

Programación

UD 11: Utilización avanzada de clases

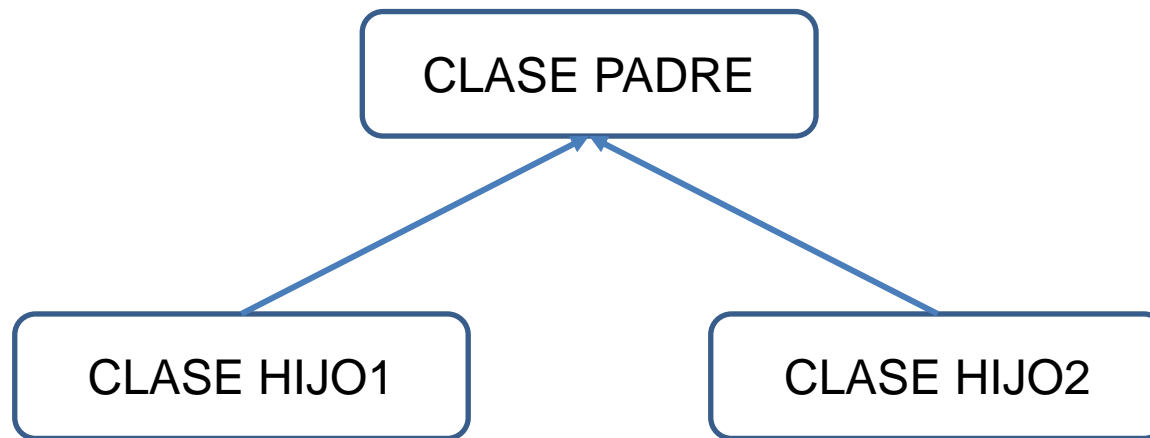
Utilización avanzada de clases

- 1.– Herencia
- 2.– Polimorfismo
- 3.– Interfaces
- 4.– Clases abstractas

1.- Herencia

1.- Herencia

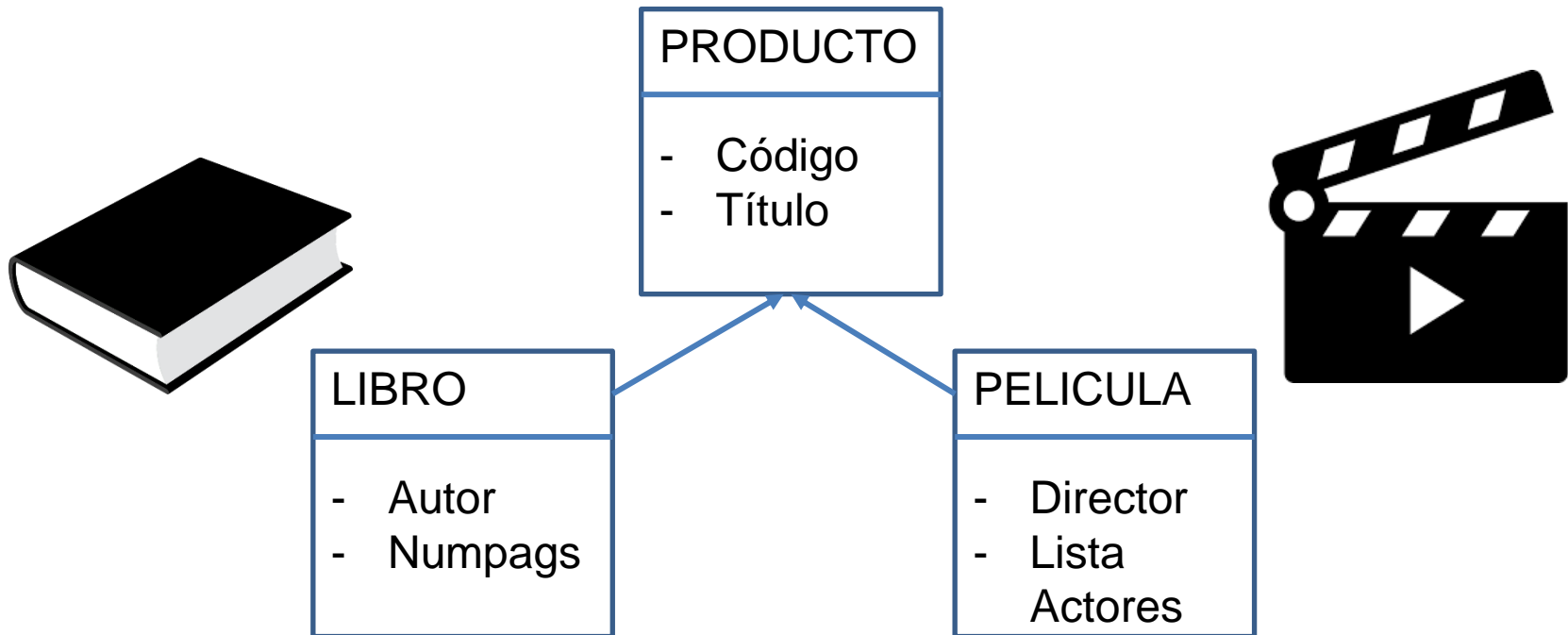
- La herencia permite definir una clase a partir de otra, siempre que entre ellas se pueda establecer una relación de tipo “ES UN”.
- Al heredar de una clase, se incorpora automáticamente todo su contenido (atributo y métodos) y podemos añadir además los elementos propios de la nueva clase.
- La clase de la que se hereda se suele llamar “clase padre” o “superclase” y la clase que hereda se llama “clase hija” o “subclase”.



1.- Herencia

-Por ejemplo, supongamos de un establecimiento que vende Películas y Libros, podríamos aglutinar las características de estos productos en una clase genérica llamada “Producto”, de la cual heredarían “Película” y “Libro”.

-De esta forma, “Película” y “Libro” tendrían todas las características propias de “Producto” además de definir las suyas propias.



1.- Herencia

-Primero vamos a definir la clase “Producto” que tendrá como únicos atributos el código y el título.

-Definimos su constructor que tendrá como argumentos justamente esos 2 atributos y, por razones de espacio, suponemos que también tendríamos el resto de funciones como getters y/o setters.

```
public class Producto {  
  
    private int codigo;  
    private String titulo;  
  
    public Producto(int codigo, String titulo) {  
        this.codigo = codigo;  
        this.titulo = titulo;  
    }  
  
    //resto codigo, getters, setters, etc...  
}
```

1.- Herencia

-Ahora definimos la clase “Libro” que, para indicar que hereda de “Producto”, utilizaremos la palabra reservada “extends” en la cabecera:

```
public class Libro extends Producto{  
  
    private int numpags;  
    private String autor;  
  
    public Libro(int codigo, String titulo, String autor, int numpags) {  
        this.codigo = codigo;  
        this.titulo = titulo;  
        this.autor = autor;  
        this.numpags = numpags;  
    }  
}
```

Sin embargo, tenemos un error, y es que Java nos dice que, para asignar los valores de código y de título **necesita llamar al constructor de la clase padre**.

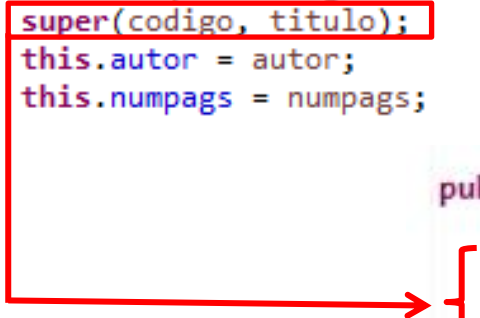
Implicit super constructor Producto() is undefined. Must explicitly invoke another constructor()

Dicho de otra forma, lo que Java nos quiere decir es que, si quieres crear un “Libro”, tienes que llamar al constructor de la clase “Producto”.

1.- Herencia

-...para llamar al constructor de la clase padre lo haremos mediante el método `super()` y le pasaremos como argumentos las variables necesarias para que se pueda crear un "Producto":

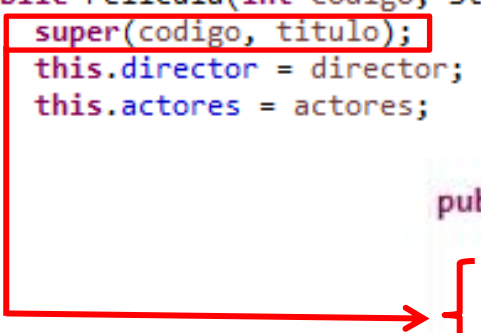
```
public class Libro extends Producto{  
  
    private int numpags;  
    private String autor;  
  
    public Libro(int codigo, String titulo, String autor, int numpags) {  
        super(codigo, titulo);  
        this.autor = autor;  
        this.numpags = numpags;  
    }  
}  
  
public class Producto {  
  
    private int codigo;  
    private String titulo;  
  
    public Producto(int codigo, String titulo) {  
        this.codigo = codigo;  
        this.titulo = titulo;  
    }  
}
```



1.- Herencia

-...de la misma forma crearíamos una “Película”, extendiendo de la clase “Producto” y añadiendo sus propios atributos, que en este caso son el director y una lista de actores.

```
public class Pelicula extends Producto {  
    private String director;  
    private String[] actores;  
  
    public Pelicula(int codigo, String titulo, String director, String[] actores) {  
        super(codigo, titulo);  
        this.director = director;  
        this.actores = actores;  
    }  
}  
  
public class Producto {  
    private int codigo;  
    private String titulo;  
  
    public Producto(int codigo, String titulo) {  
        this.codigo = codigo;  
        this.titulo = titulo;  
    }  
}
```

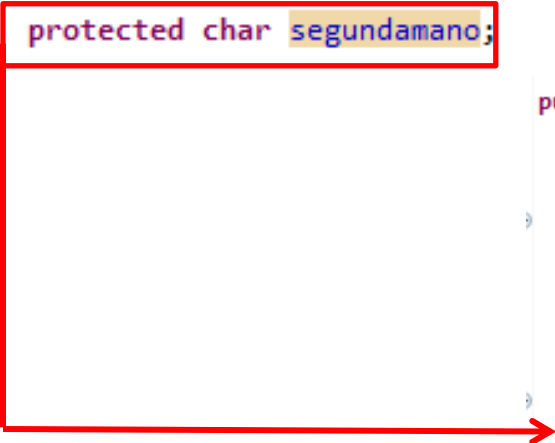


1.- Herencia

-Los atributos que se pueden heredar han de tener el calificador “protected” en la clase padre, ya que si los dejamos como “private”, las clases “hija” no podrán acceder al mismo.

Por ejemplo, imaginemos un atributo de “segundamano” que definimos en la clase padre y que luego lo utilizaremos en la clase hija. Procederíamos del siguiente modo:

```
public class Producto {  
  
    private int codigo;  
    private String titulo;  
  
    protected char segundamano;  
  
    public class Pelicula extends Producto {  
        private String director;  
        private String[] actores;  
  
        public Pelicula(int codigo, String titulo, String director, String[] actores) {  
            super(codigo, titulo);  
            this.director = director;  
            this.actores = actores;  
        }  
  
        public void set_segundamano() {  
            this.segundamano = 'S';  
        }  
    }  
}
```



1.- Herencia

-La herencia también es útil cuando queremos que un objeto reutilice los métodos de la clase padre. Imaginemos que en la clase “Producto” queremos definir un método para que el producto se anuncie:

```
public class Producto {  
    //resto codigo  
    public void anuncio() {  
        System.out.println("Hola, soy un producto genérico");  
    }  
}
```

Cuando ejecutamos el siguiente código, obtendremos el mismo mensaje, ya sea un “Libro” o una “Película”, ya que ambas heredan el método de su clase padre, “Producto”:

```
public static void main(String[] args) {  
    String actores[] = {"Ron Livingstone", "Jennifer Aniston"};  
    Pelicula peli = new Pelicula(1, "Trabajo Basura", "Mike Judge", actores);  
    Libro libro = new Libro(1, "Don Quijote", "Miguel de Cervantes", 594);  
    peli.anuncio();  
    libro.anuncio();  
}
```

<terminated> Principal (31) [Java Application] C:\Program Files\JAVA\openjdk-12.0.1.
Hola, soy un producto genérico
Hola, soy un producto genérico

1.- Herencia

-..pero, ¿y si queremos distinguir entre un libro y una película?. Pues no tendremos más que utilizar un método propio en cada clase. Incluso podemos llamarlo igual, ya que en la llamada, **Java no le hará caso al método de la clase padre, sino al método de la clase hija** o subclase:

```
public class Libro extends Producto{  
  
    //resto codigo  
  
    public void anuncio() {  
        System.out.println("Hola, soy un libro");  
    }  
  
}
```

En la ejecución ahora obtendríamos...

```
public static void main(String[] args) {  
    String actores[] = {"Ron Livingstone", "Jennifer Aniston"};  
    Pelicula peli = new Pelicula(1, "Trabajo Basura", "Mike Judge", actores);  
    Libro libro = new Libro(1, "Don Quijote", "Miguel de Cervantes", 594);  
    peli.anuncio();  
    libro.anuncio();  
}
```

```
<terminated> Principal (31) [Java Application] C:\Program Files\JAVA\openjdk-12.0.1_  
Hola, soy un producto genérico  
Hola, soy un libro
```

2.- Polimorfismo

2.- Polimorfismo

-El polimorfismo permite que un objeto de un tipo se pueda comportar como cualquiera de sus subtipos y pueda, por tanto, adoptar múltiples formas. Con el código anterior:

```
Pelicula peli = new Pelicula(1,"Trabajo Basura", "Mike Judge", actores);  
Libro libro = new Libro(1,"Don Quijote", "Miguel de Cervantes",594);
```

Podríamos crear arrays polimórficos del siguiente modo:

-Arrays estáticos:

```
Producto[] lista_prod = new Producto[10];  
lista_prod[0] = peli;  
lista_prod[1] = libro;
```

-Arrays dinámicos:

```
ArrayList<Producto> lista_productos = new ArrayList<Producto>();  
lista_productos.add(peli);  
lista_productos.add(libro);
```

2.- Polimorfismo

-También tenemos polimorfismo cuando definimos distintos métodos que reciben el mismo nombre pero que difieren en:

- Número de parámetros
- Tipo de parámetros
- Orden de los parámetros

```
public class Libro extends Producto{  
  
    private int numpags;  
    private String autor;  
  
    public Libro(int codigo, String titulo, String autor, int numpags) {  
        super(codigo, titulo);  
        this.autor = autor;  
        this.numpags = numpags;  
    }  
  
    public void anuncio() {  
        System.out.println("Hola, soy un libro");  
    }  
  
    public void anuncio(double pvp) {  
        System.out.println("El libro se llama " + titulo + " y cuesta " + pvp + " euros.");  
    }  
}
```

2.- Polimorfismo

-Desde el programa principal, podríamos llamar a cualquiera de los métodos pasándoles los argumentos que necesitemos en cada ocasión:

```
import java.util.*;
public class Principal {
    public static void main(String[] args) {

        Libro libro = new Libro(1,"Don Quijote", "Miguel de Cervantes",594);

        libro.anuncio();
        libro.anuncio(19.95);
    }
}
```

Con lo que el programa imprimiría:

```
<terminated> Principal (31) [Java Application] C:\Program Files\JAVA\openjdk-12.0.1_
Hola, soy un libro
El libro se llama Don Quijote y cuesta 19.95 euros.
```

3.- Interfaces

3.- Interfaces

- Una interfaz en Java consiste esencialmente en una lista de declaraciones de métodos sin implementar.
- Estos métodos sin implementar indican un comportamiento, un tipo de conducta, aunque no se especifica cómo será ese comportamiento (implementación), pues eso dependerá de las características de cada clase que decida implementar esa interfaz.
- Para la declaración de una interfaz se utiliza la palabra reservada “interface” en lugar de la palabra “class”.
- Todos los miembros de la interfaz (atributos y métodos) son public, de manera implícita, de forma que no es necesario indicarlo.
- Todos los atributos son constantes y públicos. Hay que darles un valor inicial.
- Todos los métodos, como se ha dicho, estarán sin implementar, de forma que sólo veremos su cabecera, sin cuerpo.

3.- Interfaces

-Un ejemplo muy sencillo de interfaz sería el siguiente:

```
public interface Interfaz_Producto {  
    final int IVA = 21;  
    public void descripcion_producto();  
    public float precio_con_iva();  
}
```

Y haremos que la clase “Libro” implemente esa interfaz con **“implements”**:

```
public class Libro extends Producto implements Interfaz_Producto{  
  
    private int numpags;  
    private String autor;  
    private float precio_sin_iva;  
  
    public Libro(int codigo, String titulo, String autor, int numpags, float precio_sin_iva) {  
        super(codigo, titulo);  
        this.autor = autor;  
        this.numpags = numpags;  
        this.precio_sin_iva = precio_sin_iva;  
    }  
  
    public void descripcion_producto() {  
        System.out.println("El libro se titula " + titulo + " y está escrito por " + autor);  
    }  
  
    public float precio_con_iva() {  
        float impuestos = (precio_sin_iva * IVA)/100;  
        return precio_sin_iva + impuestos;  
    }  
}
```

3.- Interfaces

-De forma que, desde el programa principal, podríamos crear un objeto de la clase “Libro”:

```
import java.util.*;
public class Principal {
    public static void main(String[] args) {

        Libro libro = new Libro(1,"Don Quijote", "Miguel de Cervantes",594, 50);
        libro.descripcion_producto();
        System.out.println(libro.precio_con_iva());
    }
}
```

-Lo cual produciría en la salida el siguiente resultado:

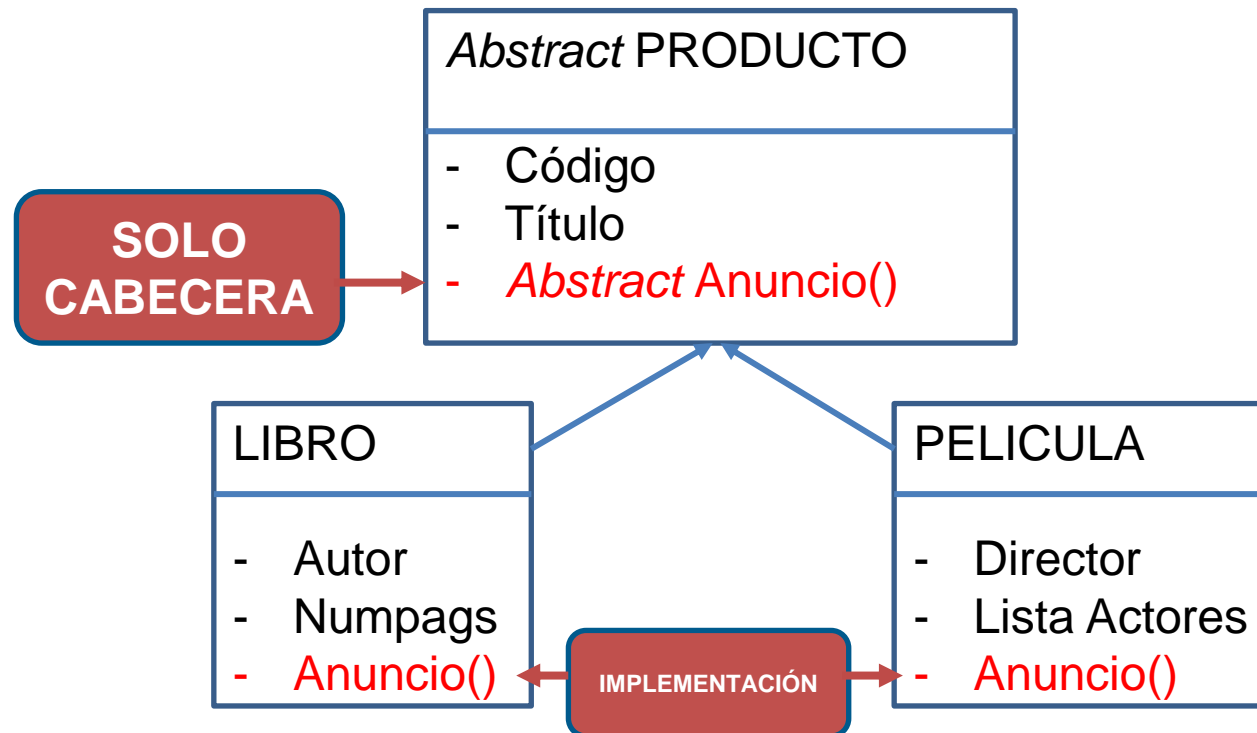
```
<terminated> Principal (31) [Java Application] C:\Program Files\JAVA\openjdk-12.0.1_window
El libro se titula Don Quijote y está escrito por Miguel de Cervantes
60.5
```

4.- Clases abstractas

4.- Clases abstractas

-Una clase abstracta es aquella que contiene, al menos, un método abstracto. Además, no se puede instanciar.

-¿Y qué es un método abstracto?, pues es aquél del cual sólo conocemos su cabecera, dejando la implementación de su cuerpo a las clases que heredan de él. De hecho, las subclases están **obligadas** a definirlo.



4.- Clases abstractas

```
public abstract class Producto {  
  
    private int codigo;  
    protected String titulo;  
  
    public Producto(int codigo, String titulo) {  
        this.codigo = codigo;  
        this.titulo = titulo;  
    }  
  
    public abstract void anuncio();  
}
```

```
public class Libro extends Producto{  
  
    private int numpags;  
    private String autor;  
    private float precio_sin_iva;  
  
    public Libro(int codigo, String titulo,  
String autor, int numpags, float precio_sin_iva) {  
        super(codigo, titulo);  
        this.autor = autor;  
        this.numpags = numpags;  
        this.precio_sin_iva = precio_sin_iva;  
    }  
  
    public void anuncio() {  
        System.out.println("Hola soy un libro");  
    }  
}
```

```
public class Pelicula extends Producto {  
    private String director;  
    private String[] actores;  
  
    public Pelicula(int codigo, String titulo,  
String director, String[] actores) {  
        super(codigo, titulo);  
        this.director = director;  
        this.actores = actores;  
    }  
  
    public void anuncio() {  
        System.out.println("Hola soy una pelicula");  
    }  
}
```