetos de tipo Instrumento por medio de sus clases hijas, pero en ningún momento se creó un instrumento como instancia directa de la clase abstracta.

6. INTERFACES

6.1.- USOS DE LAS INTERFACES

- Proporcionan un mecanismo de herencia múltiple que no puede ser utilizado empleando solo clases. Este mecanismo proporcionado por java es muy limitado respecto a las posibilidades que ofrece la herencia múltiple de la POO como tal; sin embargo, permite afrontar diferentes situaciones de diseño y programación.



- Se utilizan cuando objetos de distintas clases, y que no pertenecen a la misma jerarquía, deben compartir métodos.

- Cuando objetos que pertenecen a una misma jerarquía, no necesitan compartir un método, para evitar "contaminar" la definición de la superclase.

Al igual que ocurre con las clases, hay interface predefinidas que vienen en la API de java (se distinguen de las clases porque están escritas en cursiva), para que nosotros las usemos cuando las necesitemos; y también podemos crear las nuestras propias para incluirlas en clases.

6.2.- CARACTERÍSTICAS DE LAS INTERFACES

- Establecen los comportamientos de las clases que implementan.

- La interface no puede ser ni privada, ni protegida. Si está declarada como pública, el archivo .java que la contiene, tiene el mismo nombre que la interfaz. Al compilar el archivo .java de la interface, se genera un archivo .class.

- Las interfaces son colecciones de constantes y métodos abstractos (no tienen variables). Los métodos de forma general son siempre públicos y abstractos (no necesitamos definirlos como tal), y las constantes tienen por supuesto la palabra reservada final, son públicas y estáticas (no es necesario especificar sus atributos de acceso y modificadores). Una interface de forma general, se puede considerar una clase abstracta pura.

Sintaxis:

```
public interface Nombre {
        //constantes si las hay. Se deben inicializar en la declaración.
        //métodos abstractos
}
```

Ejm:

```
public interface DatosCentroEstudios{
        //constantes. No hace falta especificar: public, static,final.
         byte NUMERO_PISOS=5;
         byte NUMERO_AULAS=25;
        //métodos. No hace falta especificar: public, abstract.
        short numeroAprobados(int numero);
        float notaMedia (float nota);
 }
```

- No se pueden instanciar (no se puede utilizar new).

6.3.- IMPLEMENTACIÓN DE LAS INTERFACES

- Si queremos que una clase herede una interface, se emplea la palabra reservada "implements" en lugar de extends, para indicarlo. En este caso, la clase que implementa la interface, tiene que implementar los métodos de la misma, excepto si es una clase abstracta.

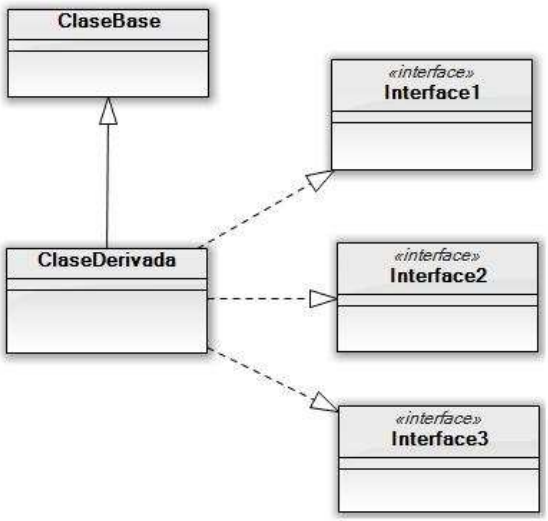
Si una clase implementa una interfaz, todas sus subclases derivadas heredan los métodos implementados de la superclase y las constantes definidas en la interface.

Ejm:

```
public class CCentroEstudios implements DatosCentroEstudios {
        public float notaMedia (float nota) {
        ----
        }
}
```

- Una clase puede implementar varias interfaces, y al mismo tiempo heredar de una clase (solo se puede tener una superclase). En este caso, los nombres de las interfaces se escriben a continuación de implements, separadas por comas. La clase debe implementar los métodos de todas las interfaces.

Ejm:




```
public class ClaseDerivada extends ClaseBase implements Interface1, Interface2, Interface3, .....
```

Cuando hay una implementación múltiple, es posible que dos interfaces tengan constantes o métodos con el mismo nombre. La clase que implementa las interfaces recibirá por tanto, métodos o constantes con el mismo nombre. A esto se le llama **colisión.**

Java establece una serie de reglas, para solucionar las colisiones:

1. Para las colisiones de nombres de constantes, se obliga a especificar el nombre de la interfaz base a la cual pertenecen , cuando vayamos a utilizarlos: (NombreInterfaz.constante).

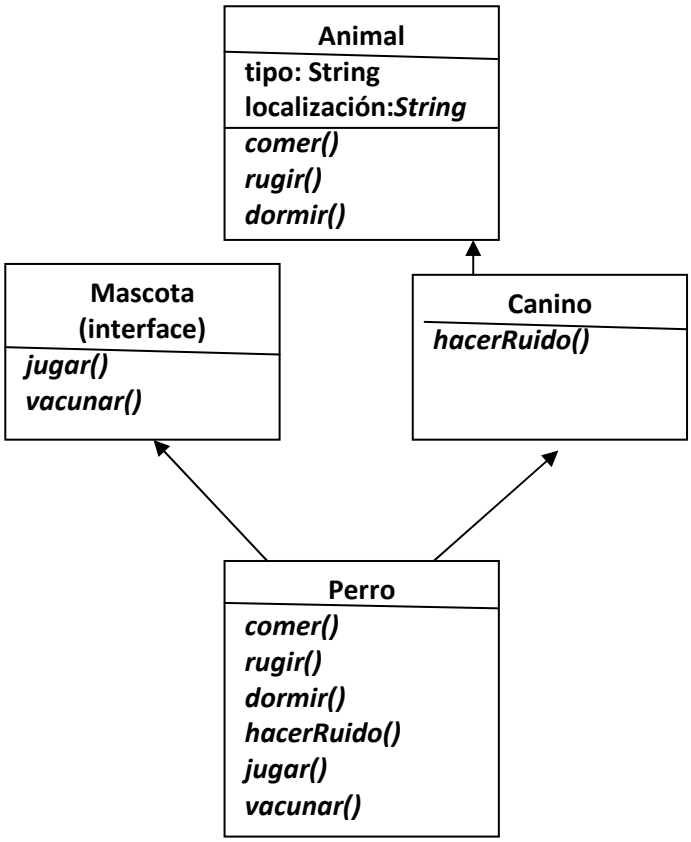
2. Para las colisiones de los nombres de métodos:

a) Si tiene el mismo nombre y diferentes parámetros, se produce sobrecarga de métodos, permitiéndose que existan varias maneras de llamar al método.

b) Si sólo cambia el valor devuelto, da un error de compilación, indicando que no se pueden implementar los dos.

c) Si coinciden en sus declaraciones, se debe eliminar uno de los dos.

Ejm:



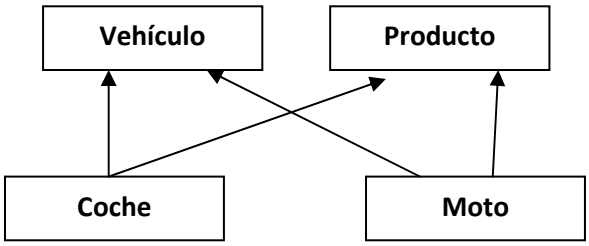

```
public interface Mascota { //todos los interfaces son públicos
        public abstract void jugar(); //son públicos automáticamente
        public abstract void vacunar();
}
```

```
public class Perro extends Canino implements Mascota {
//Hereda los métodos comer(), rugir(), dormir() y hacerRuido()
    public void comer() {...}
    public void rugir (){…}
    public void dormir(){....}
    public void hacerRuido(){...}
//Debe implementar los métodos jugar() y vacunar()
    public void jugar(){…}
    public void vacunar(){…}
}
```

Ejm:

En una tienda de vehículos de 2ª mano, existen coches y motos. Además desde un punto de vista contable, son productos de la tienda (se les puede asignar un precio y se pueden vender).

La representación gráfica de las clases existentes, y las relaciones entre ellas sería:



Su implementación:

```
public interface Producto() {
    public void definirPrecio(double precio);
    public void vender();
}

public class Coche extends Vehiculo implements Producto {
    private double precio;
```

```
        public void definirPrecio(double precio) {
                this.precio = precio;
        }
        public void vender() {
                System.out.println("Realizada operación VENTA de coche");
        }
}

 // Para la clase moto sería la misma implementación
```

6.4.- HERENCIA ENTRE INTERFACES

Al igual que las clases podemos definir una interface basada en otra, utilizando la palabra reservada "extends"; es decir, se puede establecer una jerarquía de herencia entre interfaces.

Cuando la relación es interface - interface o clase - clase se pone extends.

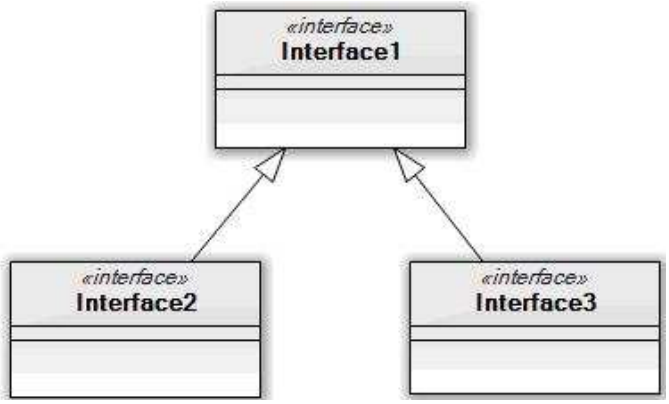
Cuando la relación es clase - interface, se pone implements.

Ejm:

```
public interface DatosCentroEstudios {
        byte NUMERO_PISOS=5;
        byte NUMERO_AULAS=25;
        short numeroAprobados(int numero);
        float notaMedia (float nota);
 }

public interface CentroEstudios extends DatosCentroEstudios {
        float numeroSuspensos(int numero);
        float varianza (float notas);
}
```

Ejm:




```
public interface Interface2 extends Interface1{
…….
}

public interface Interface3 extends Interface1{
…….
}
```

En este ejemplo, Interface2 e Interface3, heredan los métodos y constantes de Interface1, y además pueden añadir los suyos.

En estos casos, podemos hablar de superinterfaces y subinterfaces.

A diferencia de las clases, una interfaz puede extenderse simultáneamente a varias superinterfaces, lo que supone una aproximación a la posibilidad de realizar herencia múltiple.

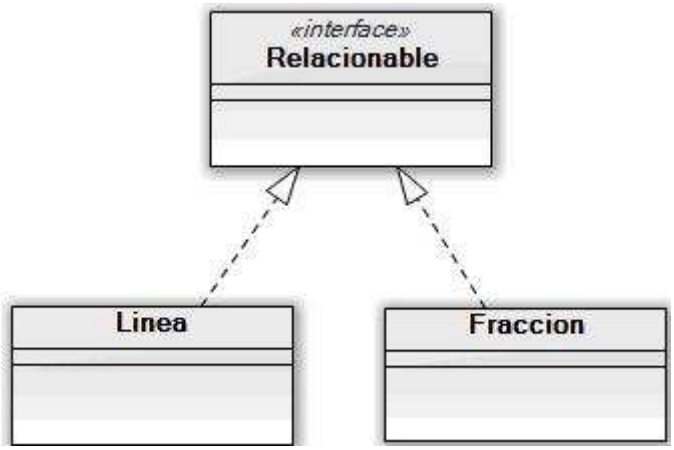
Ejm:

```
public interface CentroEstudios extends DatosCentroEstudios, CalculosCentro {
------
}
```

6.5.- INTERFACE Y POLIMORFISMO

- Se puede utilizar el polimorfismo o el principio de sustitución con la interfaz.

Ejm:



En este caso dos clases no relacionadas, *Linea* y *Fraccion*, por implementar la misma interfaz *Relacionable,* podemos manejarlas a través de referencias a la interfaz y aplicar polimorfismo.

Podemos escribir las instrucciones:

Relacionable r1 = new Linea();

Relacionable r2 = new Fraccion();

6.6.- MÉTODOS DEFAULT EN INTERFACES

Antes de Java 8, podíamos decir que toda clase que implementa a una interfaz, debe implementar los distintos métodos que contenga la interfaz. En cambio en Java 8, toda clase que implemente una interfaz, debe implementar los distintos métodos que contenga, salvo aquellos que estén definidos como métodos por defecto si no se quiere.

Estos métodos se caracterizan porque son **métodos que están implementados en la propia interfaz , y pueden ser utilizados directamente en la clase** si nos interesa su comportamiento por defecto. Para definirlos se utiliza la palabra default, seguida de la firma del método:

Ejm:

```
public interface ICalculadora {
    public void sumar (int x, int y);
    public void restar (int x, int y);
    public default int multiplicar (int x, int y) {
    return x*y;
}
```

```
public class Calculadora implements ICalculadora {
@Override
        public void sumar (int x, int y) {
                System.out.println ("La suma es: " + (x+y));
        }
@Override
        public void restar (int x, int y) {
                System.out.println ("La resta es: " + (x-y));
        }
}
public class Java8 {
        public static void main (String[] args) {
                Calculadora c = new Calculadora();
                c.sumar(4,3);
                System.out.println ("La multiplicación es "+c.multiplicar(2,3));
        }
}
```

La salida por pantalla es:

La suma es 7

La multiplicación es 6

Ahora bien, cabe destacar que los métodos default pueden ser redefinidos en las clases. Si al método multiplicar en Calculadora queremos darle un comportamiento diferente, podríamos redefinir el método de la siguiente forma:

```
public class Calculadora implements ICalculadora {
   @Override
        public void sumar (int x, int y) {
                System.out.println ("La suma es: " + (x+y));
        }

   @Override
        public void restar (int x, int y) {
                System.out.println ("La resta es: " + (x-y));
        }
```

```
    @Override
        public int multiplicar (int x, int y) {
                return (x+1)*(y+1);
        }
}
public class Java8 {
public static void main (String[] args) {
       Calculadora c = new Calculadora();
        c.sumar(4,3);
       System.out.println ("La multiplicación es "+c.multiplicar(2,3));
}
```

La salida por pantalla es:
La suma es 7
La multiplicación es 12

Como vemos el resultado de la multiplicación ahora no es el mismo, puesto que al estar el método multiplicar redefinido, cuando le hacemos la llamada, no ejecutamos el que está por defecto en la interface, sino el redefinido en la clase.

Tras ver el ejemplo cabe destacar, que en una interfaz podremos añadir **nuevos comportamientos** por defecto (**métodos default**), que no tendrán por qué ser tratados en todas las clases que implementen dicha interfaz, pero sí **podrán ser utilizados en todas ellas.**

Puede darse el caso de que una clase implemente dos interfaces que tienen definido un método default con la misma firma; en este caso, es necesario redefinir en la clase el método, e indicar a que interfaz llama.

Ejm:

```
public interface Persona  {
    default void decirHola() {
     System.out.println("Hola persona!!!!!");
   }

}
```

```
public interface Hombre  {
   default void decirHola() {
      System.out.println("Hola hombre!!!!!");
    }

}
public class Pedro implements Persona, Hombre {
   public void decirHola() {
     Hombre.super.decirHola();
   }

}
```

Debemos tener también presente que en una interfaz, no se puede implementar un método default que sobrescribe algún método de la clase Object, como: equals, hasCode o toString.

Podríamos plantearnos: ¿cuál es la diferencia entre una clase abstracta y un interfaz que implementa métodos *default*?; la diferencia es mínima. Una clase sólo puede extender una clase abstracta, pero puede implementar múltiples interfaces.

Las reglas de diseño establecen que preferiblemente se usarán antes las interfaces que las clases abstractas. De esta manera no se compromete a los objetos con una jerarquía determinada.