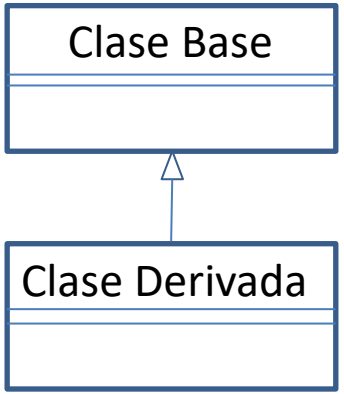
INTRODUCCIÓN A LA HERENCIA

HERENCIA

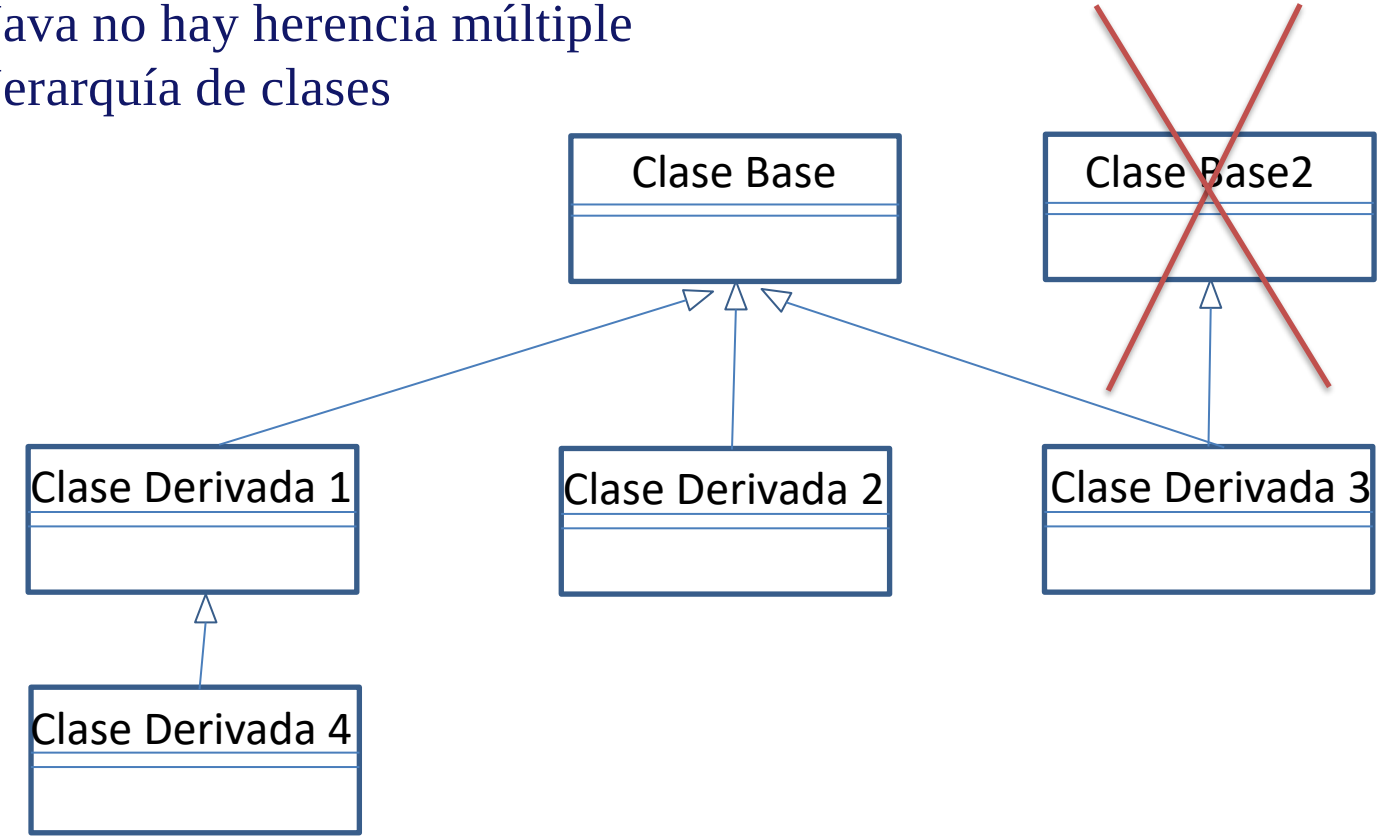
- Funcionalidad fundamental de la POO.
- Mecanismo que permite crear nuevas clases a partir de otras ya existentes.
- Representa una relación **es-un** entre dos clases, en la cual una clase (**subclase o clase derivada**) es una especialización o extensión de otra clase (**superclase o clase base**) más general.
- Palabra reservada **extends**

```
class ClaseDerivada extends ClaseBase {

      //miembros específicos de la clase
      //derivada
}
```



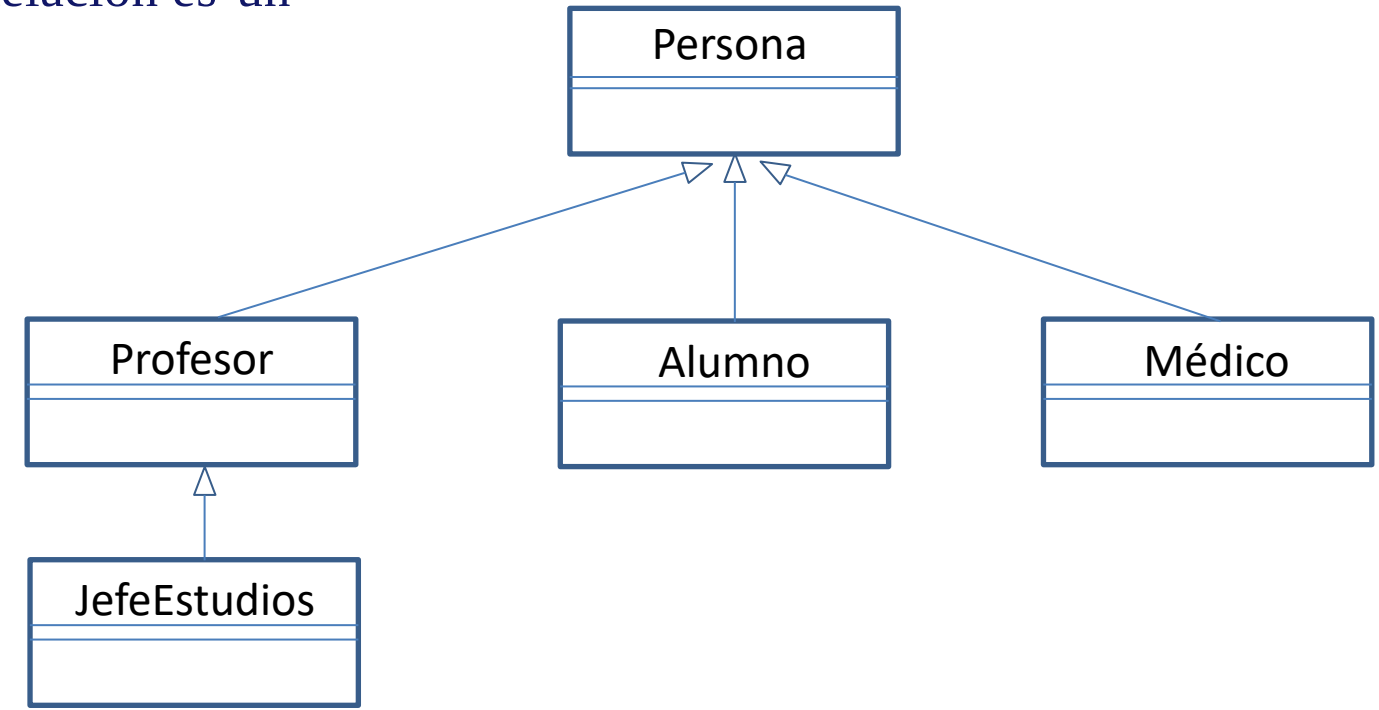
HERENCIA

- Java no hay herencia múltiple
- Jerarquía de clases



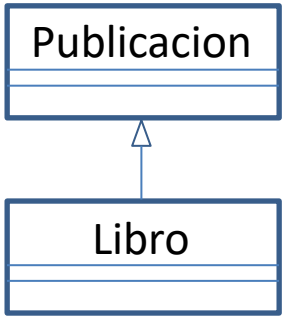
HERENCIA - Ejemplo

- Relación es-un



HERENCIA

- **Reusabilidad** del código: La subclase hereda tanto los atributos como los métodos definidos por la superclase (**no** los constructores).
- La subclase puede definir nuevos atributos y nuevos métodos (**extensión**), así como redefinir métodos de la superclase (**especialización**)



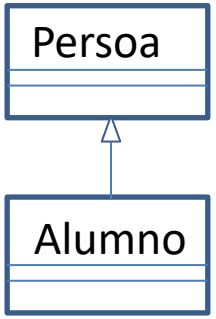

```
class Publicacion {
    private int numeroDePaxinas;
    private float prezo;
}
```

```
class Libro extends Publicacion {
    private String titulo;
    private String tipoPortada;
    private String isbn;
    private String nomeAutor;
    private String editorial;
}
```

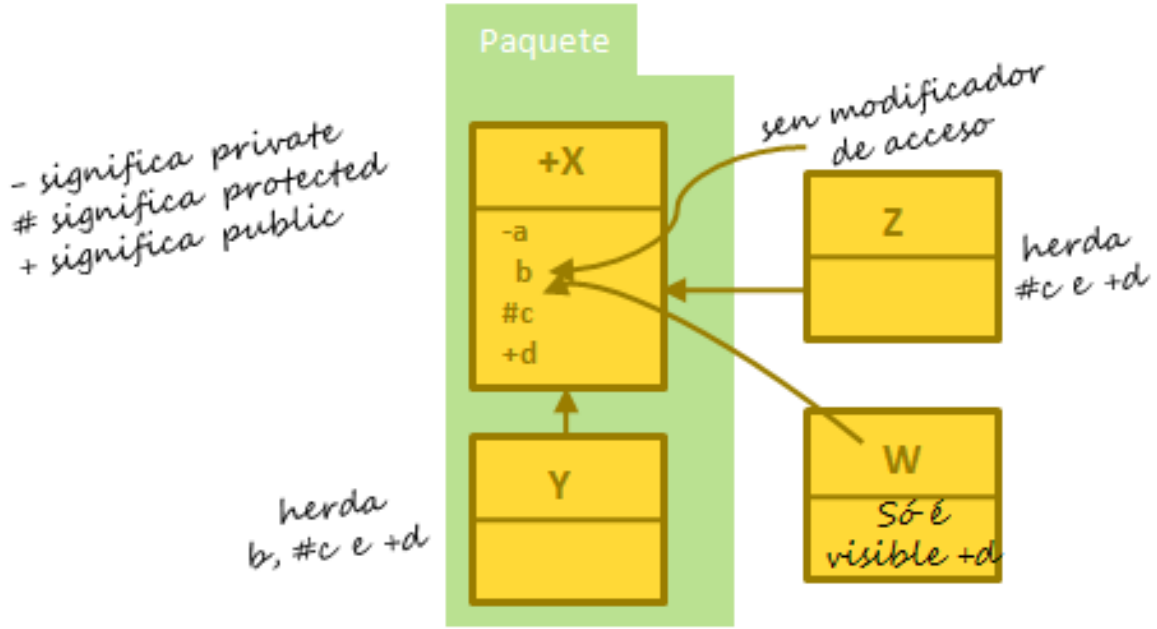
HERENCIA - ejemplo

```
public class Persoa {
	private String nif;
	private String nome;

	public String getNif() {
		return nif;
	}

	public void setNif(String nif) {
		this.nif = nif;;
	}

	public String getNome() {
		return nome;
	}

	public void setNome(String nome) {
		this.nome = nome;
	}
}
```

```
class Alumno extends Persoa {
	private String curso;

	public String getCurso() {
		return curso;
	}

	public String setCurso() {
		return curso;
	}
}
```



HERENCIA - Acceso protected



HERENCIA y SOBREESCRITURA - Override

- Una clase que extiende a otra, hereda y puede añadir todos los atributos y métodos que necesite.
- Si un nuevo atributo (o método) se llama igual que otro de una superclase, lo solapa, y ya no puede accederse al de la clase padre. Para sobreescribir un método, debe tener la misma firma y devolver el mismo tipo de datos (o una subclase).
- Si queremos acceder al método o al atributo que quedó oculto se utiliza **super**

```
super.metodo()
```

```
super.atributo
```

- La anotación **@Override** permite detectar errores al redefinir el método, como errores en el nombre o en la lista de parámetros.

CONSTRUCTORES Y SUPER

- Los constructores **no** se heredan, aunque sean **public**

- Un constructor de una subclase puede usar **super** para invocar a un constructor de su clase base.

- Si una subclase no lo invoca, la JVM lo hace por él. La clase base debe tener entonces un constructor sin parámetros.

- Cuando se crean un objeto de la clase derivada:
  - ✓ Primero se ejecuta el constructor de la superclase
  - ✓ Después se ejecuta el constructor de la clase derivada

CONSTRUCTORES Y HERENCIA

```
public class Alumno extends Persoa {
    private String curso;

    public Alumno() {
        super(); // No hace falta ponerlo
        System.out.println("Executando o construtor de Alumno");
    }
    public Alumno(String nif, String nome, String curso){
        super(nif, nome); //llamada al constructor de la superclase
        this.curso=curso;
    }
}
```

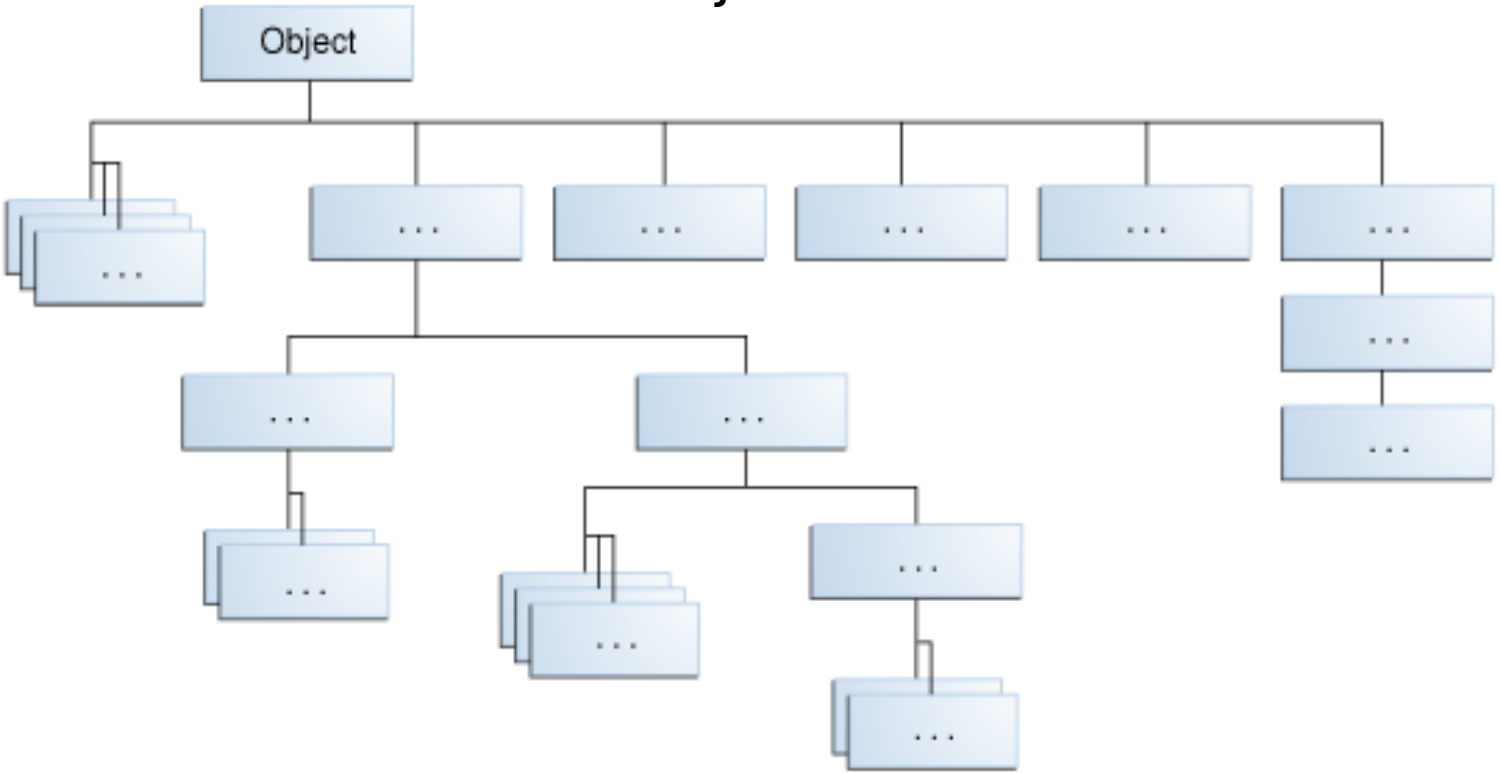

HERENCIA y final

- Para que no se pueda heredar de una clase, se incluirá el modificador **final** en su definición

```
public final class ClaseFinal {

}
```


CLASE Object

```
class Cualquiera extends Object {
}
```



HERENCIA DE OBJECT

- Es la superclase padre de todas las clases en Java.
- Todo objeto, de forma explícita o implícita, hereda de Object.
- Métodos heredados: **equals**, **getClass, hashCode, toString**, …
- Sobreescritura: @Override

MÉTODO equals

- El método **equals** devuelve un booleano indicando si un objeto es igual a otro.
- Realiza una comparación de las referencias de los objetos para determinar si hay igualdad entre ambos.
- NetBeans lo autogenera.
- Tendremos que establecer cuando dos instancias de un objeto son iguales

MÉTODO equals

Normas a seguir:

- Debe ser **reflexivo:** para *x* distinto de *null*, *x.equals(x)* devuelve *true*.
- Debe ser **simétrico:** para *x* e *y* distintos de *null*, *x.equals(y) == y.equals(x)*.
- Debe ser **transitivo:** para *x*, *y* e *z* distintos de *null*, se *x.equals(y)* es true, e *y.equals(z)* es *true*, entonces *x.equals(z)* debe ser *true*.
- Debe ser **consistente:** para *x* e *y* distintos de *null*, todas las llamadas a *x.equal(y)* devuelven siempre el mismo valor, mientras no cambien el valor de los atributos comparados.
- Si *x* es distinto de *null*, *x.equals(null)* debe ser *false*.

MÉTODO hashCode

- Calcula un valor único (un número) para un objeto.
- Se usa principalmente para la búsqueda rápida en estructuras de datos como HashMap, HashSet, etc.
- En estas estructuras, sirve para acceder a él de forma rápida, sin tener que hacer una búsqueda comparando todas sus propiedades.
- Por definición, si dos objetos son iguales, su hash code también debe serlo.
- Si almacenamos nuestros objetos en alguna estructura de tipo Hash, si sobrescribimos el método **equals**, también tenemos que sobrescribir **hashCode** para que se cumpla esa propiedad.
- NetBeans lo autogenera.

MÉTODO getClass

- Proporciona una instancia de la clase **Class** con las propiedades de la clase desde la que se creó el objeto:

  - ✓ nombre de la clase,
  - ✓ los métodos disponibles,
  - ✓ los atributos,
  - ✓ etc..

MÉTODO toString

- Devuelve una representación en String del objeto.
- Por defecto, devuelve el tipo (la clase) y su hashCode.

paquete.Persona@2a139a55

- Lo podemos sobrescribir para que represente los valores.
- Dos objetos iguales deben tener la misma representación.
- NetBeans lo autogenera.

POLIMORFISMO

- Un mismo elemento (variable, método o clase) puede adquirir distintas formas

- Ejemplo en los métodos:

```
public Persona(); //Constructor sin parámetros
public Persona(String nombre); //Constructor con 1 parámetro
public Persona(String nombre, int edad); //Constructor con 2
                                          //parámetros
```


POLIMORFISMO y HERENCIA

- La herencia da lugar a una jerarquía de tipos

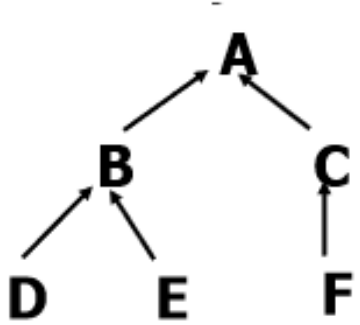
  Superclase -> define un tipo
  Subclase -> define un subtipo del tipo definido en la superclase

- El polimorfismo permite que un objeto de una **subclase** pueda ser considerado y referenciado como un objeto de la **superclase** (**Principio de sustitución**)

```
Persona persoa = new Alumno(“2525252525F”, “Ana”, “1DAW”);
```


VARIABLES POLIMÓRFICAS

- El polimorfismo implica que una variable tiene un **tipo estático** y un conjunto de **tipos dinámicos**
- El **tipo estático** es el tipo de la variable (en tiempo de compilación); y el **tipo dinámico** es el tipo del objeto al que referencia la variable (en tiempo de ejecución)




```
A oa; B ob; C oc;
```

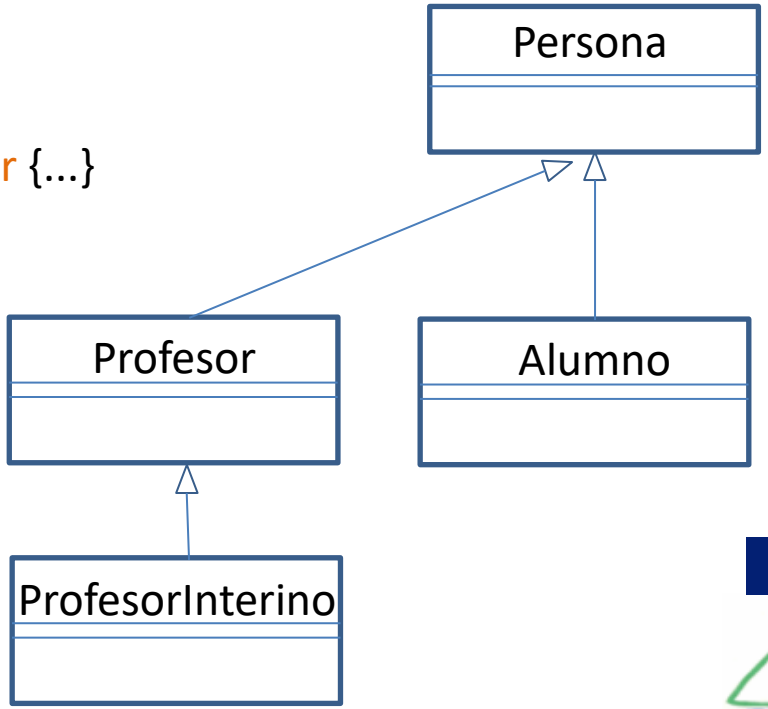
te(oa) = A ctd(oa) = {A,B,C,D,E,F}
te(ob) = B ctd(ob) = {B, D, E}
te(oc) = C ctd(oc) = {C,F}

REFERENCIAS Y SUBCLASES

- Una subclase puede ser accedida a través de una referencia de una superclase.

```
public class Persoa {...}
public class Profesor extends Persoa {...}
public class ProfesorInterino extends Profesor {...}
public class Alumno extends Persoa {...}
```

```
Persoa p1 = new Persoa();
Persoa p2 = new Profesor();
Persoa p3 = new ProfesorInterino();
Persoa p4 = new Alumno();
```



POLIMORFISMO

- Las variables polimórficas reducen la cantidad de código y facilita la reusabilidad.

```
for (Profesor tmp : listaProfesores) { … }
for (ProfesorInterino tmp : listaProfesoresInterinos) { … }

-------

for (Profesor tmp : listado) { … }
```


OPERADOR instanceof

- Compara instancias que se relacionan dentro de una jerarquía, verificando el tipo de la variable
- Sólo se usa asociado a un condicional

```
Persoa p1 = new Persoa();
Persoa p2 = new Profesor();
Persoa p3 = new ProfesorInterino();

if  (p3   instanceof   ProfesorInterino) {
    ...
}  else  {
    ...
}
```


CASTING: conversión entre clases

- **UpCasting**: conversión implícita
- **DownCasting** (o **casting**): conversión explícita
- Ejemplo con tipos primitivos:

```
int k;
byte b = 0;

k = b;  //implícito
int k = (int) 5.8;  //explícito (casting)
```


CASTING: conversión entre clases

```
public class Poligono { ... }
public class Rectangulo extends Poligono{ ... }
public class Triangulo extends Poligono{ ... }
```

- **UpCasting**: conversión implícita . Tipos compatibles

```
Poligono p;
Triangulo t = new Triangulo(3,5,2);

p = t; //porque un triángulo es un polígono.
```

p no puede invocar métodos propios de la clase Triangulo (a menos que sea un método que haya sobreescrito triángulo, entonces, se llamará al sobreescrito)

CASTING: conversión entre clases

```
public class Poligono { ... }
public class Rectangulo extends Poligono{ ... }
public class Triangulo extends Poligono{ ... }
```

- **DownCasting** (simplemente casting): conversión explícita.

```
Poligono p = new Triangulo(3,5,2); //upcasting
Triangulo t ;
t = (Triangulo) p ; //casting
```

Ahora podemos invocar todos los métodos de Triangulo sobre p

POLIMORFISMO - OCULTACIÓN DE MÉTODOS

- Una subclase puede redefinir cualquier método heredado, de manera que el método de la clase base queda oculto.

```
Persoa persoa;
persoa = new Alumno(“2525252525F”, “Ana”, “1DAW”);

persoa.mostrarDatos();
```

- Java elige en tiempo de ejecución, el tipo de objeto: Vinculación dinámica
- Lo usaremos en la llamada a métodos pasando objetos