

viewnext.adrformacion.com © ADR Infor SL  
VICTOR TENA PALOMARES

# **Constructores, herencia e interfaces**

## **© ADR Infor SL**

viewnext.adrformacion.com © ADR Infor SL  
VICTOR TENA PALOMARES

viewnext.adrformacion.com © ADR Infor SL  
VICTOR TENA PALOMARES

# Indice

<b>Competencias y Resultados de Aprendizaje desarrollados en esta unidad</b>	<b>3</b>
<b>Constructores, herencia e interfaces</b>	<b>4</b>
Objetivo	4
Constructores	4
Constructores de clase	4
Destructores	6
Palabra reservada this	7
Sobrecarga de métodos	11
Sobrecarga de constructores	14
Herencia	16
Notas básicas sobre herencia	17
Polimorfismo	23
Instanceof	24
Palabra reservada super	26
Interfaces	29
Contenido de una interface	30
Implementación de interfaces en una clase	30
Creación de interfaces propias	31
Interfaces muy usadas que forman parte de la API	34
<b>Ejercicios</b>	<b>37</b>
Ejercicio 1. Línea de comandos	37
Recomendaciones	37
Datos a mostrar por consola	37
Ejercicio 2. Constructor	37
Pistas	37
Datos a mostrar por consola	38
Ejercicio 3. Instancias	38
Lo necesario para comenzar	38
Datos a mostrar por consola	39
Ejercicio 4. Instancias	40
Lo necesario para comenzar	40
Ejercicio 5. Herencia	41
Lo necesario para comenzar	41
<b>Recursos</b>	<b>43</b>
Enlaces de Interés	43

## Competencias y Resultados de Aprendizaje desarrollados en esta unidad

### Competencia:

Conocer el concepto de herencia en un lenguaje de programación orientado a objetos.

### Resultados de Aprendizaje:

- Entender el uso de los constructores y objetos en Java
- Conocer las instancias de un objeto en Java.
- Conocer los conceptos básicos de la herencia en Java.
- Conocer el concepto de Interfaz en Java
- Utilizar correctamente las interfaces en Java
- Utilizar los constructores de Java para crear objetos

# Constructores, herencia e interfaces

## Objetivo

- Entender el uso de los constructores en Java.
- Entender la herencia y el polimorfismo en Java.

## Constructores.

Cuando se construye un objeto (clase Java) es necesario inicializar sus variables con valores coherentes.

Podemos imaginar una clase **Vehiculo** en el que el número de ruedas, al crearlo, debe de ser 4. Para solucionar este apartado en los lenguajes orientados a objetos es emplear los constructores.

### Vocabulario: Constructor

**Los constructores son métodos especiales asociados a una clase que sirven para crear o instanciar objetos de esa clase dando valores coherentes a sus variables.**

En este tema se van a estudiar los constructores de clases creadas por el programador. Los constructores de clases de la API ya se comentaron en la unidad 2 del curso. A efectos prácticos se comportan del mismo modo.

## Constructores de clase.

Los constructores tienen **varias características** que los distinguen del resto de métodos:

Nombre
Los métodos constructores deben tener el mismo nombre que su clase asociada.
Tipo de retorno
No tienen tipo de retorno, ni siquiera debemos escribir la cláusula void en la definición del método.
Modificadores admitidos
Solamente admiten modificadores de acceso (public, private, protected, ...) y sin modificador. No se admiten otro tipo de modificadores como native, abstract, static, synchronized o final.

**Ejecución.**

Pueden existir varios métodos constructores en una clase, pero tan sólo se ejecutará uno al crear un objeto de la clase.

Al constructor de una clase se le invoca mediante el nombre de la clase y la palabra reservada `new`.

```
UnaClase uc = new UnaClase();
```

Cuando se invoca al constructor de una clase se realizan varias funciones

- **Se crea un objeto de la clase** con el fin de acceder a sus métodos y/o variables de instancia.
- **Se ejecuta el código** del constructor de la clase.
- **Se inicializan a su valor por defecto las variables de instancia** asociadas a los objetos de la clase, en caso de que no se inicialicen explícitamente dentro del constructor.

Pueden darse dos casos diferentes

**Se define el constructor en el código**

Su código se ejecutará siempre.

Suele contener líneas de inicialización de variables de instancia además de todo lo que el programador considere oportuno.

Si el programador no lo escribe, el compilador añadirá la línea **`super()`**; al inicio del método constructor. Esta línea hará que se ejecute el método constructor de la clase padre antes de ejecutar el constructor propio.

**No se define el constructor en el código**

El compilador crea un método constructor implícitamente.

Este constructor no tiene argumentos y tiene el mismo modificador que la clase.

Este método solamente contendrá una línea que será **`super()`**;

Será el método que se ejecutará al crear un objeto de la clase.

Con `super()` se invoca al constructor de la superclase de la clase en curso.

```
1 //Dado este código
2 public class UnaClase{}
3
4 //El compilador lo interpreta como
5 /*
6 public class UnaClase{
7     public UnaClase(){
8         super();
9     }
10 }
11 */
```

Este constructor es el que permite al programador crear un objeto de la clase sin necesidad de definirlo explícitamente en el código.

## Destruyores

Algunos lenguajes de programación, como C por ejemplo, añaden métodos destructores a la creación de los objetos. En Java el programador está liberado de esta labor, existe un recolector de basura (garbage collector) que se encarga de eliminar de la memoria los objetos desreferenciados.

### Vocabulario: Garbage Collector

El Garbage Collector o **recolector de basura** en castellano, es el encargado de eliminar de la memoria todos los objetos a los que ninguna variable hace referencia. Esto puede suceder por varias circunstancias:

- porque se ha acabado el bloque de código para el que fueron creados (Caso 1 del ejemplo).
- porque esa variable apunta hacia otro objeto (Caso 2 del ejemplo) .
- porque la variable apunta a null o un objeto nulo (Caso 3 del ejemplo).

```

public class GarbageCollector{
    public static void main(String args[]){
        String s="Jesus";
        s=s+" Fernandez";
        //(CASO 2) A partir de la línea 4, la String de contenido Jesus
        //se convierte en un objeto desreferenciado y, por tanto,
        //seleccionable por el recolector de basura.

        Integer i=new Integer(10);
        i=new Integer(20);
        //(CASO 2)Dp de línea 9, el Integer de contenido 10 se desreferencia
        //y pasa a ser seleccionable por el recolector de basura.

        i=null;
        //(CASO 3) A partir de la línea 13 el Integer de contenido 20 es
        //seleccionable por el recolector de basura.
        //Causa: asignación de null a la variable que apunta al objeto

        GarbageCollector gc=new GarbageCollector();
        gc.unMetodo("1000");
    }

    //(CASO 1)Cuando se ejecute este método, el Integer de contenido
    //1000 se desreferencia y pasa a ser seleccionable por el recolector.
    public void unMetodo(String s){
        Integer intObj=new Integer(s);
    }
}

```

Es importante tener en cuenta que el recolector de basura es impredecible, no se sabe cuando va a actuar e incluso puede no hacerlo nunca si no existe falta de memoria. Existe un modo de invocarlo, se trata de la sentencia **System.gc()**. Este método no es muy fiable ya que la Máquina Virtual de Java (JVM) lo interpreta como un consejo de no obligado cumplimiento.

## Palabra reservada this

Relacionada con los constructores aunque no sólo con ellos, se tiene la palabra reservada "this". Esta palabra reservada **hace referencia al objeto en curso** asociado a la clase de Java que se está programando. Cuando se declara una clase ya se tienen en mente los objetos que pueden crearse con la misma. Bueno, pues siempre hay uno que es el actual o en curso. A ése se hace referencia cuando se emplea this.

La palabra clave this suele emplearse de estas dos formas:

**Como una instrucción seguida del operador (.)**

De esta forma se persigue acceder al método o inicializar la variable de instancia.

Se puede ver un ejemplo en 'Ejemplo 1 con constructor' en esta misma sección.

```
this.<nombreMetodo>;
this.<variableInstancia>=<algo>;
```

**Como una instrucción no seguida del operador punto**

Se emplea para invocar a un constructor sobrecargado de la clase.

Suele tener parámetros incluidos.

Se puede ver un ejemplo en el apartado 'Sobrecarga de constructores'.

```
this("Jesus");
```

**Como un parámetro**

Se pasa como argumento de un método.

Se emplea en la programación de eventos.

Se pueden ver dos ejemplos, uno en 'Ejemplo 1 con constructor' y otro al final de la unidad.

```
addActionListener(this);
```

La palabra clave this no se admite dentro de métodos estáticos. Su uso produce un error de compilación.

**Ejemplo: Ejemplo 1 sin constructor**

Se desea escribir una clase que calcule el área de dos rectángulos sin emplear constructor.



```

public class RectanguloSinConstructor{
    double x1,y1,x2,y2;

    double calculaArea(){
        return Math.abs((x2-x1)*(y2-y1));
    }
    public static void main(String args[]){

        /*
        * Se invoca al constructor sin argumentos con el que, ímplicitamente,
        * cuentan todas las clases Java que no definen constructor.
        * Esta llamada, instancia un objeto de la clase e inicializa
        * sus variables de instancia a los valores por defecto
        */
        RectanguloSinConstructor r1=new RectanguloSinConstructor();

        //Iniciación
        r1.x1=1;
        r1.y1=4;
        r1.x2=7;
        r1.y2=6;

        //Cálculo del área
        System.out.println("Area1="+r1.calculaArea());

        System.out.println("-----");

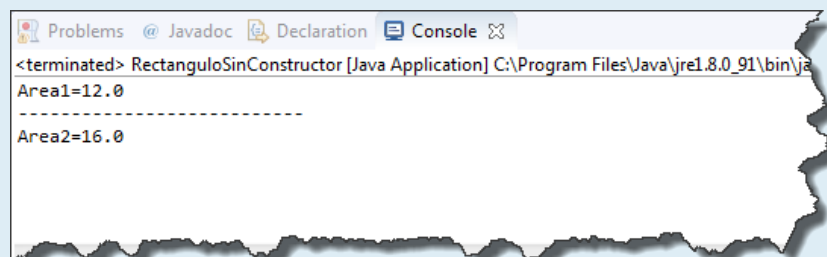
        //Otro objeto rectángulo
        RectanguloSinConstructor r2=new RectanguloSinConstructor();

        //Iniciación
        r2.x1=1;
        r2.y1=4;
        r2.x2=9;
        r2.y2=6;

        //Cálculo del área
        System.out.println("Area2="+r2.calculaArea());

    }
}

```

**Por consola:**

**Ejemplo: Ejemplo 1 con constructor**

Se trata de repetir el ejercicio anterior, pero con un constructor que cuenta con cuatro argumentos y que aparece de forma explícita en el código.  
Además, se utiliza de varias formas la palabra reservada this.

```
public class RectanguloConConstructor{
    double x1,y1,x2,y2;

    //Constructor con cuatro argumentos.
    //Mismo nombre que la clase y no devuelve nada.
    //En este caso, inicializa las variables de instancia de los dos objetos
    //que se crean.
    RectanguloConConstructor(double x1,double y1,double x2,double y2){
        System.out.println("Inicio constructor");

        /*
         * Se emplea this para asignar el valor de los argumentos que le
         * llegan al constructor a las variables de instancia siempre que
         * sus nombre coincidan, como ocurre aquí. Las variables precedidas
         * de this son de instancia, mientras que las otras son locales y
         * asociadas a los argumentos del constructor. Si los nombres, no
         * coincidieran no sería necesario emplear this.
         * this es MUY EMPLEADO PARA ESTA TAREA.
         */
        this.x1=x1;
        this.y1=y1;
        this.x2=x2;
        this.y2=y2;
        System.out.println("Fin constructor");
    }
    double calculaArea(){

        //Uso de this para invocar a un método pasándole el objeto en curso
        mostrarEstado(this);
        return Math.abs((x2-x1)*(y2-y1));
    }
    void mostrarEstado(RectanguloConConstructor r){
        System.out.println("Estado:"+r.x1+", "+r.y1+", "+r.x2+", "+r.y2);
    }
    public static void main(String args[]){

        //Llamada al constructor pasándole los parámetros que está esperando
        RectanguloConConstructor r1=new RectanguloConConstructor(1,4,7,6);
        System.out.println("Area1="+r1.calculaArea());

        System.out.println("-----");

        RectanguloConConstructor r2=new RectanguloConConstructor(1,4,9,6);
        System.out.println("Area2="+r2.calculaArea());
    }
}
```

**Por consola:**

```
<terminated> RectanguloConConstructor (1) [Java Application] C:\Program Files\Java\jdk1.8.0_121_2\bin\javaw
Inicio constructor
Fin constructor
Estado:1.0, 4.0, 7.0, 6.0
Area1=12.0
-----
Inicio constructor
Fin constructor
Estado:1.0, 4.0, 9.0, 6.0
Area2=16.0
```

Ahora conviene hacer los dos primeros ejercicios del tema

## Sobrecarga de métodos

La sobrecarga de métodos consiste en disponer en una misma clase de varios métodos con el mismo nombre pero con distinto número de argumentos, distinto tipo de argumentos o distinto orden.

Como hemos visto en la definición hay varias situaciones diferentes en las que se puede sobrecargar un método:

### Distinto número de argumentos

En la misma clase existen varios métodos con el mismo nombre pero con diferente número de argumentos.

### Distinto tipo de argumentos

En la misma clase existen varios métodos con el mismo nombre pero con diferente tipo de argumentos.

### Distinto orden de argumentos

En la misma clase existen varios métodos con el mismo nombre pero tienen los argumentos en diferente orden.

La sobrecarga de métodos es un concepto muy importante en la programación orientada a objetos. Su mayor virtud es que permite al programador crear métodos que actúen del mismo modo con independencia de los argumentos que reciba.

Los métodos **println()** y **print()** de la librería **java.io** son un ejemplo de este comportamiento.

El programador, a la hora de mostrar por consola un **int**, un boolean o un **String**, se despreocupa del tipo ya que estos dos métodos están sobrecargados y es el compilador el que se encarga de elegir el que corresponda en función del tipo de argumento que se le pasa.

Si no existiera la sobrecarga cada tipo a mostrar requeriría de un método, cada uno con un nombre diferente, lo que repercutiría en un mayor esfuerzo en memoria.

El compilador decide cuál es el método a elegir, pero la llamada debe contener el número adecuado de argumentos, el tipo adecuado y el orden correcto con respecto a alguno de los métodos sobrecargados.

El tipo de dato de retorno que devuelve un método no se considera elemento diferenciador en la sobrecarga de métodos.

viewnext.adrformacion.com © ADR Infor SL  
VICTOR TENA PALOMARES

viewnext.adrformacion.com © ADR Infor SL  
VICTOR TENA PALOMARES

ar SL

```

public class SobrecargaMetodos {
    void metodo(int numero,String nombre){
        System.out.println(numero);
        System.out.println(nombre);
        System.out.println("Estoy en el primer metodo");
    }
    void metodo(String nombre){
        System.out.println(nombre);
        System.out.println("Estoy en el segundo metodo");
    }
    void metodo(double numero,char caracter){
        System.out.println(numero);
        System.out.println(caracter);
        System.out.println("Estoy en el tercer metodo");
    }
    void metodo(String nombre,int numero){
        System.out.println(numero);
        System.out.println(nombre);
        System.out.println("Estoy en el cuarto metodo");
    }
    public static void main(String args[]){
        SobrecargaMetodos sm=new SobrecargaMetodos();
        sm.metodo(10,"hola");
        System.out.println("_____");
        sm.metodo("hola");
        System.out.println("_____");
        sm.metodo(25.5,'e');
        System.out.println("_____");
        sm.metodo("hola",10);
        System.out.println("_____");
        System.out.println("FIN DE PROGRAMA");
    }
}

```

```

<terminated> SobrecargaMetodos [Java Application] C:\Program Files\Java\jre1.8.0_91\bin\javaw.exe
10
hola
Estoy en el primer metodo
_____
hola
Estoy en el segundo metodo
_____
25.5
e
Estoy en el tercer metodo
_____
10
hola
Estoy en el cuarto metodo
_____
FIN DE PROGRAMA

```

Si se agrega este método:

```
String metodo(String nombre, int numero){  
    numero;  
    return nombre;  
}
```

se produce un error al compilar ya que el tipo de retorno no se considera válido para la sobrecarga de métodos.

En las diferentes APIs de Java encontraremos muchísimos métodos sobrecargados. Se puede acceder al API para revisar los métodos `print(..)` y `println(...)` de la clase `java.io.PrintStream`

## Sobrecarga de constructores

Un constructor puede sobrecargarse del mismo modo en que se sobrecarga un método.

De hecho, los constructores son métodos, si bien un tanto especiales.

Los constructores no admiten tipo de retorno.  
Los constructores deben tener el mismo nombre de la clase en la que se encuentran.

**Basado en el código de RectanguloConConstructor**

```

class RectanguloSobrecargado {
double x1,y1,x2,y2;
String nombre;

RectanguloSobrecargado(){

//Se invoca al constructor de 4 argumentos de tipo double.
//Recordar uso de this como instrucción no seguida del operador punto
this(1,4,9,6);
}
RectanguloSobrecargado(double x1,double y1,double x2,double y2){
this.x1=x1;
this.y1=y1;
this.x2=x2;
this.y2=y2;
}
RectanguloSobrecargado(double x1,double y1,double x2,double y2,
String nombre){
this.x1=x1;
this.y1=y1;
this.x2=x2;
this.y2=y2;
this.nombre=nombre;
}
double calculaArea(){
return (x2-x1)*(y2-y1);
}
}
public static void main(String args[]){
RectanguloSobrecargado rs;

rs=new RectanguloSobrecargado();
System.out.println("Area rectangulo= "+rs.calculaArea());

System.out.println("_____");

rs=new RectanguloSobrecargado(1,4,9,6);
System.out.println("Area rectangulo sin nombre="+rs.calculaArea());

System.out.println("_____");

rs=new RectanguloSobrecargado(1,4,9,6,"CUCU");
System.out.println("Area rectangulo "+rs.nombre+"="+rs.calculaArea());

System.out.println("_____");
System.out.println("FIN DE PROGRAMA");
}
}

```

**Por consola:**

```

<terminated> RectanguloSobrecargado [Java Application] C:\Program Files\Java\jre1.8.0_91\bin\javaw.exe
Area rectangulo= 16.0
Area rectangulo sin nombre=16.0
Area rectangulo CUCU=16.0
FIN DE PROGRAMA

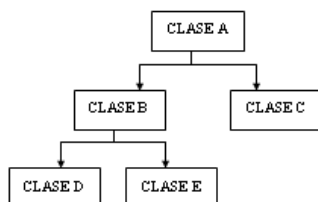
```

## Herencia

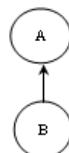
La herencia es un mecanismo que **permite compartir automáticamente variables y métodos 'no privados' entre clases** sin necesidad de reescribir código.  
Una clase heredada se llama **superclase**.  
Una clase que hereda se llama **subclase**.

Cuando un programador cuando emplea una clase que hereda **tiene acceso a todos los métodos y variables** de instancia de la clase en curso y a todos los métodos y variables de instancia no privados de la superclase.

**Esquema:**



CLASE B y CLASE C son subclases de la CLASE A.  
CLASE A es superclase de las clases B y C.  
CLASE D y CLASE E son subclases de la CLASE B.  
CLASE B es superclase de las clases D y E.



B es subclase de A.  
B hereda de A.

Una clase puede heredar de otra que haya sido previamente creada o de una de las clases de la API de Java.



En Java, la clase madre o raíz de todas las clases se llama **Object** y pertenece al paquete **java.lang**.

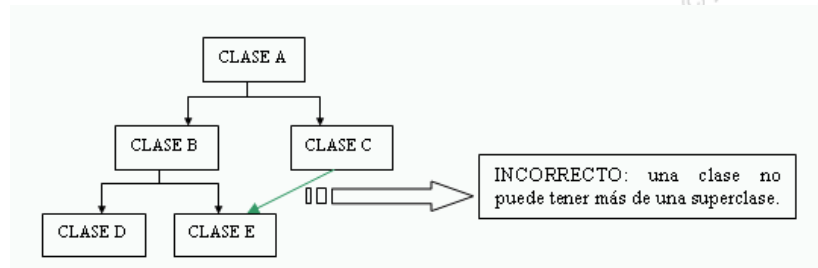
**Cuando un programador declara una clase, por defecto siempre hereda de Object** y, por tanto, puede utilizar todos sus métodos no privados (es como la raíz de cualquier clase que defina el programador).

Todas las clases de la API tienen como clase madre también a Object, aunque no todas directamente, algunas heredan a través de otras.

## Notas básicas sobre herencia

### Herencia múltiple

En Java no se admite la herencia múltiple, cada clase solamente puede tener una superclase.



### Subclases

Cada clase puede tener una cantidad ilimitada de subclases.

### Extends

Para declarar que una clase hereda de otra se utiliza la palabra reservada "extends" de este modo:

<modificadores de acceso> class <nombre de la clase> extends <nombre de la superclase>

**public class Coche extends Vehiculo**

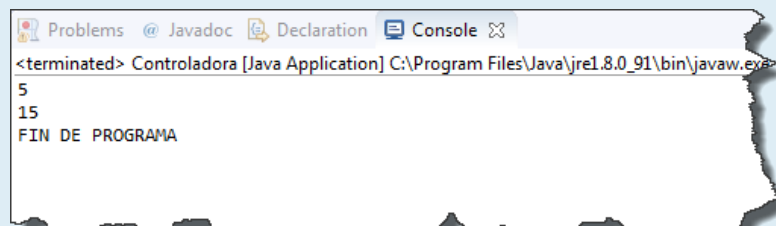
Se van a emplear tres clases en un mismo código fuente. Se podría hacer con tres códigos fuente distintos sin ningún problema.

```
class Alfa{
    int z=5;
    void sumaAlfa(){
        System.out.println(z);
    }
}

//El hecho de que Beta herede de Alfa equivale a decir que todas
//las variables de instancia no privadas y los métodos no privados
//de la clase Alfa pueden considerarse pertenecientes a la clase Beta
class Beta extends Alfa{
    int x=10;
    void sumaBeta(){
        //z se emplea como variable de instancia de la clase Beta siendo,
        //en realidad, variable de instancia de la clase Alfa.
        //Es posible gracias a la herencia.
        System.out.println(x+z);
    }
}

public class Controladora{
    public static void main(String args[]){
        Beta bet=new Beta();
        //Se puede acceder a todos los métodos no privados de Alfa mediante
        //un objeto de Beta gracias a la herencia
        bet.sumaAlfa();
        bet.sumaBeta();
        System.out.println("FIN DE PROGRAMA");
    }
}
```

**Por consola:**

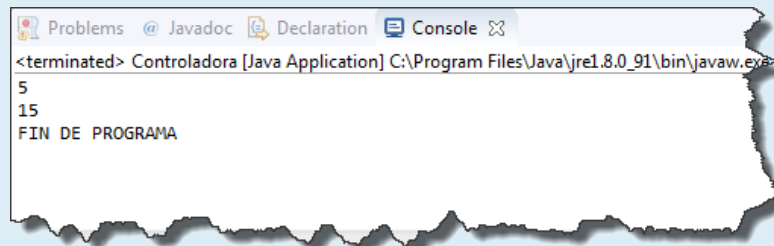


Se puede conseguir el mismo resultado haciendo que la clase principal (Controladora) herede de Beta pero, por convenciones, la clase principal nunca hereda de ninguna otra clase que no sea la Object.

```
class AlfaBis {
    int z=5;
    void sumaAlfa(){
        System.out.println(z);
    }
}

class BetaBis extends AlfaBis {
    int x=10;
    void sumaBeta(){
        System.out.println(x+z);
    }
}

public class ControladoraBis extends BetaBis {
    public static void main(String args[]){
        ControladoraBis cb=new ControladoraBis();
        cb.sumaAlfa();
        cb.sumaBeta();
        System.out.println("FIN DE PROGRAMA");
    }
}
```

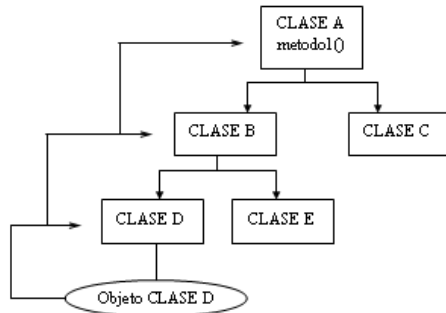


Si se considera privada la variable de instancia "z" de la clase Alfa en cualquiera de los ejemplos anteriores, ¿qué ocurrirá?

Sucedará un error de compilación pues **no puede accederse a variables ni métodos con modificador de acceso private** cuando se hereda.

## Invocación

Cuando se invoca a un método o a una variable de instancia mediante un objeto, Java busca la definición del método en la clase del objeto, y si no la encuentra, sigue buscándola siguiendo la pauta de su jerarquía de superclases.



La llamada al metodo1() se realiza mediante el objeto de la CLASE D. Java busca el método en la clase D; si no lo encuentra continúa buscando en la clase B (su superclase inmediata) y así sucesivamente hasta llegar a la CLASE A (superclase raíz de la CLASE D).

## Sobreescribir

La herencia se utiliza para métodos que tienen el mismo comportamiento, pero también puede que algunas veces nos interese que un mismo método (misma declaración, mismos argumentos y en el mismo orden) tenga comportamientos diferentes en cada clase que la heredará.

El comportamiento en este caso se denomina sobreescribir (overriding en inglés).

El método mantiene el nombre, tipo de retorno y argumentos en todas las subclases pero se modifica el contenido que se ejecutará al instanciarlo.

**Redefinición  
Sobreposición  
Sobrescritura**

**de métodos: crear una definición distinta, pero con la misma firma, de un método que ha sido definido originariamente en una superclase.**

```

class Animal{
    void comer(){
        System.out.println("Comiendo comida generica un animal generico");
    }
}

class Vaca extends Animal{
    //Redefinición del método comer() en la subclase Vaca
    void comer(){
        System.out.println("Metodo comer() redefinido en la clase Vaca.");
        System.out.println("Comiendo hierba una vaca generica");
    }
    void vaquear(){
        System.out.println("Vaqueando");
    }
}

class Toro extends Animal{
    //Redefinición del método comer() en la subclase Toro
  
```

```

    void comer(){
        System.out.println("Metodo comer() redefinido en la clase Toro.");
        System.out.println("Comiendo hierba un toro generico");
    }
    void torear(){
        System.out.println("Toreando");
    }
}

class EjemploAnimales {
    public static void main(String args[]){
        Animal ani=new Animal();
        ani.comer();
        Vaca vac=new Vaca();
        vac.comer();
        vac.vaquear();
        Toro tor=new Toro();
        tor.comer();
        tor.torear();

        //Polimorfismo:posibilidad de inicializar una variable referenciada
        //de la superclase mediante un objeto de una subclase
        //Mediante dicha variable "polimórfica",se puede acceder a todos los
        //métodos redefinidos en la subclase. NO A LOS PROPIOS DE LA CLASE.
        Animal aniVaca=new Vaca();
        aniVaca.comer();
        Animal aniToro=new Toro();
        aniToro.comer();

        //Error al compilar si se descomentan
        //aniVaca.vaquear();
        //aniToro.torear();
        System.out.println("FIN DE PROGRAMA");
    }
}

```

Código fuente

**Por consola:**

```
Problems @ Javadoc Declaration Console
<terminated> EjemploAnimales [Java Application] C:\Program Files\Java\jre1.8.0_91\bin\javaw.e
Comiendo comida generica un animal generico
Metodo comer() redefinido en la clase Vaca.
Comiendo hierba una vaca generica
Vaqueando
Metodo comer() redefinido en la clase Toro.
Comiendo hierba un toro generico
Toreando
Metodo comer() redefinido en la clase Vaca.
Comiendo hierba una vaca generica
Metodo comer() redefinido en la clase Toro.
Comiendo hierba un toro generico
FIN DE PROGRAMA
```

Los métodos estáticos o los que llevan el modificador final no pueden redefinirse, se produce un error de compilación.

Los métodos redefinidos en las subclases pueden ampliar los derechos de acceso de los métodos declarados en superclases, pero nunca restringirlos.

Así, los métodos declarados public en una superclase, si se redefinen en alguna subclase, deben ser declarados también public.

Los declarados protected en la superclase, protected o public en sus subclases.

Los declarados sin modificador en la superclase, no pueden ser private en sus subclases, etc.

**Este código no compilaría debido al modificador "private" en el método comer() de la clase Vaca**

```
package unidad06.ejemplos;
class Animal{
    void comer(){
        System.out.println("Comiendo comida generica un animal generico");
    }
}

class Vaca extends Animal{
    //Redefinición del método comer() en la subclase Vaca
    private void comer(){
        System.out.println("Metodo comer() redefinido en la clase Vaca.");
        System.out.println("Comiendo hierba una vaca generica");
    }
}

class EjemploAnimales {
    public static void main(String args[]){
        Animal ani=new Animal();
        ani.comer();
        Vaca vac=new Vaca();
        vac.comer();
        Animal aniVaca=new Vaca();
        aniVaca.comer();
        System.out.println("FIN DE PROGRAMA");
    }
}
```

## Polimorfismo

El polimorfismo es uno de los pilares de la Programación orientada a objetos.

Sus principales características son:

**Acceso a métodos redefinidos.**

El polimorfismo permite el acceso a métodos redefinidos mediante variables referenciadas polimórficas (inicializadas mediante un objeto de una subclase).

En el código anterior hay dos objetos de este tipo, **aniVaca** y **aniToro**.

**Array con elementos de diferentes clases**

Permite que un array contenga objetos de diferentes clases, siempre y cuando estén relacionados por herencia (hablando técnicamente, cumplan la relación IS A).

A nivel de programación se comprueba con el operador **instanceof**:

**Invocar mediante un objeto cualquiera de sus subclases**

Permite que un método que asume como argumento un objeto de una superclase, pueda invocarse mediante un objeto cualquiera de sus subclases.

Debe tenerse cuidado con los métodos que se invocan mediante el argumento: sería incorrecto llamar a métodos no redefinidos de la subclase, a menos que se realizara el casting adecuado.

## Instanceof

El operador instanceof devuelve un valor booleano true si la variable referenciada del miembro de la izquierda es una instancia de la clase especificada en el miembro de la derecha. Si no, false. Se produce error al compilar si no existe relación entre la variable referenciada y la clase.

Sintaxis de instanceof:  
variable\_referenciada **instanceof** clase

Partiendo del siguiente código fuente:



```

class Animal{}

class Vaca extends Animal{}

class Toro extends Animal{}

class EjemploAnimales {
    public static void main(String args[]){
        Vaca vac=new Vaca();
        Toro tor=new Toro();
        Animal aniVaca=new Vaca();
        //Devuelven true y true
        System.out.println(vac instanceof Vaca);
        System.out.println(vac instanceof Animal);
        //Si se descomenta, no compila
        //System.out.println(vac instanceof Toro);
        System.out.println("-----");
        //Devuelven true, false y true
        System.out.println(aniVaca instanceof Vaca);
        System.out.println(aniVaca instanceof Toro);
        System.out.println(aniVaca instanceof Animal);
        //Si se descomenta, no compila
        //System.out.println(aniVaca instanceof Integer);
        System.out.println("FIN DE PROGRAMA");
    }
}

```

vamos a ver unos ejemplos:

### Ejemplo: Ejemplo 1

```

//Los tres elementos del array pasan el test "IS A" con Object
Object unArray[]={new Integer(1), new String("Hola"), new Double(12.5)}

```

el método del código de abajo de nombre unMetodo(..) puede invocarse mediante un objeto Toro, pero cuidado, porque mediante el argumento del método puede invocarse a métodos redefinidos de Toro, no a los propios, a menos que se realice el casting adecuado.

Para verlo, agregar al código de EjemploAnimales las siguientes líneas:

```

57         System.out.println("FIN DE PROGRAMA");
58         unMetodo(tor);
59     }
60     public static void unMetodo(Animal animal){
61         //Debe hacerse un casting para invocar a un método propio de Toro
62         //Si se intenta animal.torear(),
63         //error al compilar: el método torear no está definido en Animal
64         Toro torito=(Toro)animal;
65         torito.torear();
66
67         //Se accede al método comer() redefinido en Toro
68         animal.comer();
69     }
70 }

```

**Ejemplo: Ejemplo 2**

```
//Los tres elementos del array pasan el test "IS A" con Animal
Animal otroArray[]={new Animal(), new Vaca(), new Toro()}
```

el método unMetodo(..) puede ser invocado pasándole cualquier objeto ya que Object es la superclase de todas las clases Java

```
1 public class Polimorfismo{
2     public static void main(String args[]){
3         Polimorfismo pol=new Polimorfismo();
4
5         //Se invoca a un método cuyo argumento es un Object,
6         //pasándole una String
7         pol.unMetodo("CUCU");
8     }
9     public void unMetodo(Object obj){
10        String s=(String)obj;
11        System.out.println(s.length());
12
13        //Si se descomenta la línea 15, error al compilar.
14        //El método length() no está definido en Object
15        //System.out.println(obj.length());
16    }
17 }
```

**Palabra reservada super**

Se utiliza, cuando una clase hereda de otra, para hacer referencia desde la subclase a métodos o variables de instancia de la superclase en el caso de que haya coincidencia de nombres. Dos formas de uso:

**En una instrucción.**

La palabra super se puede utilizar en una instrucción, seguida del operador punto (.) para acceder a un método o variable de instancia de la superclase.

```

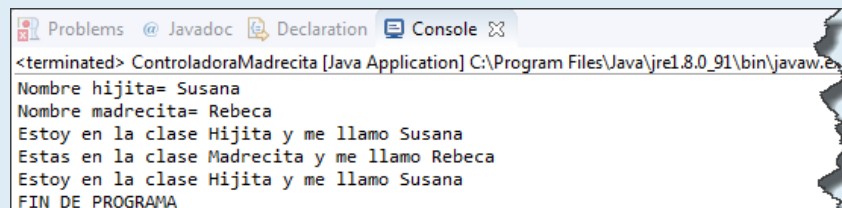
super.<nombreMetodo>;
super.<variableInstancia>;

class Madrecita{
    String nombre="Rebeca";
    void saludo(){
        System.out.println("Estas en la clase Madrecita y me llamo "+nombre);
    }
}

class Hijita extends Madrecita{
    //Variable de instancia de nombre coincidente con el de su superclase.
    String nombre="Susana";
    //Método de nombre coincidente con el de su superclase
    void saludo(){
        System.out.println("Estoy en la clase Hijita y me llamo "+nombre);
    }
    void muestraNombreHijita(){
        System.out.println("Nombre hijita= "+nombre);
        //Se muestra la variable de la superclase
        System.out.println("Nombre madrecita= "+super.nombre);
        saludo();
        //Se invoca el método de la superclase
        super.saludo();
    }
}

public class ControladoraMadrecita{
    public static void main(String args[]){
        Hijita hij=new Hijita();
        hij.muestraNombreHijita();
        hij.saludo();
        System.out.println("FIN DE PROGRAMA");
    }
}

```

**Por consola:**


```

<terminated> ControladoraMadrecita [Java Application] C:\Program Files\Java\jre1.8.0_91\bin\javaw.E
Nombre hijita= Susana
Nombre madrecita= Rebeca
Estoy en la clase Hijita y me llamo Susana
Estas en la clase Madrecita y me llamo Rebeca
Estoy en la clase Hijita y me llamo Susana
FIN DE PROGRAMA

```

**Pasando parámetros a un método**

La palabra super se puede utilizar como llamada a un método al que se le pasa parámetros.

El método que se ejecutará es, con mucha frecuencia, el constructor de la clase. Esta es la forma más habitual de uso.

```

super(titulo);

class SerHumano {
    String planetaHabitado;
    int coefInteligencia;
    //Constructor de la clase raíz de dos de las clases de este código
    SerHumano(String planetHabit,int cIntelig){
        planetaHabitado=planetHabit;
        coefInteligencia=cIntelig;
    }
    void comunicarseHumanos(){
        System.out.println("Un ser humano se comunica con otro mediante "+
            "una lengua comun");
    }
}

class Español extends SerHumano {
    String lengua;
    String pais;
    Español(String lengua,String pais){
        //Llamada al constructor de la superclase.
        super("Tierra",150);
        this.lengua=lengua;
        this.pais=pais;
    }
    void comunicarseEspañoles(){
        comunicarseHumanos();
        System.out.println("Un ser humano espa"+(char)164+"ol habita en la "+
            planetaHabitado+" y tiene un CI superior a "+coefInteligencia);
        System.out.println("Ademas vive en "+pais+" y habla el "+lengua);
    }
}

class Italiano extends SerHumano {
    String lengua;
    String pais;
    Italiano(String lengua,String pais){
        //Llamada al constructor de la superclase.
        super("Tierra",150);
        this.lengua=lengua;
        this.pais=pais;
    }
    void comunicarseItalianos(){
        comunicarseHumanos();
        System.out.println("Un ser humano italiano habita en la "+
            planetaHabitado+" y tiene un CI superior a "+coefInteligencia);
        System.out.println("Ademas vive en "+pais+" y habla el "+lengua);
    }
}

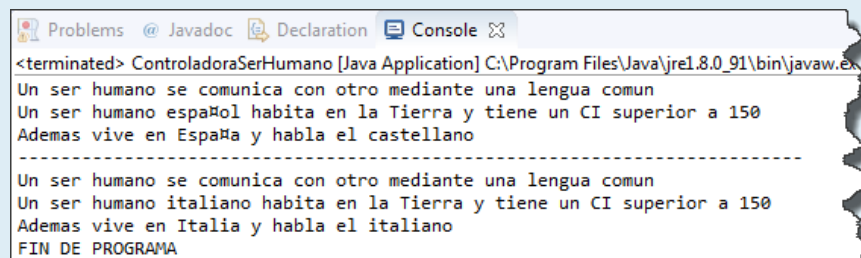
```

```

public class ControladoraSerHumano {
    public static void main(String args[]){
        Español esp=new Español("castellano","Espa"+(char)164+"a");
        esp.comunicarseEspañoles();
        System.out.print("-----");
        System.out.println("-----");
        Italiano ita=new Italiano("italiano","Italia");
        ita.comunicarseItalianos();
        System.out.println("FIN DE PROGRAMA");
    }
}

```

**Por consola:**



```

<terminated> ControladoraSerHumano [Java Application] C:\Program Files\Java\jre1.8.0_91\bin\javaw.exe
Un ser humano se comunica con otro mediante una lengua comun
Un ser humano español habita en la Tierra y tiene un CI superior a 150
Ademas vive en España y habla el castellano
-----
Un ser humano se comunica con otro mediante una lengua comun
Un ser humano italiano habita en la Tierra y tiene un CI superior a 150
Ademas vive en Italia y habla el italiano
FIN DE PROGRAMA

```

**super** no se admite dentro de métodos estáticos, se produce un error de compilación.

Ahora conviene hacer los ejercicios cuarto y quinto del tema.

## Interfaces

Concepto general de interface: **conjunto de elementos que permiten la relación entre dos sistemas de distinta naturaleza.**

En informática, cuando se habla de interfaces habitualmente uno se refiere a interfaces gráficas de usuario, es decir, al conjunto de elementos que permite la interacción "amigable" del usuario con el ordenador, generalmente a través de entornos gráficos, que cuentan con iconos, ventanas, botones, cuadros de texto, listas desplegables, etc. Java dispone de varios paquetes que contienen clases para crear y gestionar este tipo de interfaces gráficas. Se verán más adelante.

En este tema no se van a tratar las interfaces gráficas sino las interfaces en general. **Una interface**, en este sentido, **se puede considerar como un almacén de provisiones (variables y métodos) para la clase que la implementa**. El programador cuando implementa una o varias interfaces en una clase Java está adquiriendo un compromiso de utilización de esas provisiones basado en unas reglas de obligado cumplimiento. Al final del tema hay un ejemplo que ayudará a clarificar todo lo dicho anteriormente.

Durante el curso, las interfaces se emplearán muy a menudo cuando se trabaje con hilos y eventos.

El concepto de interface es mucho más profundo de lo que aquí se ha explicado, pero dada las características del curso no se amplía más. Cabe decir, por ejemplo, que las interfaces son una parte fundamental en la programación de EJBs (Enterprise JavaBeans). Un EJB es un componente software residente en un servidor de aplicación, destinado al desarrollo y despliegue de aplicaciones empresariales bajo la especificación EJB definida dentro de la plataforma J2EE.

## Contenido de una interface

Una interface sólo puede componerse de:

- **Variables de instancia públicas, estáticas y finales.** Todas las clases que implementen interfaces tendrán acceso a dichas variables sin más que escribir el nombre de la interface seguido del operador punto (.) y el nombre de la variable. En este sentido, las interfaces pueden considerarse como almacenes de constantes a disposición de las clases que las implementen.
- **Métodos sin código implementado.** Todas las clases que implementen interfaces deberán llenar de código todos métodos declarados por la interfaces y asignarles el modificador **public**. En este sentido, las interfaces se comportan como contratos por los que las clases que las implementan se comprometen a llenar de código (puede ser vacío { }) cada método declarado por las interfaces implementadas. Si no se cumple esto, error de compilación.

Las variables de instancia de una interface son, implícitamente, públicas, estáticas y finales. Los correspondientes modificadores no necesitan especificarse en el código, aunque suele hacerse. Respecto a los métodos, implícitamente, se consideran públicos y abstractos.

En las interfaces pueden aplicarse relaciones de herencia como en las clases, pero con una gran diferencia: cuando se trabaja con interfaces se admite la herencia múltiple. Una interface puede tener varias superinterfaces del mismo nivel jerárquico.

## Implementación de interfaces en una clase

Se utiliza la palabra clave **"implements"**:

```
public class miClase implements interface1, interface2, ... {
    código
}
```

La API cuenta con numerosas interfaces que muchas de sus clases **implementan por construcción**. Cuando esto ocurre, las instancias de esas clases pueden usar los métodos declarados por dichas interfaces directamente. Son interfaces que contienen métodos con código implementado a disposición del programador. Para saber si una clase implementa por construcción una interface se consulta la API.

Esta es la gran diferencia entre clases que **implementan explícitamente** interfaces **mediante** la palabra clave **"implements"** y clases de la API que, por construcción, implementan interfaces.

**Por ejemplo, la clase java.awt.Frame implementa por construcción la interface MenuContainer** que declara tres métodos (ir a la API para comprobarlo). Pues bien, estos métodos cuando se emplee la clase Frame no tienen por qué ser usados; el programador los utilizará si le conviene. En la API se puede ver que esta interface también la implementan por construcción las clases Component, MenuBar y Menu.

**Otro ejemplo: la clase java.util.Vector implementa por construcción cinco interfaces:** List, RandomAccess, Collection, Cloneable y Serializable. La interface List declara un montón de métodos que pueden usarse cuando se trabaje con un objeto de Vector. La interface Serializable es la interface madre de multitud de subinterfaces que contienen multitud de métodos.

## Creación de interfaces propias

El proceso es similar al de una clase:

```
<modificador acceso> interface <nombreInterface> {
    código
}
```

Debe tenerse en cuenta lo siguiente:

- **Los modificadores de acceso admisibles en las interfaces son public y sin modificador.** Además, implícitamente, toda interface lleva el modificador **abstract**
- **En un mismo código fuente puede haber varias interfaces.** Ocurre como con las clases: sólo se admite una interface con modificador public en una mismo código fuente. Además, el nombre del código fuente debe coincidir con el de la interface (si no, error de compilación)
- **En un mismo código fuente pueden coexistir interfaces y clases que implementen dichas interfaces.** Ocurre que sólo se admite un modificador public, sin importar que se asigne a una clase o a alguna interface. Ahora bien, el nombre del código fuente deberá coincidir con el de la interface o clase que tenga el modificador public (si no, error de compilación).

**Va a crearse una interface que declara una constante y un método sin cuerpo.**

```
public interface Poligono2D {
    public static final double CONSTANTE= Math.PI;
    double calcularArea();
}
```

Se guardará en un código fuente de nombre coincidente al de la interface, ya que es pública. Si no, error de compilación.

Luego se compilará para obtener su class asociado.

Ahora se crearán dos clases que implementen la interface anterior. **Esto define una pauta de programación en el sentido de que todas las clases que la implementen deben tener el método calcularArea. Cada una lo llenará de código de modo distinto, pero todas deberán tenerlo.** Es un modo de indicarles qué deben implementar (métodos declarados en la interface) sin especificar el cómo (código de esos métodos: cada clase lo hará de forma distinta).

**Los códigos fuentes se guardarán en el mismo directorio que la interface:**

```
public class Cuadrado implements Poligono2D{
    double x1,y1,x2,y2;
    Cuadrado(double x1, double y1, double x2, double y2){
        this.x1=x1;
        this.y1=y1;
        this.x2=x2;
        this.y2=y2;
    }
    public double calcularArea(){
        return Math.abs((x2-x1)*(y2-y1));
    }
    public static void main(String args[]){
        Cuadrado cua=new Cuadrado(1,2,3,4);
        System.out.println("Area="+cua.calcularArea());
        System.out.println("FIN DE PROGRAMA");
    }
}
```

Al ejecutar la clase:

**Por consola:**



```
<terminated> Cuadrado [Java Application] C:\Program Files\Java\jre1.8.0_101\bin\javaw.exe
Area=4.0
FIN DE PROGRAMA
```

### Código de la otra clase que implementa Poligono2D:

```
public class Circulo implements Poligono2D{
    double radio;
    Circulo(double radio){
        this.radio=radio;
    }
    public double calcularArea(){
        return Poligono2D.CONSTANTE*Math.pow(radio,2);
    }
    public static void main(String args[]){
        Circulo cir=new Circulo(2);
        System.out.println("Area="+cir.calcularArea());
        System.out.println("FIN DE PROGRAMA");
    }
}
```

**Por consola:**

```
Console Problems Javadoc Declaration Search Debug
<terminated> Circulo [Java Application] C:\Program Files\Java\jre1.8.0_101\bin\javaw.exe
Area=12.566370614359172
FIN DE PROGRAMA
```

Se podrían ir definiendo más clases que modelaran a polígonos que implementaran Poligono2D, como Rectangulo, Trapecio, Pentagono, etc. Todas deberían llenar de código adecuado el método calcularArea().

Se estaría creando una especie de familia de clases con una propiedad común a todas ellas: el área de los polígonos. Una forma elegante de conseguir esto es el empleo de interfaces.

## Interfaces muy usadas que forman parte de la API

Unas interfaces que se emplean con mucha frecuencia **son las asociadas a la gestión de eventos y a la de hilos**. Para hacerse una idea de las mismas ir a la API. Para las primeras, se va al paquete java.awt.event y para la segunda a java.lang.

para gestionar un evento de ratón, se emplea la **interface java.awt.event.MouseListener** que declara los siguientes métodos:

```
public void mouseClicked(MouseEvent e)
public void mousePressed(MouseEvent e)
public void mouseReleased(MouseEvent e)
public void mouseEntered(MouseEvent e)
public void mouseExited(MouseEvent e)
```

En función de los eventos que se van a gestionar, unos métodos se llenarán de código no vacío y otros de código vacío ({}).

Para gestionar un evento de ventana, se implementa la **interface java.awt.event.WindowListener** que declara los siguientes métodos:

```
public void windowActivated(WindowEvent e)
public void windowClosed(WindowEvent e)
public void windowClosing(WindowEvent e)
public void windowDeactivated(WindowEvent e)
public void windowDeiconified(WindowEvent e)
public void windowIconified(WindowEvent e)
public void windowOpened(WindowEvent e)
```

Para gestionar hilos se emplea la **interface java.lang Runnable** que declara sólo un método:

```
public void run()
```

**Ejemplo final IMPORTANTE:** para finalizar el tema y repasar la herencia, las palabras reservadas this y super y la implementación de interfaces, se va a mostrar un código que despliega una ventana o frame cerrable.

Se van a utilizar métodos relacionados con propiedades de la ventana no estudiadas todavía, que permiten asignar color de relleno a la ventana, posicionarla, dimensionarla, hacerla visible, agregarle escuchadores de eventos, etc. Todo esto se estudiará más adelante. Por el momento se explican de forma somera.

- **Color de relleno:**

Método empleado: `void setBackground(Color c)` Clase: `java.awt.Component`

- **Punto de referencia para el despliegue:** esquina superior izquierda.

Método empleado: `void setLocation(int x,int y)` Clase: `java.awt.Component`

- **Dimensiones:**

Método empleado: `void setSize(int anchura,int a)` Clase: `java.awt.Component`

- **Visibilidad:**

Método empleado: `void setVisible(boolean b)` Clase: `java.awt.Component`

Se va a crear una clase que hereda de `Frame` y que implementa la interface `WindowListener`. Lo primero se hace para construir la ventana y lo segundo para poder cerrarla. El cierre de la ventana está asociado al evento "pulsar el aspa".

Para que una `Frame` pueda escuchar eventos provocados por el usuario, es necesario agregarle un escuchador de eventos o unas "orejas" que le permitan detectarlo. Esto se hace mediante el método de `java.awt.Window` "`void addWindowListener(WindowListener wl)`". Su argumento tiene que ser una clase que implemente la interface `WindowListener`.

**Frame** es una clase que pertenece al paquete `java.awt`. Consultar la API para echar un vistazo a sus constructores y a todos los métodos que se pueden utilizar. Son muchísimos dada su estructura jerárquica de clases.

**WindowListener** es una interface que pertenece a `java.awt.event`. Consultar la API y comprobar los métodos que declara.

```
import java.awt.*;
import java.awt.event.*;

public class MiVentana extends Frame implements WindowListener{

    MiVentana(String titulo){

        //Invocar el constructor de Frame que tiene como argumento una String
        super(titulo);

        //Asignar al objeto en curso, o sea a una Frame, color de fondo
        //Es equivalente a this.setBackground(Color.cyan)
        setBackground(Color.cyan);
    }
}
```

```

    /*
    * Agregar al objeto en curso un escuchador de eventos. En este caso,
    * se emplea para el cierre de la ventana. El argumento del método
    * tiene que ser un objeto WindowListener, es decir, una clase que
    * implemente la interface WindowListener. Como la clase en curso
    * la implementa, se emplea this.
    */
    addWindowListener(this);
}

/*
* Se llenan de código TODOS los métodos que declara la interface
* implementada. Si sólo se desea gestionar el cierre de la ventana
* se llenará de código el método windowClosing(..) con la línea
* System.exit(0). Esto provoca el final de la ejecución de la máquina
* virtual. Consultar este método estático de la clase System en la API.
* El resto de métodos se llena de código vacío {}.
*/
public void windowActivated(WindowEvent e){}
public void windowClosed(WindowEvent e){}
public void windowClosing(WindowEvent e){
    System.out.println("Adios");
    System.exit(0);
}
public void windowDeactivated(WindowEvent e){}
public void windowDeiconified(WindowEvent e){}
public void windowIconified(WindowEvent e){}
public void windowOpened(WindowEvent e){}

public static void main(String args[]){

    //Invocar al constructor de la clase pasándole lo que está esperando
    MiVentana mv=new MiVentana("Mi primera ventana");

    //Situación la ventana
    mv.setLocation(100,40);

    //Dimensionar la ventana
    mv.setSize(250,250);

    //Hacer visible la ventana
    mv.setVisible(true);
}
}

```

# Ejercicios

## Ejercicio 1. Línea de comandos

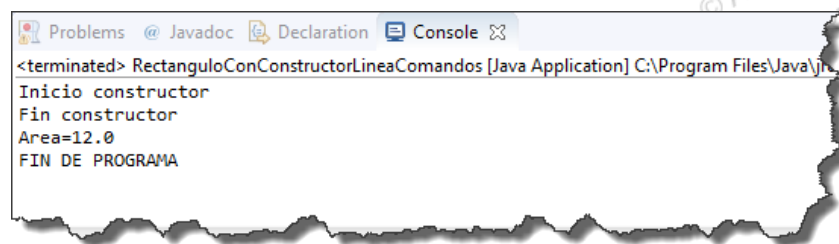
20

Crear una clase pública de nombre **RectanguloConConstructorLineaComandos** con código similar a la clase **RectanguloConConstructor**, pero teniendo en cuenta que **las coordenadas de los puntos de las esquinas del rectángulo se introducirán a través de la línea de comandos**.

## Recomendaciones

Todos los códigos en `c:\cursojava\tema6` o en `eclipse_home\MyProjects\tema6` si se emplea Eclipse

## Datos a mostrar por consola



```
<terminated> RectanguloConConstructorLineaComandos [Java Application] C:\Program Files\Java\jre...
Inicio constructor
Fin constructor
Area=12.0
FIN DE PROGRAMA
```

## Ejercicio 2. Constructor

30

Crear un programa Java con una clase pública llamada **Alumno** con la siguiente estructura de métodos:

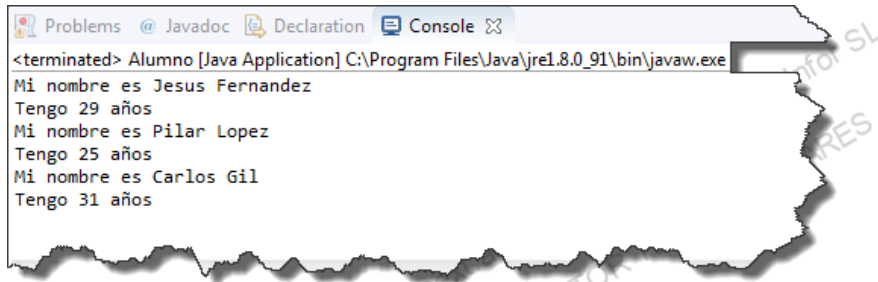
- Un constructor con tres argumentos: dos String y una variable int. La primera String almacenará el nombre, la segunda los apellidos y la variable int la edad.
- Un método sin argumentos llamado `mostrarNosotros` que devuelva void y que muestre por consola el nombre completo y la edad.

## Pistas

El método `main` se encargará de crear e inicializar **tres objetos de la clase** mediante el constructor, al que se le pasará el **nombre, apellidos y edad** de dos compañeros, además de los vuestros.

La clase contará con **tres variables de instancia** de nombres y tipos coincidentes con los argumentos del constructor.

## Datos a mostrar por consola



```
<terminated> Alumno [Java Application] C:\Program Files\Java\jre1.8.0_91\bin\javaw.exe
Mi nombre es Jesus Fernandez
Tengo 29 años
Mi nombre es Pilar Lopez
Tengo 25 años
Mi nombre es Carlos Gil
Tengo 31 años
```

## Ejercicio 3. Instancias

20

Partiendo del código de la clase **MiCompañero**, completar las clases **Calculos** y **Controladora1** en base a las indicaciones de los comentarios incluidos en el esqueleto aportado:

## Lo necesario para comenzar

```

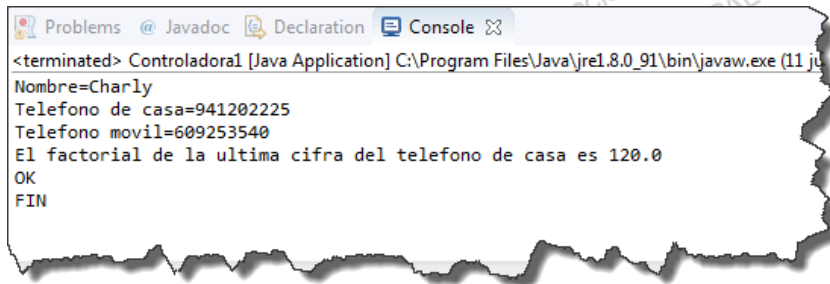
class MiCompañero {
    String nombre="Charly";
    String tefCasa="941202225";
    String tefMovil="609253540";
    double factorial=1;
    void muestraTelefonos(){
        System.out.println("Nombre="+nombre);
        System.out.println("Telefono de casa="+tefCasa);
        System.out.println("Telefono movil="+tefMovil);
    }
    /* A este método se le debe pasar la variable de instancia tefCasaUltimo
    * de Calculos y debe devolver el factorial de ese entero.
    */
    double calculaFactorial(int numero){
        for(int i=1;i<=numero;i++){
            factorial*=i;
        }
        return factorial;
    }
}

class Calculos extends MiCompañero {
    int edad=2;
    int tefCasaUltimo= // almacena la última cifra de tefCasa de MiCompañero
    void calculito(double fact){
        /*
        * El método mostrará por consola OK si el cociente
        * entre fact y edad es mayor de 15 y KO si es menor
        * o igual. Al método se le pasa el factorial obtenido en
        * el método calculaFactorial
        */
    }
}

public class Controladora1 {
    public static void main(String args[]){
        /*
        * Se encarga del control de la ejecución creando los objetos
        * adecuados y llamando a los métodos y variables pertinentes
        */
    }
}

```

## Datos a mostrar por consola



```
<terminated> Controladora1 [Java Application] C:\Program Files\Java\jre1.8.0_91\bin\javaw.exe (11 j...
Nombre=Charly
Telefono de casa=941202225
Telefono movil=609253540
El factorial de la ultima cifra del telefono de casa es 120.0
OK
FIN
```

## Ejercicio 4. Instancias

20

En este ejercicio debemos anticipar cuál es el resultado que se mostrará por consola al ejecutar el siguiente código:

### Lo necesario para comenzar



```

class Padre{

    String nombre = "PEPE";
    void saludo(){
        System.out.println("Estas en la clase Padre y me llamo " + nombre);
    }
}

class Hijo extends Padre{
    String nombre = "Carlos";
    void saludo(){
        System.out.println("Estas en la clase Hijo y me llamo " + nombre);
    }
    void muestraInformacion(){
        System.out.println("Nombre hijo=" + nombre);
        System.out.println("Nombre padre=" + super.nombre);
        saludo();
        super.saludo();
    }
}

public class ControladoraPadre {
    public static void main(String[] args) {
        Hijo hijo = new Hijo();
        hijo.muestraInformacion();
        System.out.println("-----");
        hijo.saludo();
        System.out.println("FIN DE PROGRAMA");
    }
}

```

## Ejercicio 5. Herencia

20

En este ejercicio debemos anticipar cuál es el resultado que se mostrará por consola al ejecutar el siguiente código:

### Lo necesario para comenzar

```

class MiJefe{

    String nombre = "Pepito";
    String telefonoCasa = "123456";
    String telefonoMovil = "987654";
    void muestraTelefonos(){
        System.out.println("Nombre= " + nombre);
        System.out.println("Telefono de casa= " + telefonoCasa);
        System.out.println("Telefono movil= " + telefonoMovil);
    }
}

public class EjercicioHerencia extends MiJefe{
    public static void main (String args[]){
        EjercicioHerencia eh = new EjercicioHerencia();
        eh.muestraTelefonos();
    }
    void muestraTelefonos(){
        System.out.println("Los datos de mi jefe:");
        super.muestraTelefonos();
        String miNombre = "Jesus";
        String miTelefonoCasa = "941200639";
        String miTelefonoMovil = "678905582";
        System.out.println("Mis datos:");
        System.out.println("Nombre= " + miNombre);
        System.out.println("Telefono de casa= " + miTelefonoCasa);
        System.out.println("Telefono movil= " + miTelefonoMovil);
        System.out.println("FIN DE PROGRAMA");
    }
}

```

## Recursos

### Enlaces de Interés



<http://docs.oracle.com/javase/tutorial/java/javaOO/index.html>

<http://docs.oracle.com/javase/tutorial/java/javaOO/index.html>

*Herencia (inheritance en inglés)*