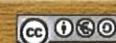


# **Unidad 7: Herencia y Relaciones entre clases**

---

## **Fundamentos de Programación. 1º de ASI**



Esta obra está bajo una licencia de Creative Commons.

Autor: Jorge Sánchez Asenjo (año 2010) <http://www.jorgesanchez.net>  
e-mail:info@jorgesanchez.net

---

Esta obra está bajo una licencia de Reconocimiento-NoComercial-CompartirIgual de Creative Commons

Para ver una copia de esta licencia, visite:

<http://creativecommons.org/licenses/by-nc-sa/2.5/es/legalcode.es>

o envíe una carta a:

Creative Commons, 559 Nathan Abbot



## Reconocimiento-NoComercial-CompartirIgual 2.5 España

### Usted es libre de:



- copiar, distribuir y comunicar públicamente la obra
- hacer obras derivadas

### Bajo las condiciones siguientes:



**Reconocimiento.** Debe reconocer los créditos de la obra de la manera especificada por el autor o el licenciador (pero no de una manera que sugiera que tiene su apoyo o apoyan el uso que hace de su obra).

**No comercial.** No puede utilizar esta obra para fines comerciales.

**Compartir bajo la misma licencia.** Si altera o transforma esta obra, o genera una obra derivada, sólo puede distribuir la obra generada bajo una licencia idéntica a ésta.

- Al reutilizar o distribuir la obra, tiene que dejar bien claro los términos de la licencia de esta obra.
- Alguna de estas condiciones puede no aplicarse si se obtiene el permiso del titular de los derechos de autor
- Apart from the remix rights granted under this license, nothing in this license impairs or restricts the author's moral rights.

Advertencia

Los derechos derivados de usos legítimos u otras limitaciones reconocidas por ley no se ven afectados por lo anterior.  
Esto es un resumen legible por humanos del texto legal (la licencia completa) disponible en los idiomas siguientes:  
Catalán Castellano Euskera Gallego

Para ver una copia completa de la licencia, acudir a la dirección <http://creativecommons.org/licenses/by-nc-sa/2.5/es/legalcode.es>

# (7) herencia y relaciones entre clases

## esquema de la unidad

|                                                                 |    |
|-----------------------------------------------------------------|----|
| (7.1) relaciones entre clases                                   | 6  |
| (7.1.1) asociaciones                                            | 6  |
| (7.1.2) agregación y composición                                | 8  |
| (7.2) herencia                                                  | 9  |
| (7.2.1) introducción                                            | 9  |
| (7.2.2) métodos y propiedades heredables. modificador protected | 11 |
| (7.2.3) anulación de métodos                                    | 12 |
| (7.2.4) constructores                                           | 14 |
| (7.3) casting de clases                                         | 18 |
| (7.3.1) instanceof                                              | 19 |
| (7.4) clases abstractas                                         | 19 |
| (7.5) final                                                     | 21 |
| (7.6) interfaces                                                | 23 |
| (7.6.1) utilizar interfaces                                     | 23 |
| (7.6.2) creación de interfaces                                  | 23 |
| (7.6.3) subinterfaces                                           | 24 |
| (7.6.4) variables de interfaz                                   | 25 |
| (7.6.5) interfaces como funciones de retroinvocación            | 25 |
| (7.7) la clase Object                                           | 26 |
| (7.7.1) comparar objetos. método equals                         | 27 |
| (7.7.2) código hash                                             | 28 |
| (7.7.3) clonar objetos                                          | 28 |
| (7.7.4) método toString                                         | 29 |
| (7.7.5) lista completa de métodos de la clase Object            | 30 |

se puede relacionar cada objeto de la clase que está en la base de la flecha. Puede ser:

- ◆ 0..1. Significa que se relaciona con uno o ningún objeto de la otra clase.
- ◆ 0..\*. Se relaciona con cero, uno o más objetos
- ◆ 1..\* Se relaciona al menos con uno, pero se puede relacionar con más
- ◆ **un número concreto**. Se puede indicar un número concreto (como 3 por ejemplo) para indicar que se relaciona exactamente con ese número de objetos, ni menos, ni más.

La dirección de la flecha determina de dónde a dónde nos referimos. Así esa misma relación al revés:

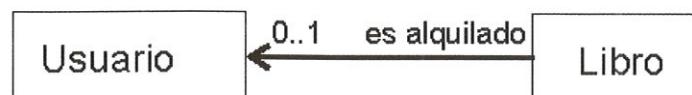


Ilustración 7-2, Asociación UML

Por eso se suele reflejar así de forma completa:

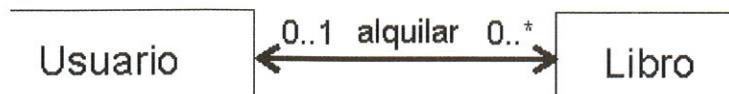


Ilustración 7-3, Asociación UML completa

En muchos casos en las asociaciones no se indica dirección de flecha, se sobreentenderá que la asociación va en las dos direcciones.

Las asociaciones como es lógico implican decisiones en las clases. La clase usuario tendrá una estructura que permita saber qué libros ha alquilado. Y el libro tendrá al menos una propiedad para saber qué usuario le ha alquilado. Además de métodos para relacionar los usuarios y los libros.

Normalmente la solución a la hora de implementar es que las clases incorporen una propiedad que permita relacionar cada objeto con la otra clase. Por ejemplo si la clase usuario Alquila cero o un libro:

```
public class Usuario{  
    ...  
    Libro libro; //representa la relación Usuario->Libro con  
    //cardinalidad 0..1 o 1  
    ...  
}
```

En este caso se refleja que un edificio consta de pisos. De hecho con ello lo que se indica es que un piso sólo puede estar en un edificio. La existencia del piso está ligada a la del edificio. Como se ve la asociación es más fuerte.

### implementación

La implementación en Java de clases con relaciones de agregación y composición es similar. Pero hay un matiz importante. Puesto que en la composición, los objetos que se usan para componer el objeto mayor tienen una existencia ligada al mismo, se deben **crear** dentro del objeto grande. Por ejemplo (composición):

```
public class Edificio {  
    private Piso piso[];  
    public Edificio(...){  
        piso=new Piso[x]; //composición  
        .....  
    }
```

En la composición (como se observa en el ejemplo), la existencia del piso está ligada al edificio por eso los pisos del edificio se deben de crear **dentro** de la clase **Edificio** y así cuando un objeto **Edificio** desaparezca, desaparecerán los pisos del mismo.

Eso no debe ocurrir si la relación es de agregación. Por eso en el caso de los cursos y los módulos, como los módulos no tienen esa dependencia de existencia según el diagrama, seguirán existiendo cuando el módulo desaparezca, por eso se deben declarar fuera de la clase cursos. Es decir, no habrá **new** para crear módulos en el constructor. Sería algo parecido a esto:

```
public class Cursos {  
    private Módulo módulos[];  
    public Edificio(..., Módulo m[]){  
        módulos=m; //agregación  
        .....  
    }
```

Herencia.pdf  
Herencia.tipos  
Herencia.zip.

## (7.2) herencia

### (7.2.1) introducción

La herencia define una relación entre clases en la cual una clase posee características (métodos y propiedades) que proceden de otra. Esto permite estructura de forma muy atractiva los programas y reutilizar código de forma más eficiente. Es decir genera relaciones entre clases del tipo **es como...** (también se las llama **es un** en inglés **is a**).

A la clase que posee las características a heredar se la llama **superclase** y la clase que las hereda se llama **subclase**. Una subclase puede incluso ser superclase en otra relación de herencia.

También se emplea habitualmente los términos **madre** para referirnos a una superclase e **hija** para una subclase (e incluso **abuela** y **nieta**).

Ejemplo de herencia múltiple:

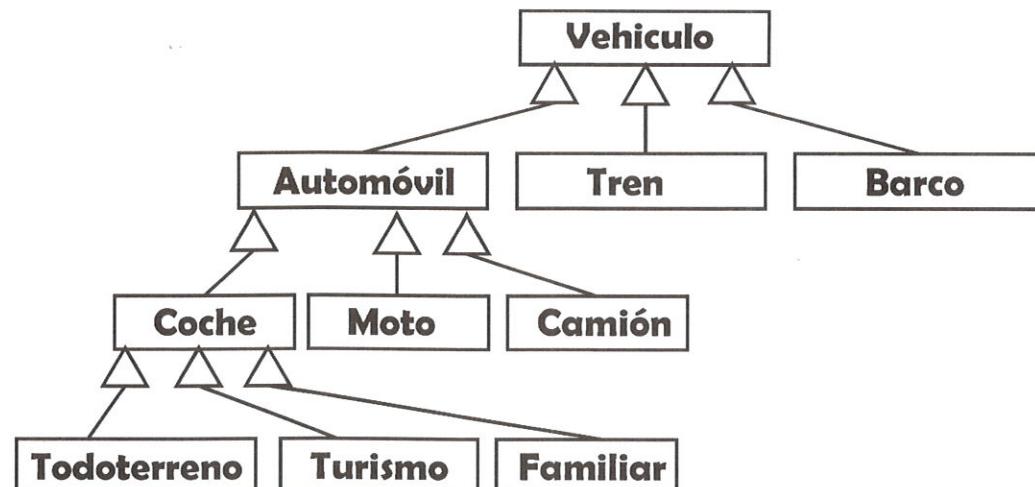


Ilustración 7-7, Diagrama UML de herencia compleja

En ese esquema se puede apreciar como los todoterrenos son coches, que a su vez son automóviles y a su vez son vehículos. Con lo que los todoterrenos adquieren las características heredadas de todas las clases previas.

En Java cada clase sólo puede heredar de una superclase. Eso significa que sólo hay una clase padre. Este tema fue controvertido ya que lenguajes como C++ admiten **herencia múltiple** que significa que una clase puede tener varias clases padre de las que heredaría sus características.

La decisión de no permitir este tipo de herencia se debe a que a la hora de determinar la clase de un objeto habría problemas ya que un objeto se podría considerar perteneciente a varias clases al instanciar.

Las interfaces permiten utilizar cuestiones relativas a la herencia múltiple como se verá más adelante.

### (7.2.2) métodos y propiedades heredables. modificador protected

Además de **public**, **private** y **friendly** en Java se dispone del modificador **protected** (protegido). Este modificador de acceso está específicamente pensado para la herencia. Cuando se utiliza sobre una propiedad o un método indica que dicha propiedad o método serán visibles por las subclases que además heredarán la propiedad o el método. Sin embargo permanecerán invisibles para el resto.

En la práctica se utiliza más que el privado, ya que las conveniencias de colocar en modo privado las propiedades, las impediría ser heredadas por los descendientes de clase; por ello se utiliza más el protected, para asegurar la descendencia y además mantener la ocultación.

Se heredan todos los métodos y propiedades **protected** y **public** (no se heredan los **private** ni los **friendly**). La subclase por su parte puede definir nuevos métodos y propiedades (es lo habitual).

Ejemplo:

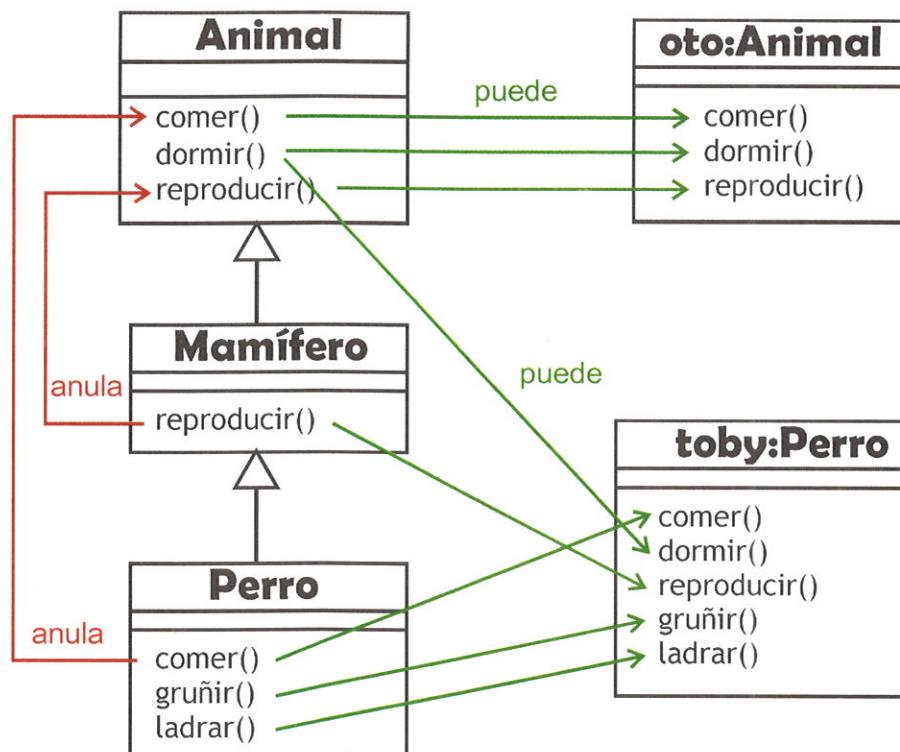


Ilustración 7-8, Anulación de métodos

En el diagrama el objeto **toby** que es un perro, usa el método **comer** de la clase **Perro** (los mamíferos y animales también puede comer pero **toby** usa el método redefinido en la clase **Perro**), sin embargo **toby** duerme como cualquier otro animal ya que dormir no ha sido redefinido, se reproduce como un mamífero ya que la reproducción es distinta en un mamífero según el diagrama. Lógicamente puede gruñir y ladear ya que es un perro.

**oto** es un animal y por ello todo lo que puede usar procede sólo de la clase **Animal**.

A veces se requiere llamar a un método de la superclase. Eso se realiza con la palabra reservada **super**. Anteriormente hemos visto que **this** hace referencia a la clase actual, bien pues **super** hace referencia a la superclase respecto a la clase actual. con lo que es un método imprescindible para poder acceder a métodos anulados por herencia. Ejemplo:

```

public class Vehiculo {
    public double velocidad;
    ...
    public void acelerar(double kmh) {
        velocidad += kmh;
    }
}
  
```

```
    } // si no coloco el constructor por defecto el compilador lo hace automaticamente
```

La clase **A** simplemente tiene una propiedad (heredable) que se coloca con el valor **2** en el constructor por defecto.

```
public class B extends A {  
    public void escribir(){  
        System.out.println(valor*2);  
    }  
}
```

La clase **B** es derivada de **A**. Ha redefinido el método escribir y además no tiene constructor. Pero puede utilizar la propiedad **valor** que procede de la clase A. Así este código:

```
public static void main(String[] args) {  
    A objA=new A();  
    B objB=new B();  
    objA.escribir(); //escribe 2  
    objB.escribir(); //escribe 4  
}
```

El método escribir del **objB** escribe un 4. La razón, aunque la clase B no tiene constructor, el compilador crea un constructor por defecto cuya única línea será una llamada al constructor de la superclase (**super()**). Ahí es donde valor se pone a dos y como el método escribe dobla el valor al escribir, escribe un cuatro.

Sin embargo si la clase A no tuviera constructor por defecto:

```
public class A {  
    protected int valor;  
    public A(int v){  
        valor=v;  
    }  
    public void escribir(){  
        System.out.println(valor);  
    }  
}
```

---

```
public class B extends A { //error no se puede invocar al  
//constructor por defecto  
    public void escribir(){  
        System.out.println(valor*2);  
    }  
}
```

## Ejercicio33

---

En un banco tenemos varios tipos de cuentas bancarias en las cuales es posible *retirar fondos y hacer depósitos*. Consideremos tres tipos de cuentas: **cuenta corriente, libreta de ahorro y libreta 2020**.

- i. **Las cuentas corrientes** se usan para realizar pagos y no proporcionan ningún interés. La cuenta corriente puede tener asociada una libreta de ahorro para cubrir descubiertos de la cuenta corriente.
- ii. **Las libretas de ahorro** proporcionan un interés del 4%.
- iii. **La libreta 2020** es un tipo de libreta de ahorro que rinde un mayor interés (del 8%) pero la retirada de fondos está penalizada con un 2% de la cantidad retirada.

*Con estas condiciones:*

- a) Diseñar las clases resultantes.
- b) Construir un programa ejemplo que use las clases del apartado a). En el programa deben crearse objetos de cada una de las clases, inicializando las cuentas con unos valores iniciales. Además, se retirarán fondos de cada uno de los tipos de cuenta, se actualizarán los intereses de la libreta 2020 y se depositará una determinada cantidad en la libreta 2020.

Solución:

```
public class B extends A {  
    char carácter;  
    public B(int v){  
        super(v);  
    }  
    public B(int v, char c){  
        this(v); //Ahora sí!! También valdría super(v)  
        carácter=c;  
    }  
    public void escribir(){  
        System.out.println(valor*2+" "+carácter);  
    }  
}
```

Esto puede marear. Pero en realidad las acciones de Java son lógicas. Hay que tener en cuenta permanentemente que en los constructores Java realiza estas acciones:

*Herramienta. Constructores.pdf*

- ◆ Si la primera instrucción de un constructor de una subclase no es una invocación a otro constructor con **this** o **super**, Java añade de forma invisible e implícita una llamada **super()** con la que invoca al constructor por defecto de la superclase. Luego continúa con las instrucciones de inicio de las variables de la subclase y luego sigue con la ejecución normal. Si en la superclase no hay constructor por defecto (sólo hay explícitos) ocurrirá un error.
- ◆ Si se invoca a constructores de superclases mediante **super(...)** en la primera instrucción, entonces se llama al constructor seleccionado de la superclase, luego inicia las propiedades de la subclase y luego sigue con el resto de sentencias del constructor.
- ◆ Finalmente, si esa primera instrucción es una invocación a otro constructor de la clase con **this(...)**, entonces se llama al constructor seleccionado por medio de **this** y realiza sus instrucciones, y después continúa con las sentencias del constructor. La inicialización de variables la habrá realizado el constructor al que se llamó mediante **this(..)**.

El uso de **super** y **this** no puede ser simultáneo puesto que ambas tienen que ser la primera instrucción. Lo que significa que hay que elegir entre ambas.

*Ejercicio nº 33*

*Pdf Adjunto*

## Conversión hacia abajo (casting de objetos)

En la conversión hacia arriba se gana generalidad pero se pierde información acerca del tipo concreto con el que se trabaja y, consecuentemente, se reduce la interfaz. Si se quiere recuperar toda la información del tipo con el que se trabaja (esto es, recuperar el acceso a todos los miembros del objeto) será necesario moverse hacia abajo en la jerarquía mediante un cambio de tipo.

Para ello se debe cambiar el tipo de la referencia (tipo estático) para que coincida con el tipo real del objeto (tipo dinámico) o al menos con algún otro tipo que permita el acceso al miembro del objeto deseado. Esta conversión a un tipo más específico es lo que se conoce como **conversión hacia abajo** y se realiza mediante una operación de casting.

```
Figura f = new Circulo(); // Conversion hacia arriba
Circulo c = (Circulo)f; // Conversion hacia abajo mediante
// cambio de tipo (casting)
// Ahora mediante la referencia c podemos acceder a TODOS los miembros
// de la clase Circulo. Con la referencia f SOLO podemos acceder a los
// miembros declarados en la clase Figura
```

También se puede hacer un cambio de tipo temporal sin necesidad de almacenar el nuevo tipo en otra variable referencia:

```
Figura f = new Circulo(); // Conversion hacia arriba
((Circulo)f).radio=1; // Acceso a miembros de la subclase
// A continuacion f sigue siendo de tipo Figura
```

En este caso mediante la sentencia `((Circulo)f)` se cambia el tipo de `f` únicamente para esta instrucción, lo que permite acceder a cualquier miembro definido en `Circulo` (por ejemplo al atributo `radio`). En sentencias posteriores `f` seguirá siendo de tipo `Figura`.

Las conversiones de tipo, sin embargo, deben realizarse con precaución: la conversión hacia arriba siempre es segura (por ejemplo un círculo, con toda certeza, es una figura, por lo que este cambio de tipo no generará problemas), pero la conversión hacia abajo puede no ser segura (una figura puede ser un círculo, pero también un rectángulo o cualquier otra figura). Dicho de otro modo, en la conversión hacia arriba se reduce la "visibilidad" que se tiene del objeto (se reduce la interfaz) mientras que en la conversión hacia abajo se amplía.

Debemos estar seguros de que esta ampliación de la parte visible se corresponde realmente con el objeto que hemos creado, de lo contrario se producirá un error en tiempo de ejecución. Imaginemos que tenemos una jerarquía de clases como la que se muestra en la siguiente figura:

```

1 public class ConversionHaciaAbajo {
2
3     public static void main(String[] args) {
4         Figura[] v = new Figura[2];
5
6         double aleatorio = Math.random();
7         if (aleatorio > 0.5) {
8             v[0] = new Rectangulo();
9             v[1] = new Circulo();
10        } else {
11            v[0] = new Circulo();
12            v[1] = new Rectangulo();
13        }
14    }
15 }
16 }
17
18

```

Si ahora pretendemos cambiar el radio de cualquier círculo que tengamos en el vector `v` nos encontramos con que no sería seguro hacer una conversión hacia abajo, ya que realmente desconocemos el tipo de objeto que hay almacenado en cada componente del vector. Sería interesante disponer de algún mecanismo que nos permitiera conocer el tipo de un objeto (tipo dinámico) para poder hacer la conversión hacia abajo con total seguridad. En Java, este mecanismo nos lo proporciona el operador `instanceof`.

El operador `instanceof` toma como primer operando una variable referencia y como segundo un tipo, según la siguiente sintaxis:

```
referencia instanceof Tipo
```

El resultado de la expresión anterior es `true` en caso de que el objeto asociado a la referencia (tipo dinámico) coincida con el tipo del segundo operador, y `false` en caso contrario.

Por ejemplo, para cambiar el radio de todos los círculos que se encuentran almacenados en un vector de figuras podría emplearse el siguiente código:

```

1 for( int i=0; i<v.length; i++ )
2     if( v[i] instanceof Circulo ) // Comprobacion del tipo dinamico
3         ((Circulo)v[i]).setRadio(10); // Conversion hacia abajo
segura

```

Como recomendación final cabría decir que el uso del operador `instanceof` debería restringirse a los casos en que sea estrictamente necesario. En la medida de lo posible **es preferible emplear métodos polimórficos** que permitan realizar las operaciones requeridas sin necesidad de conocer con exactitud el tipo concreto del objeto con el que se trabaja.

## Casting de Objetos

El uso de polimorfismo es una muy buena práctica de desarrollo para reutilizar funcionalidad, hacer el código escalable y entre otras cosas aprovechar las características de la programación orientada a objetos; pero a su vez nos lleva a hacer uso del Casting (cuando la ocasión lo permita), lo que consiste en convertir de un tipo de objeto a otro.

Suponiendo que tenemos las clases:

```
view source
print?
01.public class MathOperator {
02.    protected double operator1;
03.    protected double operator2;
04.    public double getOperator1() {
05.        return operator1;
06.    }
07.    public void setOperator1(double operator1) {
08.        this.operator1 = operator1;
09.    }
10.    public double getOperator2() {
11.        return operator2;
12.    }
13.    public void setOperator2(double operator2) {
14.        this.operator2 = operator2;
15.    }
16.
17.    public double calculate()
18.    {
19.        return 0;
20.    }
21.
22.    protected MathOperator(double operator1, double operator2){
23.        this.operator1 = operator1;
24.        this.operator2 = operator2;
25.    }
26.
27.}
28.
29.
30.public class Sum extends MathOperator {
31.
32.    protected Sum(double operator1, double operator2) {
33.        super(operator1, operator2);
34.    }
35.
36.    @Override
37.    public double calculate() {
38.        return operator1 + operator2;
```

```
Exception in thread "main" java.lang.ClassCastException: javahispano.oop.MathOperator  
cannot be cast to javahispano.oop.Sum  
  
at javahispano.oop.Sample.main(Sample.java:11)
```

Así es, un MathOperator no es un Sum por lo que el casting no es válido, y aquí notamos la diferencia entre una RuntimeException y un error de compilación.

Como mencionamos, existe el voto de confianza que nos da el compilador cuando hacemos éste tipo de casting, ya que si cambiamos un poco el código tenemos el siguiente caso:

```
view source  
print?  
01. public static void main(String[] args) {  
02.     MathOperator sum2 = new Sum(2, 3);  
03.     Sum sum3 = (Sum) sum2;  
04.     doCalculate(sum3);  
05. }  
06.  
07. private static void doCalculate(MathOperator operator){  
08.     System.out.println("Result: " + operator.calculate());  
09. }
```

El resultado de la ejecución es:

Result: 5.0

Podemos ver que el casting es de igual manera un downgrade, pero en éste caso sum2 aunque está declarado como tipo MathOperator tiene una instancia del tipo Sum, por lo cual el casting es válido; y adicionalmente el método doCalculate no está tipificado para recibir como parámetro un objeto de tipo Sum, sino que es un método general que va a presentar el cálculo de cualquier operación que herede de MathOperator, por lo que el siguiente código es perfectamente válido:

```
view source  
print?  
1.MathOperator sum2 = new Sum(2, 3);  
2.Sum sum3 = (Sum) sum2;  
3.Multiply m1 = new Multiply(4, 2);  
4.doCalculate (sum3);  
5.doCalculate (m1);
```

El resultado de la ejecución es:

De ésta manera nos aseguramos de que en tiempo de ejecución los castings no lancen una excepción. Cabe recalcar que el compilador si detecta cuando estamos tratando de hacer una conversión que realmente no va a ser posible, por ejemplo si tenemos:

```
view source  
print?  
1.Sum sum = (Sum) new File("file.txt");
```

Definitivamente vamos a tener un error de compilación ya que de ninguna manera va a ser posible que un objeto de tipo File sea convertido a uno de tipo Sum.

#### Tips de Examen:

- Es muy importante reconocer cuando el código no compila por un casting inválido y cuando van a ocurrir excepciones en tiempo de ejecución
- En las preguntas puedes encontrar que las superclases tratan de invocar métodos que sólo existen en los objetos heredados, por ejemplo:

```
view source  
print?  
1.MathOperator sum = new MathOperator (2, 3);  
2.sum.sumAbsolute();
```

Esto causa un error de compilación ya que quien tiene el método sumAbsolute es la clase Sum, inclusive si tenemos la siguiente forma:

```
view source  
print?  
1.MathOperator sum = new Sum (2, 3);  
2.sum.sumAbsolute();
```

Aunque el objeto sum es una instancia de Sum, al hacer el upgrade el compilador no reconoce el método sumAbsolute como parte de la clase MathOperator. Para que la ejecución sea correcta tendría que existir un casting:

```
view source  
print?  
1.MathOperator sum = new Sum (2, 3);  
2.((Sum)sum).sumAbsolute();
```

En éste caso es válido ya que sum es una instancia de Sum; y en el siguiente caso?:

```
view source  
print?  
1.MathOperator sum = new MathOperator (2, 3);
```

### (7.3.1) instanceof

### InstanceofTest.java

Permite comprobar si un determinado objeto pertenece a una clase concreta.  
Se utiliza de esta forma:

**objeto instanceof clase**

Comprueba si el objeto pertenece a una determinada clase y devuelve un valor **true** si es así. Ejemplo:

```
Coche miMercedes=new Coche();
if (miMercedes instanceof Coche)
    System.out.println("ES un coche");
if (miMercedes instanceof Vehículo)
    System.out.println("ES un coche");
if (miMercedes instanceof Camión)
    System.out.println("ES un camión");
```

En el ejemplo anterior aparecerá en pantalla:

```
ES un coche
ES un vehículo
//pero no es un camión
```

## (7.4) clases abstractas

### ClasesAbstractas.java

A veces resulta que en las superclases se desean incluir métodos teóricos, métodos que no se desea implementar del todo, sino que sencillamente se indican en la clase para que el desarrollador que desee crear una subclase heredada de la clase abstracta, esté obligado a sobrescribir el método.

A las clases que poseen métodos de este tipo (**métodos abstractos**) se las llama **clases abstractas**. Son clases creadas para ser heredadas por nuevas clases creadas por el programador. Son clases base para herencia, plantillas de clases. Las clases abstractas no pueden ser instanciadas (no se pueden crear objetos pertenecientes a clases abstractas).

Una clase abstracta debe ser marcada con la palabra clave **abstract**. Cada método abstracto de la clase, también llevará el **abstract**.

## (7.5) modificador final

Fuente: Java

Ya se ha comentado anteriormente el significado de esta palabra para utilizarse con constantes. No obstante en Java realmente no hay constantes como tal; el significado real de final en un elemento de Java es que dicho elemento no puede cambiar de valor, que es definitivo (en la práctica, una constante).

Pero hay matices dependiendo de donde se utilice final, así:

- ◆ **Delante del nombre de una propiedad o de una variable en su definición,** crea una constante. Pero puede ser una constante en **tiempo de compilación** (**final int X=9**) o en **tiempo de ejecución** (**final int X=(int) Math.random()\*10+1**). En el último caso el valor de X no se puede cambiar (es constante) pero es un valor que sólo se conocerá en tiempo de ejecución.
- ◆ **Delante del nombre de una variable de referencia a objetos.** Es decir se trata de una variable que toma la referencia a un objeto. En este caso también es un valor que se resuelve en tiempo de ejecución que es cuando se crea el objeto. Se trata de instrucciones del tipo:

**final Coche COCHE=new Coche();**

En este caso el valor **final** se refiere a que **COCHE** no puede hacer referencia a otro coche. Sin embargo el valor al que se hace referencia, sí se puede cambiar. Es decir:

**final Coche COCHE=new Coche();**  
**Coche c=COCHE;**  
**c.mover(200);**

El coche se ha movido, pese a que la referencia **COCHE** es constante. Pero es que el objeto no es constante, es la referencia la que no puede cambiar. **COCHE** no puede apuntar a otro objeto, pero el objeto al que apunta sí puede cambiar.

- ◆ **Para crear constantes blancas.** Se trata de constantes que se declaran sin indicar el valor:

**final int X;**

Que son utilizadas para recoger valores en la construcción y hacer que ese valor sea inmutable:

**public Constructor(int i) {**  
    **X=i; //ese valor es el tomado como constante, X no podrá**  
        **//cambiarlo**  
**}**

## Ejercicio 34

Se plantea desarrollar un programa Java que permita la gestión de una empresa agroalimentaria que trabaja con tres tipos de productos:

- Productos frescos.
- Productos refrigerados.
- Productos congelados.

Todos los productos llevan esta **información común**:

- Fecha de caducidad
- Número de lote.

A su vez, cada tipo de producto lleva alguna **información específica**:

- Los productos frescos deben llevar la fecha de envasado y el país de origen.
- Los productos refrigerados deben llevar el código del organismo de supervisión alimentaria.
- Los productos congelados deben llevar la temperatura de congelación recomendada.

Crear el código de las clases Java implementando una **relación de herencia** desde la *superclase Producto* hasta las *subclases ProductoFresco, ProductoRefrigerado y ProductoCongelado*.

Cada clase debe disponer de constructor y permitir establecer (set) y recuperar (get) el valor de sus atributos y tener un método que permita mostrar la información del objeto.

Crear una *clase ejecutable* llamada **Ejercicio34** donde se cree un objeto de cada tipo y se muestren los datos de cada uno de los objetos creados.

## Ejercicio35

---

Crear la clase final Math2 con los métodos estáticos máximo, mínimo, sumatorio y media, ampliando la funcionalidad de la clase Math. Los métodos recogerán un array de números reales (double) dado como parámetro y devolverán el resultado. Crear un método main para pruebas.

## Interfaces en Java

Como sabemos en Java no existe la herencia múltiple, pudiendo heredar solamente de una clase, las Interfaces son una gran herramienta para simular este concepto.....

En esta entrada describiremos un poco sobre lo que son las interfaces, sus principales características y veremos un diagrama que nos puede colaborar para su mejor entendimiento...

Para empezar, debemos saber que una Interface es una Clase completamente Abstracta, como regla, sabemos que las **clases abstractas** poseen como mínimo un método abstracto, pero hablando de una **interface**, todos sus métodos tienen que serlo...

Como vimos en la entrada sobre **conceptos básicos**, las interfaces muchas veces son definidas como un tipo de contrato entre las clases concretas que la implementen, ya que la clase que lo haga se encuentra obligada a definir los métodos abstractos que la componen.

Lo anterior quiere decir que cuando creamos un Interface, lo que hacemos es definir lo que la clase que la implemente podrá hacer, pero no indicamos la forma en que lo hará.

## Herencia Múltiple?

Podemos decir que las interfaces simulan la herencia múltiple ya que una clase puede implementar cualquier número de interfaces, además las interfaces pueden heredar uno o más números de interfaces mediante la palabra **extends**, pero jamás podrán heredar clases concretas...

En java se usa la palabra reservada **implements** para indicar que implementamos una interface, la estructura general de una clase que implementa una interface es la siguiente.

?

```
1 interface InterfacePrincipal {  
2  
3     public void metodoAbstracto();  
4  
5     public String otroMetodoAbstracto();  
6  
7 }  
8
```

Algunas de las anteriores características se verían así.

?

```
1  /**Las interfaces Heredan de cualquier cantidad de interfaces usando
2  extends*/
3  public interface PrimerInterface extends SegundaInterface,
4  TercerInterface
5  {
6
7  /**Las siguientes son declaraciones validas para los atributos*/
8  int ATRIBUTO1= 5;
9  public int ATRIBUTO2= 5;
10 public static int ATRIBUTO3= 5;
11 public final int ATRIBUTO4= 5;
12 static int ATRIBUTO5= 5;
13 final int ATRIBUTO6= 5;
14 static final int ATRIBUTO7= 5;
15 public static final int ATRIBUTO8= 5;
16
17 /**Las siguientes son declaraciones validas para los métodos*/
18 void metodoPrimerInterface1();
19 public void metodoPrimerInterface2();
20 abstract void metodoPrimerInterface3();
21 public abstract void metodoPrimerInterface4();
22 }
```

Como vemos las anteriores son algunos ejemplos de la forma de declarar atributos o métodos, cualquiera de las opciones anteriores es válida....

Hay que tener presente algo, ya vimos que tanto para clases Abstractas como para Interfaces la **herencia** es permitida, pero por ejemplo para este tipo componentes, si una **interface hereda** de otra, esta no está obligada a implementar los métodos que posee la **Interface padre**, ya que la implementación tanto de los métodos de la **clase padre** como de la **interface** que los **hereda** depende de la **clase concreta** que **implemente** dicha **interface**..... este principio también aplica a las **clases Abstractas**, si **una clase abstracta implementa una interface**, los métodos de esta no necesariamente se deben implementar en la **clase Abstracta**, pero si se tienen que implementar en la **clase concreta que herede de la clase abstracta**....

## ¿Cuándo Utilizarlas?

Su uso está muy ligado al concepto de herencia y cumple el mismo principio que aplicamos al usar clases abstractas, lo que buscamos es establecer un mecanismo donde podamos compartir características comunes entre clases diferentes, además al igual que

## (7.6) interfaces

En lenguajes como C++, las clases pueden heredar de varias clases. La limitación de que sólo se puede heredar de una clase, hace que haya problemas ya que muchas veces se deseará heredar de varias clases. Aunque ésta no es la finalidad directa de las interfaces, sí que tiene cierta relación

Mediante interfaces se definen una serie de **comportamientos** de objeto. Estos comportamientos puede ser **implementados** en una determinada clase. No definen el tipo de objeto que es, sino lo que pueden hacer (sus capacidades). Por ello lo normal es que el nombre de las interfaces terminen con el sufijo **able** (*configurable*, *modifiable*, *cargable*).

Por ejemplo en el caso de la clase *Coche*, esta deriva de la superclase *Vehículo*, pero además puesto que es un vehículo a motor, podría implementar métodos de una interfaz llamada por ejemplo *Arrancable*. Se dirá entonces que la clase *Coche* es *arrancable*.

En Java el manejo de interfaces es fundamental para comprender íntegramente todos los elementos del lenguaje. Las interfaces juegan un papel fundamental en la creación de aplicaciones Java. Permiten establecer un protocolo de funcionamiento de una serie de clases y eso facilita su uso y comprensión.

### (7.6.1) utilizar interfaces

Para hacer que una clase utilice una interfaz, se añade detrás del nombre de la clase la palabra **implements** seguida del nombre del interfaz. Se pueden poner varios nombres de interfaces separados por comas (solucionando, en cierto modo, el problema de la herencia múltiple).

```
class Coche extends Vehiculo implements Arrancable {
    public void arrancar (){
        ....
    }
    public void detenerMotor(){
        ....
    }
}
```

Hay que tener en cuenta que la interfaz *Arrancable* no tiene porque tener ninguna relación de herencia con la clase *Vehículo*, es más se podría implementar el interfaz *Arrancable* a una bomba de agua.

### (7.6.2) creación de interfaces

Una interfaz en realidad es una serie de **constants y métodos abstractos**. Es más el parecido entre interfaces y clases abstractas es muy grande. Cuando una clase implementa un determinado interfaz **debe** definir los métodos abstractos de éste. Esta es la base de una interfaz, en realidad no hay una

#### (7.6.4) variables de interfaz

Las variables de interfaz no son objetos. Se parece su uso, pero en realidad son variables que son capaces de hacer referencia a cualquier objeto de una clase que haya implementado la interfaz. Es decir una variable de la interfaz **Arrancable** puede utilizarse para hacer referencia a objetos de la clase **Coche** y de la clase **BombaAgua**, ya que ambas son clases que implementan dicha interfaz.

Esto permite cosas como:

```
Arrancable motorcito; //motorcito es una variable de tipo
                      // arrancable
Coche c=new Coche(); //Objeto de tipo coche
BombaAgua ba=new BombaAgua(); //Objeto de tipo BombaAgua
motorcito=c; //Motorcito apunta al coche
motorcito.arrancar(); //Se arrancará el coche
motorcito=ba; //Motorcito apunta a la bomba de agua
motorcito.arrancar(); //Se arranca la bomba de agua
```

El juego que dan estas variables es impresionante, manipulan todo tipo de objetos. Eso facilita la escritura de métodos genéricos para clases que implementen la misma interfaz

#### (7.6.5) interfaces como funciones de retroinvocación

En C++ una función de retroinvocación es un puntero que señala a un método o a un objeto. Se usan para controlar eventos. En Java se usan interfaces para este fin. La idea es utilizar los métodos de la interfaz sobre objetos de los que no conocemos la clase (uno de los usos más interesantes de las interfaces). Ejemplo:

```
public interface Escribible {
    void escribe(String texto);
}

class Texto implements Escribible {
    ...
    public void escribe(String texto){
        System.out.println(texto);
    }
}
```

Ejercicio uº 36 (nº 35b)

Crear la clase final Matriz que amplie las declaraciones de los métodos estáticos de la clase Math y que implemente una interfaz llamada Extremos compilada con el siguiente código fuente:

```
public interface Extremos(25)
    int min (int a[]); 
    int max (int a[]); 
    double min (double a[]); 
    double max (double a[]);
```

## Ejercicio36

---

Crearemos una superclase llamada **Electrodomestico** con las siguientes características:

- Sus atributos son **precio base, color, consumo energético** (letras entre A y F) y **peso** (podrán heredarse).
- Por defecto, el color será blanco, el consumo energético será F, el precioBase es de 100€ y el peso de 5 kg.
- Los colores disponibles son blanco, negro, rojo, azul y gris. No importa si el nombre está en mayúsculas o en minúsculas.
- Los constructores que se implementarán serán
  - Un constructor por defecto.
  - Un constructor con el precio y peso. El resto por defecto.
  - Un constructor con todos los atributos.
- Los métodos que implementará serán:
  - Métodos get de todos los atributos.
  - **comprobarConsumoEnergetico(char letra)**: comprueba que la letra es correcta, si no es correcta usará la letra por defecto. Se invocará al crear el objeto y no será visible.
  - **comprobarColor(String color)**: comprueba que el color es correcto, si no lo es usa el color por defecto. Se invocará al crear el objeto y no será visible.
  - **precioFinal()**: según el consumo energético, aumentará su precio, y según su tamaño, también. Esta es la lista de precios:

| Letra | Precio |
|-------|--------|
| A     | 100 €  |
| B     | 80 €   |
| C     | 60 €   |
| D     | 50 €   |
| E     | 30 €   |
| F     | 10 €   |

## Ejercicio37

---

Dada la interfaz **Ordenador** descrita a continuación:

```
public interface Ordenador {  
    int getFrecuenciaMicro();  
    int getTamañoDisco();  
    int getMemoria();  
    void mostrarCaracteristicas(int numCaracteristica);  
}
```

- a. Crear una nueva clase llamada **OrdenadorIncomplete** que implemente la interfaz ordenador y que sólo defina el método *getFrecuenciaMicro()*.
  - b. Declara la clase **OrdenadorPersonal** que extienda OrdenadorIncomplete, añadiendo un nuevo atributo que indique el tamaño del monitor en pulgadas.
  - c. Definir la clase **OrdenadorPortatil** que extiende a *OrdenadorIncomplete* en la que aparece un nuevo atributo que indica el peso del ordenador.
  - d. Definir una clase **Ejercicio37** en la que se definirá un main() donde se creará un vector catalogo que contenga 3 ordenadores portátiles y 2 personales.
  - e. Definir en la clase **Ejercicio37** un método mostrarCatalogo que mostrará por pantalla las características de todos los ordenadores del catálogo.
- Nota: El método mostrarCaracteristicas de la interfaz, en función del entero que reciba por parámetro, mostrará en pantalla una de las características del objeto.

y para ello habrá que redefinir esos métodos a fin de que se ajusten las necesidades particulares de cada clase.

### (7.7.1) comparar objetos. método equals;

La clase **Object** proporciona un método para comprobar si dos objetos son iguales. Este método es **equals**. Este método recibe como parámetro un objeto con quien comparar y devuelve **true** si los dos objetos son iguales.

No es lo mismo **equals** que usar la comparación de igualdad. Ejemplo (suponiendo que se ha definido correctamente el método **equals**):

```
Coche uno=new Coche("Renault","Megane","P4324K");
Coche dos=uno; //dos y uno son referencias al mismo coche
boolean resultado=(uno.equals(dos)); //Resultado valdrá true
resultado=(uno==dos); //Resultado también valdrá true
dos=new Coche("Renault","Megane","P4324K"); //los mismos datos
resultado=(uno.equals(dos)); //Resultado valdrá true
resultado=(uno==dos); //Resultado ahora valdrá false
```

En el ejemplo anterior **equals** devuelve **true** si los dos coches tienen el mismo modelo, marca y matrícula . El operador “**==**” devuelve **true** si las dos referencias que se comparan apuntan al mismo objeto.

Realmente en el ejemplo anterior la respuesta del método **equals** sólo será válida si en la clase que se está comparando (**Coche** en el ejemplo) se ha redefinido el método **equals**. Esto no es opcional sino obligatorio si se quiere usar este método.

El resultado de **equals** depende de cuándo consideremos nosotros que devolver verdadero o falso (es el creador de la clase el que sabe como compararla).

Para que el ejemplo anterior funcione, el método **equals** de la clase **Coche** sería:

```
public class Coche extends Vehículo{
    ...
    public boolean equals (Object o){
        if ((o!=null) && (o instanceof Coche)){
            if (((Coche)o).matricula==matricula &&
                ((Coche)o).marca==marca &&
                ((Coche)o).modelo==modelo))
                return true;
        } else
            return false
    }
}
```

### **Ejercicio 38**

---

Crear la clase Persona que implemente la Interface Comparable y que permita determinar las diferencias en edad y/o altura de dos personas.

//Clonación

```
Coche uno=new Coche();
Coche dos=(Coche)uno.clone();
```

En la última línea del código anterior, el cast (*Coche*) es obligatorio ya que *clone* devuelve forzosamente un objeto tipo *Object*.

Esta es una definición vaga del método *clone* ya que utiliza el clone de la propia clase *Object*. No es lo habitual. Lo habitual es arreglar los problemas que provoca *clone* con las referencias a objetos internas:

```
public class Coche extends Vehiculo implements Arrancable,
Cloneable{
    ...
    public Object clone(){
        try{
            Coche c=(Coche) super.clone();
            c.motor=(Motor) motor.clone();
            return c;
        }catch(CloneNotSupportedException cnse){
            return null;
        }
    }
}
```

Suponiendo que los coches constan de una propiedad llamada *motor* de clase *Motor*, clase que a su vez es *clonable*; el código anterior sería el correcto para clonar, de otro modo con el código anterior a éste, los dos coches, el original y el clonado, compartirían motor.

#### (7.7.4) método *toString*

Este es un método de la clase *Object* que da como resultado un texto que describe al objeto. La utiliza, por ejemplo el método *println* para poder escribir un método por pantalla. Normalmente en cualquier clase habría que definir el método *toString*. Sin redefinirlo el resultado podría ser:

```
Coche uno=new Coche();
System.out.println(uno); //Escribe: Coche@26e431 el código de clase
```

Si redefinimos este método en la clase *Coche*:

```
public String toString(){
    return("Velocidad :" + velocidad + "\nGasolina: " + gasolina);
}
```

Ahora en el primer ejemplo se escribiría la velocidad y la gasolina del coche.

Ejercicio nº 38

Crear la clase *Persona* que implemente la *interface Comparable* y que permita determinar las diferencias en edad y/o altura de dos personas.

Compararla y que permita determinar las diferencias en edad y/o altura de dos personas.

## (7.8.2) **clases internas regulares**

Las clases internas regulares son simplemente clases definidas dentro de otras clases (sean de forma pública o privada). Al definir una clase dentro de otra, estamos haciendo totalmente dependiente. Normalmente se realiza esta práctica para crear objetos internos a una clase (el motor de un coche por ejemplo), de modo que esos objetos pasan a ser atributos de la clase.

Por ejemplo:

```
public class Coche {  
    public int velocidad;  
    public Motor motor;  
  
    public Coche(int cil) {  
        motor=new Motor(cil);  
        velocidad=0;  
    }  
  
    public class Motor{ //Clase interna  
        public int cilindrada;  
        public Motor(int cil){  
            cilindrada=cil;  
        }  
    }  
}
```

El objeto **motor** pertenece a la clase **Motor** que es interna a la clase **Coche**. Si quisieramos acceder al objeto motor de un coche sería:

```
Coche c=new Coche(1200);  
System.out.println(c.motor.cilindrada); //Saldrá 1200
```

Las clases internas pueden ser privadas, protegidas o públicas.

Al compilar un clase que contiene clases internas se crean dos archivos **.class**, en el ejemplo sería **Coche.class** y **Coche\$Motor.class**. Sin embargo el comando **java Coche\$Motor** generaría un error ya que no podemos acceder a esa clase para su ejecución ya que no se puede definir nada estático en ella (y por lo tanto **main** tampoco).

Los objetos de la clase interna (en inglés **inner class**) sólo tienen sentido dentro de la clase contenedora o externa (en inglés **outer class**). Pero se pueden crear fuera si primero creamos un objeto de la clase contenedora. La sintaxis es compleja, sería así:

```
Coche c=new Coche(150);  
Coche.Motor m=c.new Motor(200);  
System.out.println(m.cilindrada); //200  
System.out.println(c.motor.cilindrada); //150
```

```
public class Motor{  
    public int cilindrada;  
    public Motor(int cil){
```

**Coche.this.cilindrada=cil;//Coche  
this.cilindrada=cil;//Motor, también vale sin la palabra this**

```
    }  
}
```

#### (7.8.4) Clases internas a un método

Se trata de clases que se definen dentro de un método. Sólo se pueden utilizar dentro del método. Su uso es un tanto controvertido. Ejemplo de uso:

```
public class Externa {  
    private String nombre = "Externa";  
    int aleatorio(){  
        class Interna extends Random{  
            Interna(){  
                super();  
            }  
            public int aleatorioPar(){  
                return nextInt() * 2;  
            }  
        }  
        Interna i=new Interna();  
        return i.aleatorioPar();  
    }  
    public static void main(String[] args) {  
        Externa e=new Externa();  
        System.out.println(e.aleatorio());  
    }  
}
```

En el código anterior la clase **Interna** se crea dentro del método **aleatorio**. La clase sólo está disponible para el método. Esta clase hereda de la clase **Random** el método **nextInt** (entre otros) que permite obtener un número entero aleatorio.

En el ejemplo esta técnica no parece muy útil ya que es evidente que lo que realiza la función aleatoria, se puede hacer de otras maneras (sin crear una clase aleatoria heredera de la clase **Random**).

La cuestión es ¿por qué utilizar este tipo de clases internas? Normalmente se usan cuando el método intenta solucionar un problema complicado y necesita apoyarse en una clase, pero no se necesita que esta clase esté disponible fuera. Son por tanto clases que quedan fuera del diseño.

Lo único que hace la clase creada es enviar el número 25 al propio ordenador a través del puerto de red 1200. Esto se podría hacer de forma más sencilla, pero se ha considerado hacer de esta forma y es completamente válido.

La clase **Msg** se ha creado dentro del **if** por ello sólo se puede utilizar en el bloque interior al **if**. Por eso intentar hacer un nuevo mensaje fuera de este bloque causa un error.

El uso de este tipo de clases internas es aún más limitado, nuevamente se usan sólo cuando se considera que ayudan a solucionar el problema. Pero es difícil encontrar situaciones reales para su uso.

### (7.8.6) Clases internas anónimas

Son el caso más curioso de clases internas. Se trata de clases que sirven para crear objetos cuya clase se crea sobre la marcha sin nombre. Un equivalente al ejemplo anterior mediante clase anónima es:

```
public static void main(String[] args) {  
    int i;  
    do{  
        i=Integer.parseInt(  
            JOptionPane.showInputDialog(  
                "Elija entre 1) Mandar un mensaje o 2) Salir"));  
        if(i==1){  
            try{  
                new Socket("localhost",1200){  
                    public void enviar() throws IOException {  
                        super.sendUrgentData(25);  
                    }  
                }.enviar();  
            }  
            catch (Exception e) {  
                JOptionPane.showMessageDialog(null, "Error");  
            }  
        }  
    }while(i!=2);  
}
```

No es el ejemplo ideal de clase anónima, pero sin duda es curioso. En el **if** se crea un objeto que desaparecerá cuando finalice el bloque ya que no se asigna a nadie.

En realidad es un objeto de tipo **Socket**, pero en lugar de simplemente crearle y utilizar después los métodos de esa clase, resulta que hay una llave justo detrás de la invocación al constructor.

Esa llave (en el código marcado en naranja fuerte) inicia la definición de la clase anónima. En esa definición se crea el método **enviar** para enviar el número 25 a través de la red.

## Ejercicio39

---

Crear una clase Java denominada **PilaInt** que atienda al funcionamiento de una **pila de enteros**.

La Pila estará construida sobre un array, y tendrá, entre otras, ~~con~~ las siguientes funcionalidades:

- Apilar elemento.
- Des apilar elemento.
- Borrar pila.
- ...

La Pila, debe contar también con un **Iterador** que posibilite recorrerla sin eliminar elementos. Este Iterador lo representará la **Clase Interna PasoaPaso**.

Crear un Método **main()**, que a través de un **Menú**, demuestre todas las funcionalidades de la Pila.

## ¿PARA QUÉ SE USAN LAS CLASES INTERNAS?

Después de todo lo visto la pregunta que nos hacemos es: ¿para qué sirven las clases internas? Normalmente la clase interna hereda de otra clase o implementa una interfaz y manipula el objeto de la clase externa dentro del cual hubiera sido creado. Por tanto, una clase interna proporciona una especie de ventana hacia la clase externa.

Pero, si simplemente necesitamos una referencia a una interfaz, ¿por qué no hacemos simplemente que la clase externa implemente dicha interfaz? La respuesta es: "Si eso es todo lo que necesita entonces esa es la manera de hacerlo". Por tanto, ¿qué distingue una clase interna que implementa una interfaz de una clase externa que implementa la misma interfaz? La respuesta es que no siempre disponemos de la posibilidad de trabajar con interfaces sino que a veces nos vemos obligados a trabajar con implementaciones. Por tanto, la razón más evidente para utilizar clases internas es la siguiente:

*Cada clase interna puede heredar de una implementación de manera independiente. Por tanto, la clase interna no está limitada por el hecho de si la clase externa ya está heredando de una implementación.*

Si las clases internas no pudieran heredar de más de una clase concreta o abstracta, algunos problemas de diseño y de programación serían intratables. Así, una forma de ver las clases internas es que representan al resto de la solución del problema de la herencia múltiple. Las interfaces resuelven parte del problema, pero las clases internas permiten en la práctica una "herencia de múltiples implementaciones". En otras palabras, las clases internas nos permiten en la práctica heredar de varios elementos que no sean interfaces.

Pensemos en una situación en la que tuviéramos dos interfaces que deban de ser implementadas dentro de una clase. Debido a la flexibilidad de las interfaces tenemos dos opciones: una única clase o una clase interna.

La definición de las clases anónimas está resaltado en naranja. Las referencias **msg** y **msg2** son de tipo **Escribible**, pero cada uno tiene una implementación distinta a través de la clase anónima. El primero permite escribir en la consola y el segundo en una ventana de mensaje.

Es muy importante tener en cuenta que el cierre de las llaves tiene que incorporar el punto y coma, ya que son clases definidas dentro de una instrucción de creación de objeto que tiene que finalizar como lo hace normalmente, es decir con el punto y coma.

Este código es muy interesante ya que permite observar el funcionamiento de estas clases. En principio la creación de los objetos parece normal hasta que llega la llave, entonces todo el código cambia y lo que parecía una instrucción sencilla de creación de objetos, se convierte en un código de creación de clase con una nueva implementación de la interfaz **Escribible**.

Esta definición de clase sólo está disponible en cada objeto que se ha creado, por lo que su uso parece muy limitado, de hecho lo es. En la práctica real implementar de forma muy particular una interfaz, una clase o una clase abstracta suele ser el uso más habitual de este tipo de clases.

### (7.8.7) **clases internas estáticas**

En realidad no son clases internas. Se las llama también clases anidadas estáticas para evitar el uso de la palabra **interna**, pero en general todo el mundo las conoce como clases internas estáticas.

Estas clases no mantienen ningún vínculo con la clase contenedora. Se pueden crear objetos de la clase interna sin crear ningún objeto contenedor. Ejemplo:

```
public class Coche {  
    public int velocidad;  
    public Motor motor;  
    public Coche(int cil) {  
        motor=new Motor(cil);  
        velocidad=0;  
    }  
    public static class Motor{ //Clase interna estatica  
        public int cilindrada;  
        public Motor(int cil){  
            cilindrada=cil;  
        }  
    }  
}  
  
public class PruebaCoche{  
    public static void main(String[] args) {  
        Coche.Motor m=new Coche.Motor(2000);  
        System.out.println(m.cilindrada);  
    }  
}
```

(37)

EJERCICIO N° 39

PdfAdjunto. Pila y do. Ejercicios

## Ejercicio 4

---

- Crea una clase llamada Persona que siga las siguientes condiciones:

· Sus atributos son: **nombre**, **edad**, **DNI**, **sexo** (H hombre, M mujer), **peso** y **altura**. No queremos que se accedan directamente a ellos. Piensa qué modificador de acceso será el más adecuado, así como su tipo. Si quieras añadir algún atributo más puedes.

Por defecto, todos los atributos menos el DNI serán valores por defecto según su tipo (0 números, cadena vacía para String, etc.). Sexo será H por defecto, usa una constante para ello.

· Se implantarán varios constructores:

- Un constructor por defecto.
- Un constructor con el nombre, edad y sexo, el resto por defecto.
- Un constructor con todos los atributos como parámetro.

· Los métodos que se implementaran son:

**calcularIMC()**: calculará si la persona está en su peso ideal (peso en kg/(altura<sup>2</sup> en cm)), si esta fórmula devuelve un valor menor que 20, la función devuelve un -1, si devuelve un número entre 20 y 25 (incluidos), significa que está ~~por debajo de~~ en su peso ideal la función devuelve un 0 y si devuelve un valor mayor que 25 significa que tiene sobrepeso, la función devuelve un 1. Se recomienda el uso de constantes para devolver estos valores.

**esMayorDeEdad()**: indica si es mayor de edad, devuelve un booleano.

**comprobarSexo(~~char~~ sexo)**: comprueba que el sexo introducido es correcto. Si no es correcto, sera H. No sera visible al exterior.

**toString()**: devuelve toda la información del objeto.

**generaDNI()**: genera un número aleatorio de 8 cifras, genera a partir de este su número su letra correspondiente. Este método será invocado cuando se construya el objeto. Puedes dividir el método para que te sea más fácil. No será visible al exterior.

Métodos set de cada parámetro, excepto de DNI.

- Ahora, crea una clase ejecutable que haga lo siguiente:

Pide por teclado el **nombre**, la **edad**, **sexo**, **peso** y **altura**.

• Crea 3 objetos de la clase anterior, el primer objeto obtendrá las anteriores variables pedidas por teclado, el segundo objeto obtendrá todos los anteriores menos el peso y la altura y el último por defecto, para este último utiliza los métodos set para darle a los atributos un valor.

Para cada objeto, deberá comprobar si está en su peso ideal, tiene sobrepeso o por debajo de su peso ideal con un mensaje.

Indicar para cada objeto si es mayor de edad.

Por último, mostrar la información de cada objeto.

## Uso de los paquetes en Java (packages)

Los paquetes en Java (packages) son la forma en la que Java nos permite agrupar de alguna manera lógica los componentes de nuestra aplicación que estén relacionados entre sí.

Los paquetes permiten poner en su interior casi cualquier cosa como: clases, interfaces, archivos de texto, entre otros. De este modo, los paquetes en Java ayudan a darle una buena organización a la aplicación ya que permiten modularizar o categorizar las diferentes estructuras que componen nuestro software.

Los paquetes en Java, adicionalmente al orden que nos permite darle a la aplicación, también nos brindan un nivel adicional de seguridad para nuestras clases, métodos o interfaces, pues como veremos más adelante podremos especificar si una clase o interfaz en particular es accesible por todos los componentes del software (sin importar el paquete) o si en realidad es solo accesible por las clases que estén en el mismo paquete que ésta. Veremos con más detalle este tema del acceso más adelante.

Ahora veamos cómo crear paquetes en Java

### ¿Cómo crear paquetes en Java?

Ya sabemos para qué sirven los paquetes en Java y sus características principales. Vamos ahora a aprender cómo usarlos y qué cambios generan estos en la estructura de nuestro proyecto.

Para declarar un paquete en Java se hace uso de la palabra reservada "package" seguido de la "ruta" del paquete, como se muestra a continuación.

```
package ruta.del.paquete;
```

Hay varias cosas que clarificar aquí. Como verás, la sintaxis es bastante simple y comprensible pero hay algunos detalles a tener en cuenta, veamos.

### Tips para crear paquetes en Java

- **El paquete en Java se declara antes que cualquier otra cosa:**

La declaración del paquete debe estar al principio del archivo Java, es decir, es la primera línea que se debe ver en nuestro código o archivo .java. Primero se declara el paquete, y luego podremos poner los [imports](#) y luego las clases, interfaces, métodos, etc.

- **Cada punto en la ruta del paquete es una nueva carpeta:**

Cuando se escribe la ruta del paquete en Java, se pueden especificar una ruta compleja usando el punto ".", por ejemplo en el código anterior, nuestra clase, interfaz o archivo estará ubicado en una carpeta llamada "paquete" la cual a su vez está en la carpeta "del" y esta también se encuentra dentro de una carpeta llamada "ruta". De este modo si queremos por ejemplo que una clase quede al interior de una carpeta llamada "mi\_paquete" y ésta al interior de una carpeta llamada "otro\_paquete" pondríamos:

```
package otro_paquete.mi_paquete;  
/*Se usa el punto para separar cada carpeta  
   equivale a la ruta otro_paquete/mi_paquete dentro del proyecto*/  
public class mi_clase  
{  
}
```

- **Si no se declara un paquete (paquete por defecto):**

Si decidimos no declarar un paquete para nuestra clase, ésta quedará en un paquete que se conoce como paquete por defecto (default package), en este paquete estarán



La clase número uno fue declarada en el paquete "mis\_clases.clases\_publicas.clase\_1" por lo tanto quedará al interior de una carpeta llamada "clase\_1" la cual estará al interior de otra carpeta llamada "clases\_publicas" y esta a su vez estará dentro de la carpeta llamada "mis\_clases".

La clase número dos se le declaró el paquete "mis\_clases.clase\_2" por lo tanto ha quedado al interior de una carpeta llamada "clase\_2" la cual a su vez está al interior de la carpeta "mis\_clases".

A la clase número tres se le impuso el paquete "mis\_clases", esto indica que esta estará al interior de la carpeta "mis\_clases".

Por último, la clase número cuatro no tiene declarado ningún paquete por lo tanto el paquete asignado será el paquete por defecto, en otras palabras, esta clase se mantendrá siempre en la carpeta raíz del proyecto (en netbeans y eclipse la carpeta raíz se llama src).

## Paquetes Java

Un **Paquete** en Java es un contenedor de clases que permite agrupar las distintas partes de un programa y que por lo general tiene cuya funcionalidad elementos comunes, definiendo la ubicación de dichas clases en un directorio de estructura jerárquica.

### Ventajas

El uso de paquetes proporciona las siguientes ventajas:

- Agrupamiento de clases con características comunes. *(finalidad, ámbito o horizonte)*
- Reutilización de código al promover principios de programación orientada a objetos como la encapsulación y modularidad.
- Mayor seguridad al existir niveles de acceso.
- Evita la colisión de clases que tengan el mismo nombre. Pueden existir clases con el mismo nombre siempre y cuando su **fully qualified class name** sean únicos.
- Mantenibilidad de código. Si un paquete se enfoca en la agrupación de clases con características comunes, el cambio en la funcionalidad se limita a las clases contenidas en dicho paquete, además, si es un paquete grande soporta la reusabilidad, si por el contrario es pequeño soporta su mantenibilidad.

### Contenido de un paquete

Un paquete puede contener:

- Clases
- Interfaces
- Tipos Enumerados
- Anotaciones

### Atributos de un paquete Java

Un paquete Java tiene los siguientes atributos:

- Son opcionales. Si no se define un paquete para un fichero de código Java se definirá un paquete llamado "*default*" automáticamente.
- Es definido un paquete por cada fichero de código Java.
- Los nombres de paquete que inician con el nombre de `java.*` y `javax.*` son reservados.
- El nombre del paquete equivale a una estructura de ficheros. El nombre de paquete `com.dominio_empresa.utilidades` debería ser igual al directorio `[ruta_directorio]\com\dominio_empresa\utilidades`. Si una clase incluida en un paquete no se relaciona con su respectiva estructura de directorio, la clase no podrá usarse.

Gracias a esta instrucción para utilizar la clase *Coche* no hace falta indicar el paquete en el que se encuentra, basta indicar sólo *Coche*. Se puede utilizar el símbolo asterisco como comodín.

Ejemplo:

```
import locomoción.*;  
//Importa todas las clase del paquete locomoción
```

Esta instrucción no importa el contenido de los paquetes interiores a *locomoción* (es decir que si la clase *Coche* está dentro del paquete *motor*, no sería importada con esa instrucción, ya que el paquete *motor* no ha sido importado, sí lo sería la clase *locomoción.BarcoDeVela*). Por ello en el ejemplo lo completo sería:

```
import locomoción.*;  
import locomoción.motor.*;
```

Cuando desde un programa se hace referencia a una determinada clase se busca ésta en el paquete en el que está colocada la clase y, si no se encuentra, en los paquetes que se han importado al programa. Si ese nombre de clase se ha definido en un solo paquete, se usa. Si no es así podría haber **ambigüedad** por ello se debe usar un prefijo delante de la clase con el nombre del paquete.

Es decir:

```
paquete.clase
```

O incluso:

```
paquete1.paquete2.....clase
```

En el caso de que el paquete sea subpaquete de otro más grande.

En el caso de los nombres de paquete, todo el nombre debe estar en minúsculas aunque lo formen varias palabras (por ejemplo *mipaquete*).

### (7.9.1) organización de los paquetes

Los paquetes en realidad son subdirectorios cuyo raíz debe ser absolutamente accesible por el sistema operativo. Para ello es necesario usar la variable de entorno **CLASSPATH**. Esta variable se debe definir en los archivos de inicio del sistema (como se comentó en el primer tema, véase **(1.3.3) instalación del SDK.**, Hay que añadirla las rutas a las carpetas que contienen los paquetes (normalmente todos los paquetes se suelen crear en la misma carpeta), a estas carpetas se las suele llamar **filesystems** de Java, nomenclatura muy habitual en los sistemas Linux/Unix. Ejemplo de classpath (Windows):

```
CLASSPATH=.;C:\Users\Jorge\java;D:\paquetesJava
```

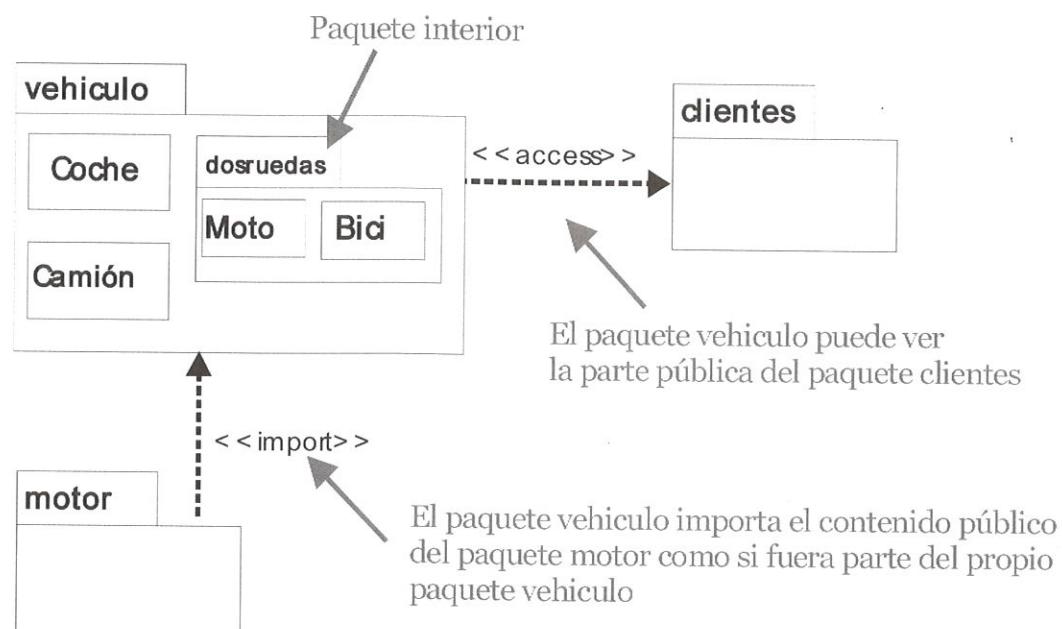


Ilustración 7-10, diagrama de paquetes UML