



Herencia

# Índice



## Herencia

1   Concepto de herencia en Java	3
2   Constructores y herencia, la palabra super	5
3   Sobrescribir métodos	7
4   Métodos y datos protegidos en la herencia	9
5   Clases abstractas	12
6   Polimorfismo	14
7   La utilización de final	15
8   La clase Object	16

# 1. Concepto de herencia en Java

En Java la herencia se implementa con la palabra reservada **extends**, quizás la elección de esta palabra fue con el propósito de enfatizar que las subclases (clases que heredan) son una extensión u especialización de las superclases (clases de las que se hereda). Las superclases son una generalización y las subclases una especialización.

Una subclase sólo puede tener un superclase y una superclase puede tener varias subclases, porque Java sólo implementa la **herencia simple**.

La herencia es la implementación de la jerarquía de clases.

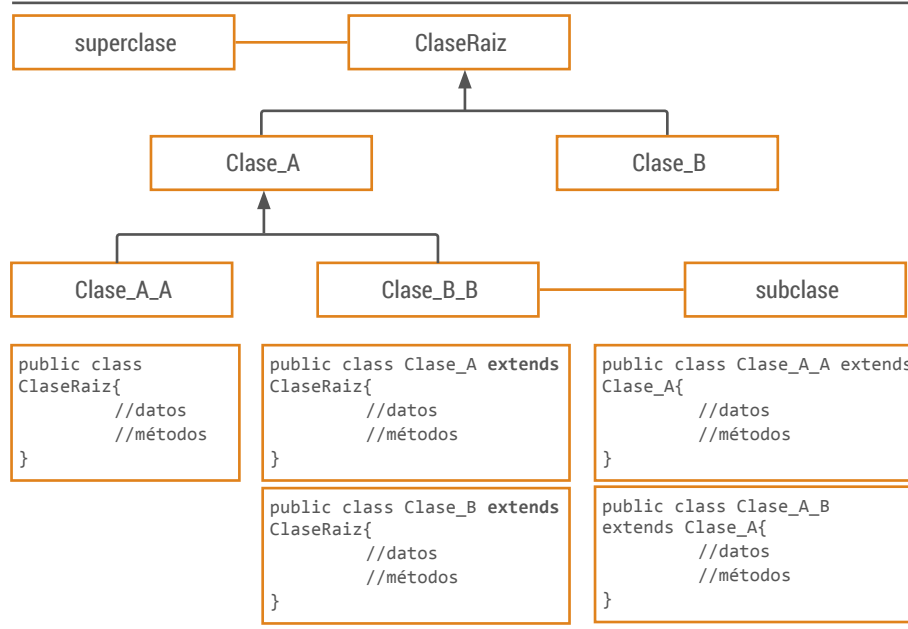


FIGURA 6.1: JERARQUÍA DE CLASES, HERENCIA EN JAVA

En Java la clase raíz de todas las clases es la clase **Object** y cuando una clase no hereda explícitamente de ninguna otra, lo está haciendo de **Object**.

<https://docs.oracle.com/javase/8/docs/api/>

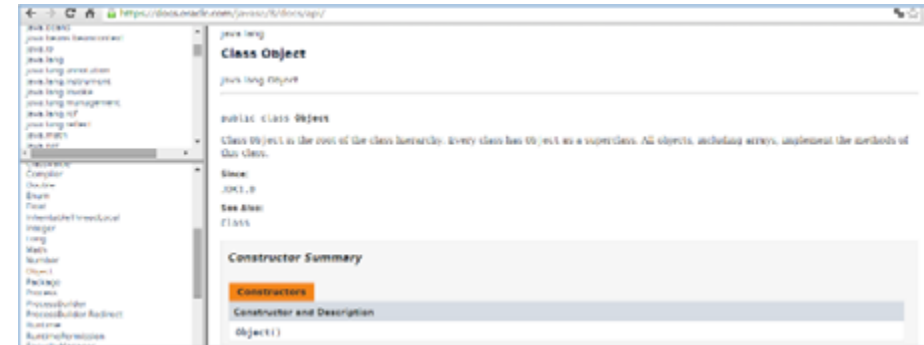


FIGURA 6.2: OBJECT EN API JDK 8

Cuando se instala JDK 8, se pueden utilizar todas las clases de la jerarquía de clases JSE 8, cuya documentación se puede consultar en:

Las subclases heredan las variables miembro (datos) de la superclase, respetando el tipo de acceso. Por tanto un objeto instanciado de una subclase ocupa el espacio de un objeto de la superclase más el espacio específico de las variables miembro propias.

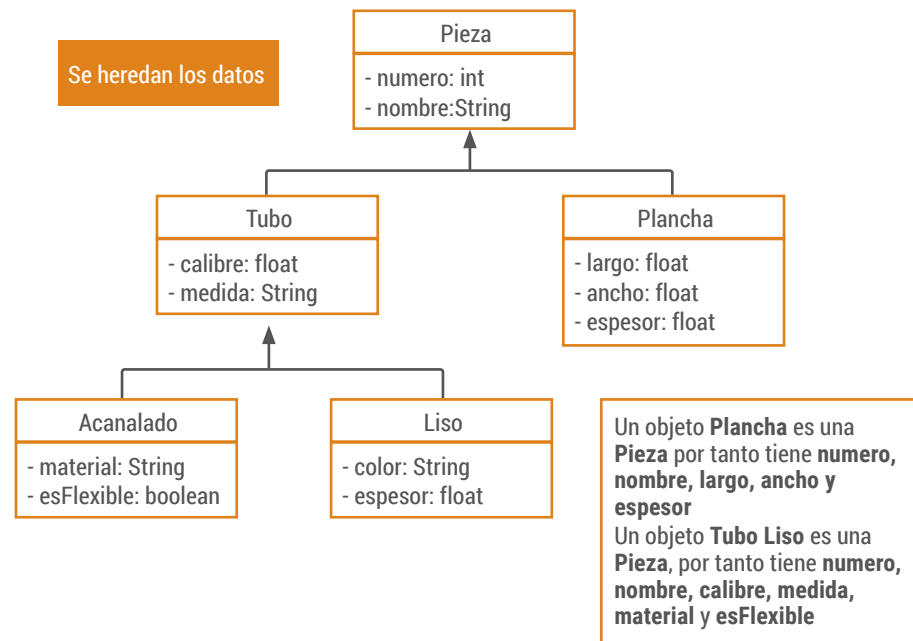


FIGURA 6.3: SE HEREDAN LOS DATOS.

Las subclases heredan las funciones miembro (métodos) de la superclase, respetando el tipo de acceso. Por tanto a un objeto instanciado de una subclase se le podrán invocar las funciones de la parte pública de su propia clase y las de de la superclase.

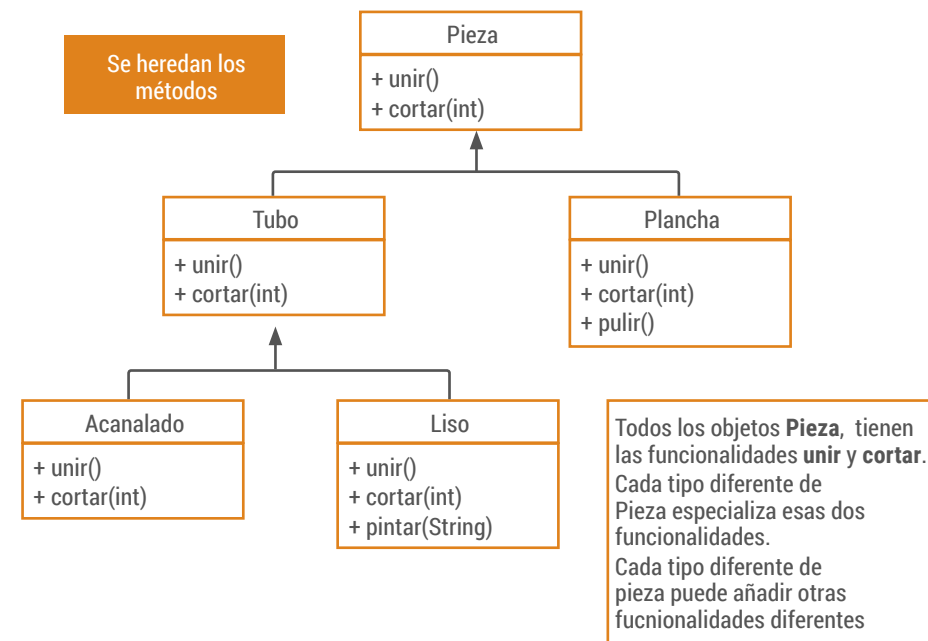


FIGURA 6.4: SE HEREDAN LOS MÉTODOS.

## 2. Constructores y herencia, la palabra super

Cuando se crea un objeto de una subclase, el objeto se tiene que construir como un objeto de la superclase. Esto se implementa utilizando la palabra `super` dentro de los constructores de las subclases invocando a uno de los constructores de la superclase. Por tanto el orden de construcción de los objetos de subclases es primero el constructor de la superclase, después el de la subclase. En la jerarquía de las imágenes anteriores las clases con sus datos y un constructor por cada una de ellas se muestra en el código siguiente:

```
public class Pieza {  
    private int numero;  
    private String nombre;  
  
    public Pieza(int numero, String nombre) {  
        super();           //constructor de  
Object, no hace falta  
        this.numero = numero;  
        this.nombre = nombre;  
    }  
}
```

```
public class Plancha extends Pieza {  
    private float largo;  
    private float ancho;  
    private float espesor;  
    public Plancha(int numero, String nombre,  
float  
largo, float ancho,  
float espesor) {  
        super(numero, nombre); //constructor de  
Pieza  
        this.largo = largo;  
        this.ancho = ancho;  
        this.espesor = espesor;  
    }  
}  
public class Tubo extends Pieza {  
    private float calibre;  
    private String medida;  
    public Tubo(int numero, String nombre,
```

```

                                float calibre,
String medida) {
    super(numero, nombre); //constructor de
Pieza
    this.calibre = calibre;
    this.medida = medida;
}
}

```

La forma de invocar a **super** tiene que estar de acuerdo con la forma de uno de los constructores de la superclase. A continuación también se muestran los códigos de las subclases que heredan de una de las subclases, que para estas dos es su superclase.

```

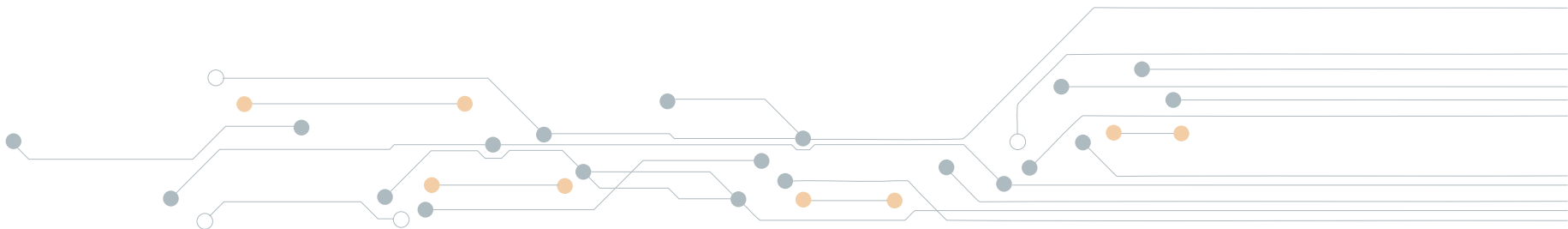
public class Acanalado extends Tubo {
    private String material;
    private boolean esFlexible;
    public Acanalado(int numero, String
nombre,
                                float
calibre, String medida,

```

```

String material, boolean esFlexible) {
    super(numero, nombre, calibre, medida);
    this.material = material;
    this.esFlexible = esFlexible;
}
}
public class Liso extends Tubo {
    private String color;
    private float espesor;
    public Liso(int numero, String nombre,
                                float calibre,
String medida,
                                String color, float
espesor) {
    super(numero, nombre, calibre, medida)
    this.color = color;
    this.espesor = espesor;
}
}

```



Las funciones también se heredan, esto quiere decir que a cualquier objeto instanciado de una subclase se le pondrá invocar una de las funcionalidades definidas en la superclase.

En las clases del ejemplo se ha definido la función **toString** en la superclase de todas las demás clases. Se crean objetos de cada una de las clases y se invoca la función **toString**, ejecutándose el código de esta función definido en la superclase.

```
public static void main(String[] args) {
    Pieza pieza = new Pieza(25, "Arandela centrifugadora");
    Tubo tubo = new Tubo(101, "Especial", 1.0f, "pulgadas");
    Plancha plancha = new Plancha(201, "Extra", 2.1f, 3.5f, 0.2f );
    Acanalado acanalado = new Acanalado (102, "Humos", 30.5f, "mms", "plastico", true);
    Liso liso = new Liso(103, "Canal", 1.0f, "pulgadas");

    System.out.println(pieza.toString());
    System.out.println(tubo.toString());
    System.out.println(plancha.toString());
    System.out.println(acanalado.toString());
    System.out.println(liso.toString());
}

public class Pieza {
    //datos
    //constructor
    public String toString() {
        return "Pieza [numero=" + numero + ", nombre=" +
            nombre + ", clase=" + this.getClass() + "]";
    }
}
```

```
C:\Juan\JavaTelefonica\PiezasHerencia\bin>java -cp . com.juan.piezas.PruebaConstructores
Pieza [numero=25, nombre=Arandela centrifugadora, clase=class com.juan.piezas.Pieza]
Pieza [numero=101, nombre=Especial, clase=class com.juan.piezas.Tubo]
Pieza [numero=201, nombre=Extra, clase=class com.juan.piezas.Plancha]
Pieza [numero=102, nombre=Humos, clase=class com.juan.piezas.Acanalado]
Pieza [numero=103, nombre=Canal, clase=class com.juan.piezas.Liso]
```

FIGURA 6.5: EJECUCIÓN DE MÉTODO HEREDADO.

### 3. Sobrescribir métodos

Cuando se necesita que los objetos de las subclases tengan funcionalidades especializadas y distintas de las heredadas, se definen esas funciones en dichas subclases. Este mecanismo se denomina "**override**", **sobrescribir** el funcionamiento del código heredado de los métodos de la superclase.

Para que exista "**override**" las funciones en la superclase y en las subclases tienen que tener exactamente la misma forma: **mismo nombre**, **mismo valor devuelto** y **misma lista de parámetros**.

Dentro de la función sobrepasada en la subclase se invoca a esa función o cualquier otra de la superclase con la palabra **super**. Un ejemplo de “**override**” de métodos es el código siguiente:

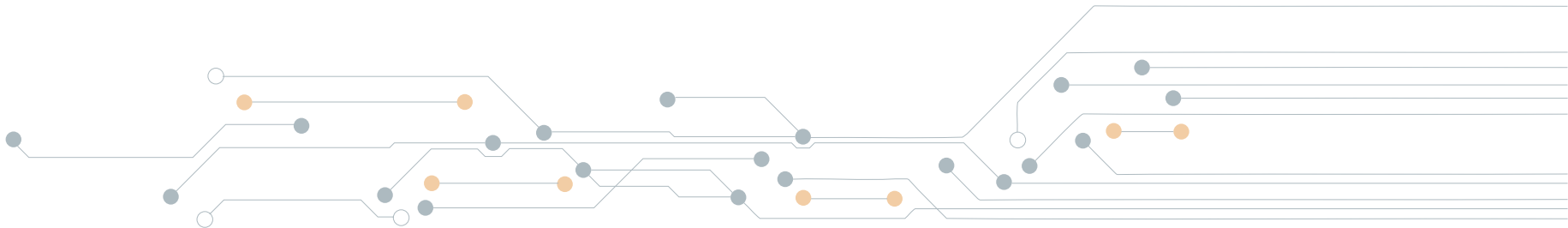
```
public class Plancha extends Pieza {
    private float largo;
    private float ancho;
    private float espesor;
    //constructor
    @Override
    public String toString() {
        return "Plancha [largo=" + largo + ",
ancho=" + ancho
        + ", espesor=" + espesor + ", " + super.
toString()+ "];"
    }
}

public class Tubo extends Pieza {
    private float calibre;
    private String medida;
```

```
//Constructor
@Override
public String toString() {
    return "Tubo [calibre=" + calibre + ",
medida="
    + medida + ", " + super.toString() +
    "];"
}

public class Acanalado extends Tubo {
    private String material;
    private boolean esFlexible;

    //Constructor
    @Override
    public String toString() {
        return "Acanalado [material=" + material
        + ", esFlexible=" + esFlexible + ", " +
super.toString() + "];"
    }
}
```



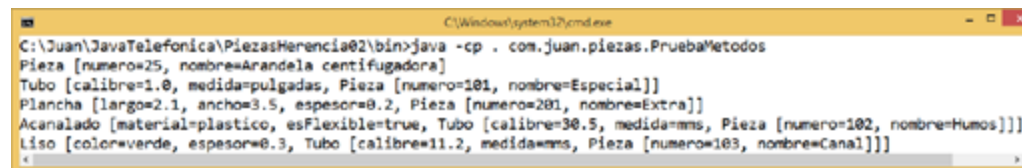


Un ejemplo de utilización de la función toString sobrepasada es el que se muestra en la figura siguiente.

```
public static void main(String[] args) {
    Pieza pieza = new Pieza(25, "Arandela centrifugadora");
    Tubo tubo = new Tubo(101, "Especial", 1.0f, "pulgadas");
    Plancha plancha = new Plancha(201, "Extra", 2.1f, 3.5f, 0.2f);
    Acanalado acanalado = new Acanalado(102, "Humos", 11.2f, 0.3f, "verde");
    Liso liso = new Liso(103, "Canal", 11.2f, 0.3f, "verde");

    System.out.println("Pieza: " + pieza.toString());
    System.out.println("Tubo: " + tubo.toString());
    System.out.println("Plancha: " + plancha.toString());
    System.out.println("Acanalado: " + acanalado.toString());
    System.out.println("Liso: " + liso.toString());
}

public class Acanalado extends Tubo {
    //datos
    //constructor
    @Override
    public String toString() {
        return "Acanalado [material=" + material + ", esFlexible=" + esFlexible + ", " + super.toString() + "];";
    }
}
```



```
C:\Windows\system32\cmd.exe
C:\Juan\JavaTelefonica\PiezasHerencia02\bin>java -cp . com.juan.piezas.PruebaMetodos
Pieza [numero=25, nombre=Arandela centrifugadora]
Tubo [calibre=1.0, medida=pulgadas, Pieza [numero=101, nombre=Especial]]
Plancha [largo=2.1, ancho=3.5, espesor=0.2, Pieza [numero=201, nombre=Extra]]
Acanalado [material=plastico, esFlexible=true, Tubo [calibre=30.5, medida=mm, Pieza [numero=102, nombre=Humos]]]
Liso [color=verde, espesor=0.3, Tubo [calibre=11.2, medida=mm, Pieza [numero=103, nombre=Canal]]]
```

FIGURA 6.6: EJECUCIÓN DE MÉTODO SOBREPASADO.

## 4. Métodos y datos protegidos en la herencia

Para poder acceder a las variables y funciones miembro heredados de la superclase estos no pueden estar en el ámbito private. Si se modifica su acceso a ámbito public o de paquete, en cierta manera se estaría incumpliendo el principio de la ocultación y protección de las encapsulación.

Para mantener ese principio se implementa el modificador de acceso **protected**. Cuando un miembro de una superclase tiene ese modificador puede ser accedido desde las subclases que la hereden. en estas subclases ese miembro es como si estuviera en ámbito **private** para las que hereden de ellas.

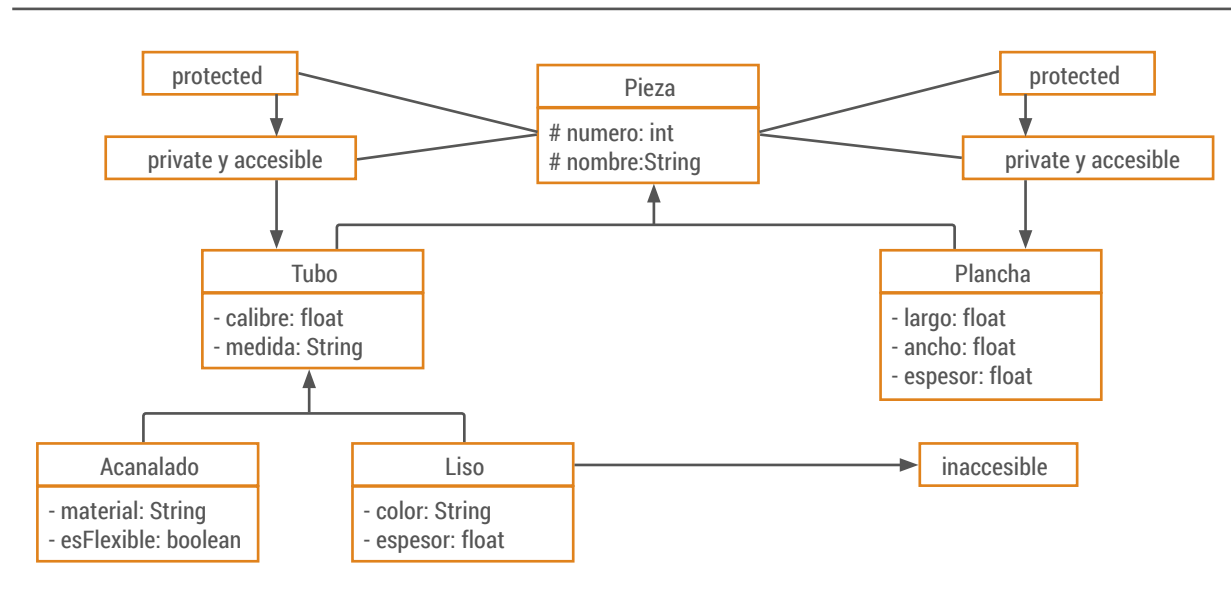


FIGURA 6.7: DATOS PROTECTED.

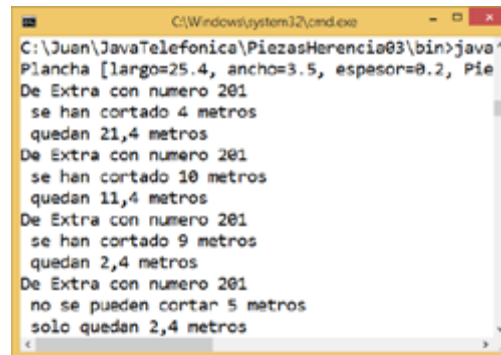


En la imagen se muestra una función de una subclase que accede a datos protected de la superclase.

Si no se utiliza el modificador **protected**, la superclase tendría que proporcionar métodos **public** de tipo "getter" para obtener el valor de los datos **private**.

```
public static void main(String[] args) {
    Plancha plancha = new Plancha(201,"Extra", 25.4f, 3.5f);
    Random aleatorio = new Random();
    int metros=0;
    System.out.println(plancha);
    do{
        metros = aleatorio.nextInt(10)+1;
    }while(plancha.cortar(metros));
}
```

Si no se utiliza el modificador **protected**, la superclase tendría que proporcionar métodos **public** de tipo "getter" para obtener el valor de los datos **private**.



```
C:\Windows\system32\cmd.exe
C:\Juan\JavaTelefonica\PiezasHerencia03\bin>java Plancha [largo=25.4, ancho=3.5, espesor=0.2, Pie
De Extra con numero 201
se han cortado 4 metros
quedan 21,4 metros
De Extra con numero 201
se han cortado 10 metros
quedan 11,4 metros
De Extra con numero 201
se han cortado 9 metros
quedan 2,4 metros
De Extra con numero 201
no se pueden cortar 5 metros
solo quedan 2,4 metros
```

```
public class Plancha extends Pieza{
    //datos
    //constructor
    //otras funciones
    public boolean cortar(int metros){
        DecimalFormat df = new
        DecimalFormat("#,###.##");
        System.out.println("De "+ nombre +
        " con numero " + numero);
        if (largo < metros){
            System.out.println(" no se
            pueden cortar "+ metros + " metros ");
            System.out.println(" solo
            quedan "+ df.format(largo) + " metros ");
            return false;
        }
        largo -= metros;
        System.out.println(" se han
        cortado "+ metros + " metros ");
        System.out.println(" quedan "+
        df.format(largo) + " metros ");
        return true;
    }
}
```

FIGURA 6.8: SUBCLASE ACCEDIE A DATOS PROTECTED.

## 5. Clases abstractas

Si de una clase **nunca se van a instanciar objetos**, se denomina **abstracta**. Esto significa que dicha clase define una representación (datos) y un comportamiento (métodos) comunes a las clases que hereden de ella, los objetos serán instanciados de dichas clases, las cuales podrán especializar las funcionalidades, sobrepasando los métodos y podrán añadir más atributos incorporando más variables miembro.

En java las clases abstractas se implementan utilizando el modificador **abstract**.

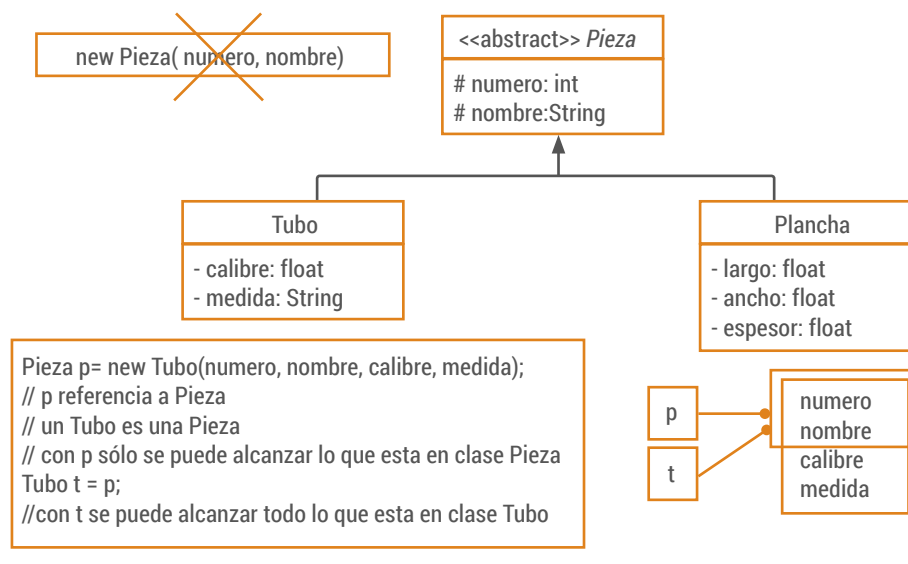


FIGURA 6.9: CLASE ABSTRACTA.

En la jerarquía de clases utilizado la clase **Pieza** es **abstracta**, nunca habrá objetos que sean sólo Pieza, serán siempre objetos instanciados de las clases Plancha o Tubo o Acanalado o Fijo. Si se califica **Pieza** como **abstract** esto implica que no se pueden crear objetos de esta clase, pero hay que tener presente que cualquier objeto de las subclasses son objetos de tipo Pieza.

También se pueden definir métodos abstractos, esto implica que no se implementa el código en dicho método, obligando que las subclasses que heredan lo implementen obligatoriamente.

En el código siguiente se califica un método como abstracto y se sobrepasa obligatoriamente en las subclasses.

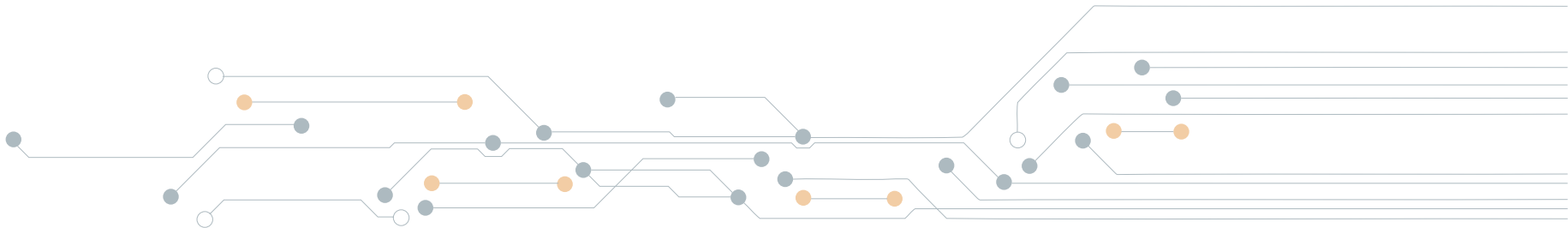
```
public abstract class Pieza {
    protected int numero;
    protected String nombre;
    //constructor
    //otras funciones

    //función abstracta
    abstract public void vender(int cantidad);
}

public class Plancha extends Pieza {
    private float largo;
    private float ancho;
    private float espesor;
    //Constructor
    //Otras funciones
    @Override
    public void vender(int metros) {
        if( cortar(metros)){
            System.out.println("==== VENDIDO
=====");
```

```
        }else{
            System.out.println("==== NO HAY
MATERIAL =====");
        }
    }

    public class Tubo extends Pieza {
        private float calibre;
        private String medida;
        //Constructor
        //Otras funciones
        @Override
        public void vender(int cantidad) {
            System.out.println("VENDIDO TUBO "+this.
nombre);
        }
    }
}
```



## 6. Polimorfismo

El polimorfismo se implementa cuando con un identificador de tipo superclase se referencia a un objeto instanciado de una subclase y se invoca a una función que dicha subclase sobrepasa.

En el código siguiente:

```
Pieza[] piezasvendidas = new Pieza[numVentas];

//Código que crea objetos Tubo y Plancha
//la referencia a cada uno de esos objetos es
//un elemento del array piezasvendidas

for (Pieza pieza : piezasvendidas) {
    int cantidad = aleatorio.nextInt(10) +
1;
    pieza.vender(cantidad);
}
```

Se invoca la función vender con una referencia a la superclase Pieza, pero en tiempo de compilación no se puede saber que vender será el que se ejecute.

La forma **pieza.vender(cantidad)** es polimórfica, independientemente de que la clase Pieza sea abstracta o no. El código del ejemplo siguiente muestra una aplicación del polimorfismo con esta jerarquía de clases.

```
public static void main(String[] args) {
    Random aleatorio = new Random();
    int metros = 0;
    int numVentas = aleatorio.nextInt(5) + 1;
    Pieza[] piezasvendidas = new
Pieza[numVentas];

    for (int i = 0; i < piezasvendidas.length;
i++) {
        int tipoPieza = aleatorio.nextInt(3) +
2;
        if (tipoPieza % 2 == 0) {
            piezasvendidas[i] =
                new Plancha(201, "Extra",
30.f, 4.5f, 0.3f);
        } else {
            piezasvendidas[i] =
                new Tubo(101, "Especial",
1.0f, "pulgadas");
        }
    }

    for (Pieza pieza : piezasvendidas) {
        int cantidad = aleatorio.nextInt(10) +
1;
        pieza.vender(cantidad);
    }
}
```

## 7. La utilización de final

El modificador final tiene varias funcionalidades en Java, estas son:

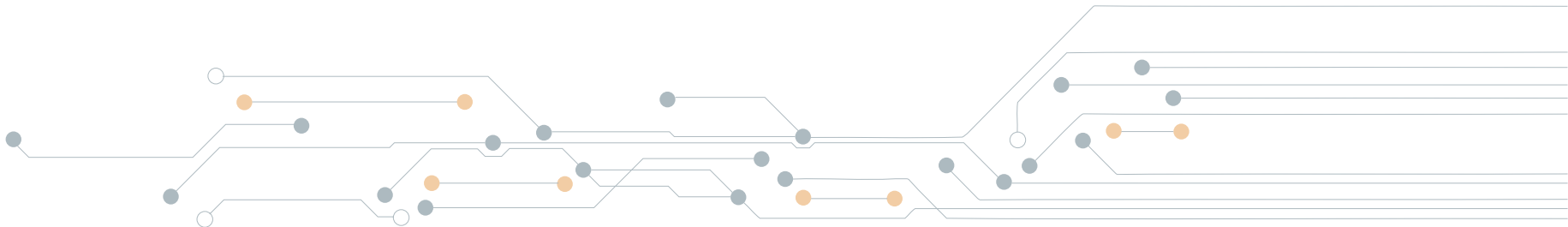
- Aplicada a **variables locales, de instancia o de clase**, hace que no se puede modificar su valor una vez creada, son las llamadas constantes.
- Aplicada a **funciones** de una superclase, implica que **las subclases no pueden sobrescribir dicha función**. Esto es así para que aquellas funcionalidades que no pueden cambiar en ninguna de las clase que hereden la función final.
- Aplicada a una **clase**, implica que **de esta clase no se puede heredar**, estas clases se suelen denominar "clase hoja" y por contraposición la clase que esta en la parte superior de la jerarquía se denomina "clase raíz".

Ejemplos:

```
public class Ejemplo{
    final public static int MAX_ELEMENTOS = 100; //
    constante de clase

    final private String mitipo = "Ejemplo"; //constante
    de instancia

    //otros datos de la clase
    //constructores
    //otras funciones
    public void haceAlgo(){
        final int numero=12; //constante local
        //Código de la función
    }
}
```



## 8. La clase Object

La clase Object es la raíz de la jerarquía de clases del API JSE. Toda clase tiene a Object como superclase. Por tanto todos los objetos, incluyendo los arrays, pueden ejecutar los códigos de las funciones implementadas en esta clase.

Como todas las clases son subclases de Object, cualquier variable de referencia a Object puede hacer referencia a un objeto de cualquier clase. En la imagen siguiente se utiliza referencia Object para un objeto de otra clase.

```
package com.juan.piezas;

public class PruebaObject {

    public static void main(String[] args) {
        Object objeto;
        //Se puede utilizar una variable Object para referenciar cualquier tipo
        objeto = new Acanalado (102,"Humos", 30.5f,"mms","plastico", true);
        System.out.println("objeto es Acanalado: "+
            (objeto instanceof Acanalado? "SI":"NO"));
        System.out.println("La clase de objeto es: "+objeto.getClass());
        objeto.

        Acanalado tubo= (Acanalado) objeto;
        tubo.

    }
}
```

Sólo se alcanzan funcionalidades de Object

No todos los objetos son Acanalado, por eso hay que moldear, así se pueden utilizar las funcionalidades de la clase específica

FIGURA 6.10: REFERENCIA A OBJECT.



Los métodos que define `Object` y están disponibles en todos los objetos son los que se muestran en la tabla siguiente, algunos ya han sido utilizados y comentados:

Método	Funcionalidad
<b><code>Object clone()</code></b>	Crea un objeto idéntico al clonado.
<b><code>boolean equals(Object o)</code></b>	Devuelve true si los dos objetos son iguales.
<b><code>void finalize()</code></b>	Se ejecuta instantes antes de liberar el espacio que ocupa el objeto.
<b><code>Class&lt;?&gt; getClass()</code></b>	Retorna la clase a la que pertenece el objeto.
<b><code>int hashCode()</code></b>	Obtiene el código hash de un objeto, el int que devuelve debe ser diferente para cada objeto.
<b><code>void notify()</code></b>	Reanuda la ejecución de un proceso.
<b><code>void notifyAll()</code></b>	Reanuda la ejecución de todos los procesos.
<b><code>String toString()</code></b>	Retorna una cadena que normalmente describe al objeto
<b><code>void wait()</code></b>	El subproceso se pone en estado de espera.
<b><code>void wait(long milis)</code></b>	
<b><code>void wait(long milis, int nanos)</code></b>	

**Tabla 6.1: Métodos de `Object`.**

- **Método `hashCode`:** Cuando se sobrepasa este método hay que garantizar en lo posible que el entero que devuelve será diferente para dos objetos distintos. Si dos objetos son iguales tendrían que devolver el mismo entero.

Las definiciones de `hashCode` y `equals` tienen que ser compatibles, de tal forma que si `x.equals(y)` es true, entonces `x.hashCode()` e `y.hashCode()` tienen que retornar el mismo entero.

Los IDEs como Eclipse y NetBeans pueden generar automáticamente el código tanto para `equals` y `hashCode`, que añadida en el código las variables miembro del objeto que se estimen oportunas para considerar la igualdad de dos objetos de la clase.

Un código generado por eclipse para estos dos métodos es el siguiente:

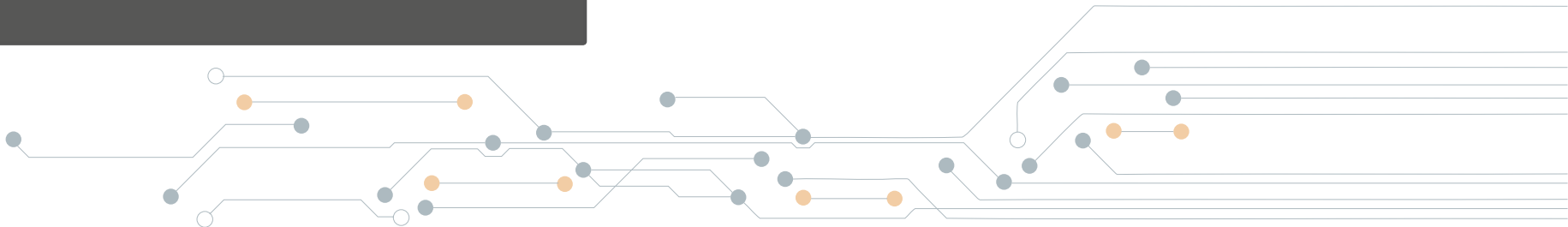
```
@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result +
        ((nombre == null) ? 0 :
nombre.hashCode());
    result = prime * result + numero;
    return result;
}

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Pieza other = (Pieza) obj;
    if (nombre == null) {
        if (other.nombre != null)
            return false;
    } else if (!nombre.equals(other.nombre))
        return false;
    if (numero != other.numero)
        return false;
    return true;
}
```

- **Método clone:** Esta definido en el ámbito protected, por tanto habrá que definirlo en la clase que sea necesario, no se puede invocar a un objeto esta funcionalidad protegida.

La función clone en todas sus definiciones (override) de todas las clases tiene que retornar una referencia a Object. Por tanto habrá que “moldear” hacer una conversión explícita al tipo concreto de la clase del objeto clonado.

Esta función lanza la excepción CloneNotSupportedException, por tanto en las funciones que la utilicen o se captura la excepción o se propaga, de momento hasta que se estudien las excepciones se propagara con throws CloneNotSupportedException.



En el código siguiente se muestra un código de la función clone y otro que prueba esta funcionalidad.

```
public class Liso extends Tubo {
    private String color;
    private float espesor;
    public Liso(int numero, String nombre,
                float calibre, String medida,
                String color, float espesor) {
        super(numero, nombre, calibre, medida);
        this.color = color;
        this.espesor = espesor;
    }
    @Override
    protected Object clone() throws CloneNotSupportedException {
        return new Liso(numero, nombre,
                        calibre, medida, color, espesor);
    }
    //Otras funciones
}
public static void main(String[] args)
throws
CloneNotSupportedException{

    Liso liso = new Liso (103, "Canal",
                        11.2f,"mms", "verde",0.3F);
    Liso clonliso = (Liso) liso.clone();
    if( liso == clonliso){
        System.out.println("No son clones");
    }
    else{
        System.out.println("Son clones");
        System.out.println(liso);
        System.out.println(clonliso);
    }
}
```

*Telefonica*

---

EDUCACIÓN DIGITAL