TEMA 5. POO AVANZADA Y EXCEPCIONES

TEMA 5. Introducción

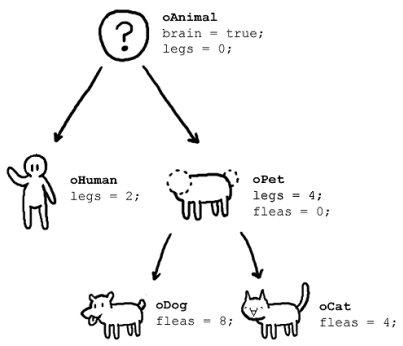
Principios POO:

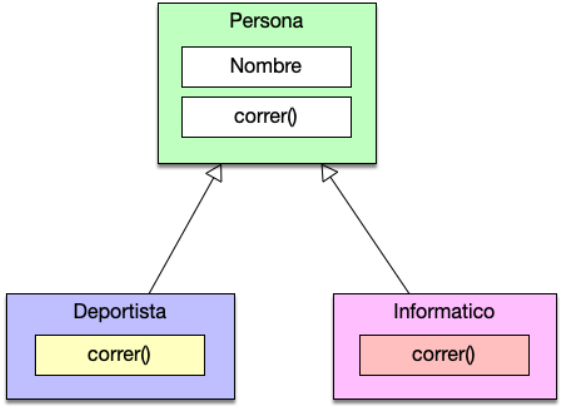
- **Abstracción.**
- **Encapsulación.**
- **Ocultación información.**
- **HERENCIA.**
- **POLIMORFISMO.**

Herencia: Describir relaciones entre clases relacionadas de forma jerárq

Polimorfismo: 2 tipos.

- Objectos:Comportamiento diferente ante acciones iguales sobre objet
- Funciones y métodos: Sobrecarga.





TEMA 5. Herencia

Reutilizar métodos y atributos.

Objetivos:

- Reutilización. No volver a escribir una y otra vez el mismo código.
- Extensibildiad. Ampliar la funcionalidad.

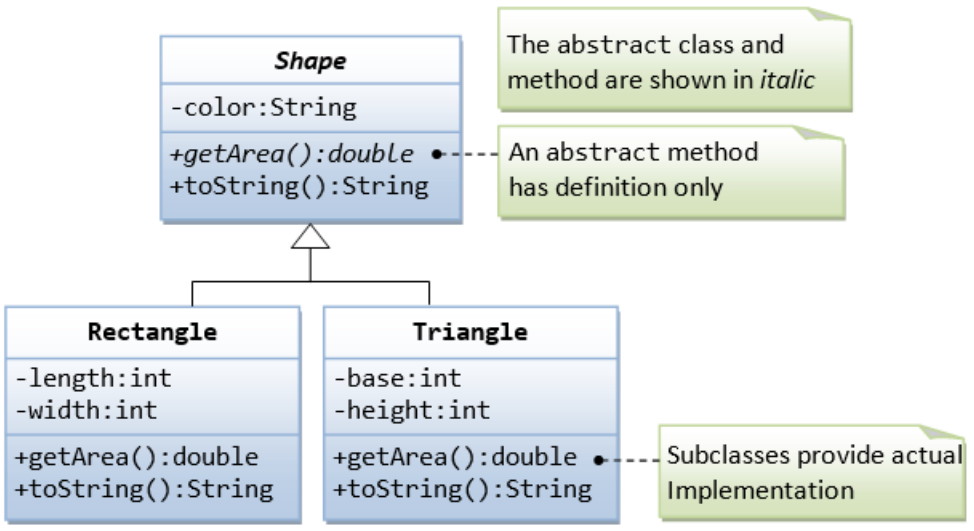
Si la clase B hereda de la clase A:

- B incorpora los atributos (estructura) y los métodos (comportamiento) de A.
- B puede:
- Añadir nuevos métodos y atributos.
- Cambiar la visibilidad.
- Redefinir o sobreescribir (@override) métodos de A

TEMA 5. Herencia

Si C hereda de B, D herdad de B y B hereda de A entonces:

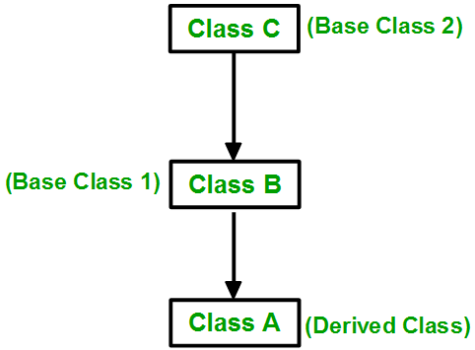
- A se denomina superclase.
- B, C y D son subclases de A.
- B descenciente directo de A.
- C descendiente indirecto de A.
- D descenciente indirecto de A.

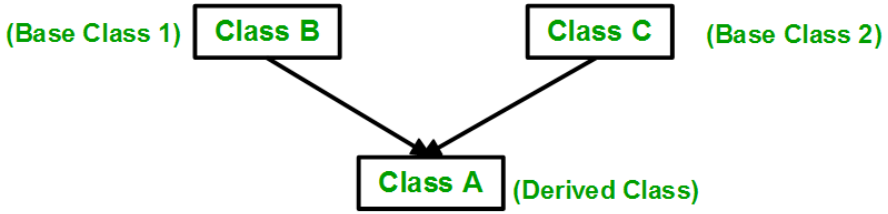


TEMA 5. Herencia

Tipos de herencia:

- Simple: Más común en la actuliadad. Una clase solo puede heredad de una y solo una clase. Ejemplos: Java, C#.
- Múltiple: Una clase puede heredad de más de una clase. Problema métodos duplicados. Ejemplo: C++.





TEMA 5. Herencia

Constructores y herencia:

El orden correcto de creación es padre→hija, ya que es necesario realizar la reserva de memoria en el padre y despues ampliarla para la clase hija.

Se puede hacer:

- Forma explícita: Llamado desde el constructor de la clase hija a alguno de los constructores válidos de la clase padre.
- Forma implícita: Llamando al constructor por defecto.

Java y Python: super(); super(parámetros).

C++: nombre_padre(); nombre_padre(parámetros).

TEMA 5. Herencia

Clases y métodos abstractos.

Situaciones:

•No es lógico poder crear objetos de la clase padre.

•Comportamientos que no se pueden definir en la clase padre, o es necesario información que está disponble en clases hijas.

Solución:

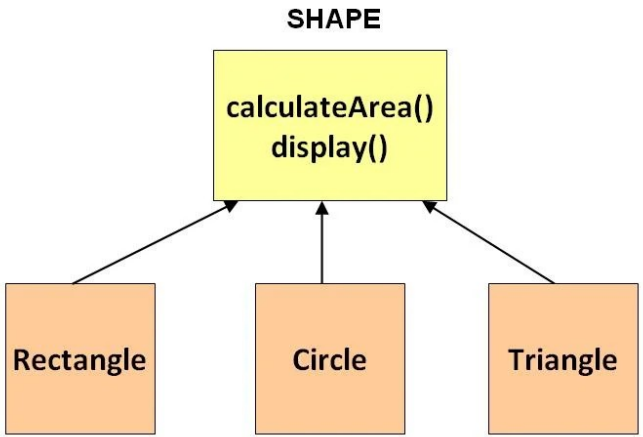
•Clases abstractas: No se puden instanciar, se definen como abstractas o al menos unos de sus métodos es abstracto. Las clases hijas habrán de definir todo lo que sea abstracto en el padre.

•Método abstracto: Se indica su existencia, pero no su implementación, que recae sobre la primera clase hija no abstracta.

NO ES POSIBLE CREAR OBJETOS DE CLASES ABSTRACTAS, PERO SI ATRIBUTOS Y VARIABLES DE ESE CLASE (BASE DEL POLIMORFISMO DE CLASES).

Shape es abstracta. Definirla en 2D.

- ¿Es posible crear objetos de tipo Shape?
- ¿Se puede calcular el área de Shape?
- ¿Es necesario definir constructores es Shape?
- ¿Cuál es el proceso normal de ejecucion al crear un rectángulo?
- ¿Cómo se definen el métodos calculateArea y display en Shape?. Indicar la razón.
- ¿Qué metodos se han de definir el Rectangle,Circle y Triangle?
- ¿La implementación de calculateArea es la misma en todas las clases?

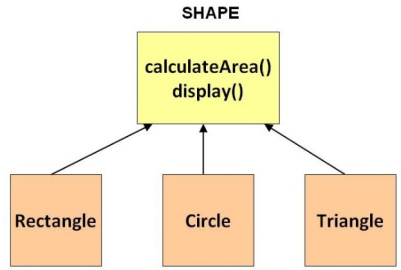


Implementación en pseudocódigo.

```
clase abstracta Figura:
        privado atributo x:entero;
         privado atributo y:entero;

        publico Constructor(){
                Yo.x=-1;
                Yo.y=-1;
         }
        publico Constructor(entero x,entero y){
                Yo.x=x;
                Yo.y=y;
        }
        ....getters y setters
        publico abstracto real calcularArea();
        publico abstracto nada mostrar();
        }
```

```
clase Rectangulo hereda_de Figura{
        private atributo ancho:entero;
        private atributo alto:entero;
        publico Constructor(){
                padre();
                Yo.ancho=-1;
                Yo.alto=-1;
        }
        public Constructor(entero x, entero y, entero ancho, entero alto){
                padre(x,y);
                Yo.ancho=ancho;
                Yo.alto=alto;
        }
        public entero calcularArea(){
                devuelve Yo.alto x Yo.ancho;
        }
        public vacio mostrar(){
                Imprimir ("Soy un rectángulo en la posición "+Yo.getX()+","+Yo,getY()+" con dimensiones "+Yo.ancho+Yo.alt
        }
}
```



- Si se retira la palabra abstracta de la clase
- ¿Qué método es el primero en llamarse e
- Para acceder al atributo x de la clase padr

TEMA 5. Herencia

Ejercicio.

Se desea crear un juego de estrategia, el juego se componen de unidades, cada unidad posee capacidades (atributos) como ataque, defensa, capacidad de trabajo, movilidad o velocidad, y toda unidad puede moverse, atacar o defenderse. Las unidades se clasifican en arqueros, infantería, caballería, artillería y trabajador.

Diseñar un sistema de clases usando herencia que permita reutilizar código y extender la funcionalidad, tener en cue

Herencia en Java.

Características:

- Herencia simple.
- Clases abstractas.
- Métodos abstractos.

Heredar de una clase: Palabra reservada extends

Restricciones visibilidad en métodos y atributos

Modificador (palabra reservada)	Misma clase	Mismo paquete	Subclase de otro paquete	Resto
private	✓	✗	✗	✗
(defecto)	✓	✓	✗	✗
protected	✓	✓	✓	✗
public	✓	✓	✓	✓

Ejemplo.

Se desea automatizar la gestión de las máquinas de un gimnasio. A las máquinas poseen una API REST (enviar comandos usando JSON y HTTP). Se tiene inicialmente 3 tipos de máquinas:

- Cinta de correr.
- Bicicleta de spinning.
- Máquina de remo.

Todas las máquinas tienen: Fecha instalación, versión software, horas de funcionamiento, estado(encendida, apagada, averiada y mantenimiento), métodos: encender, apagar, ir y pintar display.

Ejemplo.

```
public class Maquina {
   protected Date fecha_instalacion;
   protected String version_software;
   protected int horas_funcionando;
   protected Revision[] revisiones;
   protected EstadoMaquina estado;
   protected String identificador;
   public Maquina() {
      System.out.println("Soy el constructor por defecto de máquina");
      this.fecha_instalacion = new Date();
      this.revisiones = new Revision[50];
      this.estado = EstadoMaquina.APAGADA;
      this.identificador = "";
   }
   public Maquina(String version, int horas) {
      System.out.println("Soy el constructor sobrecargado de máquina");
      this.fecha_instalacion = new Date();
      this.revisiones = new Revision[50];
      this.version_software = version;
      this.horas_funcionando = horas;
      this.estado = EstadoMaquina.APAGADA;
      this.identificador = "";
   }
```

```
public void Encender() {
    this.estado = EstadoMaquina.ENCENDIDA;
 }
 public void Apagar() {
    this.estado = EstadoMaquina.APAGADA;
 }
 public void Instalar(String ruta, String version) {
    this.version_software = version;
    this.horas_funcionando = 0;
 }
 public void Resetear() {
    this.estado = EstadoMaquina.MANTENIMIENTO;
    this.horas_funcionando = 0;
    this.estado = EstadoMaquina.APAGADA;
 }
 public String pintardisplay() {
     String vuelta;
    vuelta = "Máquina:" + this.identificador + " software:" + this.version_software
          + " horas funcionamiento:" + this.horas_funcionando + " estado:" + this.estado;
    return vuelta;
 }
```

```
enum EstadoMaquina {
   ENCENDIDA,
   APAGADA,
   AVERIADA,
   MANTENIMIENTO
}
```

Ejemplo.

```
public class Maquina {
   protected Date fecha_instalacion;
   protected String version_software;
   protected int horas_funcionando;
   protected Revision[] revisiones;
   protected EstadoMaquina estado;
   protected String identificador;
   public Maquina() {
      System.out.println("Soy el constructor por defecto de máquina");
      this.fecha_instalacion = new Date();
      this.revisiones = new Revision[50];
      this.estado = EstadoMaquina.APAGADA;
      this.identificador = "";
   }
   public Maquina(String version, int horas) {
      System.out.println("Soy el constructor sobrecargado de máquina");
      this.fecha_instalacion = new Date();
      this.revisiones = new Revision[50];
      this.version_software = version;
      this.horas_funcionando = horas;
      this.estado = EstadoMaquina.APAGADA;
      this.identificador = "";
   }
```

```
public void Encender() {
    this.estado = EstadoMaquina.ENCENDIDA;
 }
 public void Apagar() {
    this.estado = EstadoMaquina.APAGADA;
 }
 public void Instalar(String ruta, String version) {
    this.version_software = version;
    this.horas_funcionando = 0;
 }
 public void Resetear() {
    this.estado = EstadoMaquina.MANTENIMIENTO;
    this.horas_funcionando = 0;
    this.estado = EstadoMaquina.APAGADA;
 }
 public String pintardisplay() {
     String vuelta;
    vuelta = "Máquina:" + this.identificador + " software:" + this.version_software
          + " horas funcionamiento:" + this.horas_funcionando + " estado:" + this.estado;
    return vuelta;
 }
```

```
enum EstadoMaquina {
   ENCENDIDA,
   APAGADA,
   AVERIADA,
   MANTENIMIENTO
}
```

Ejemplo.

Se implementa la clase cinta de correr, que además de ser una máquina ha de tener la funcionalidad siguiente:

Programas. Cada programa se define como un vector de pares que tiene tiempo y potencia.

Inclinación. Angulo de inclinación.

Velocidad: Velocidad actual.

Tiemposesion. Cuanto tiempo lleva el usuario o usuaria.

Subirvelocidad()

Bajarvelocidad();

Subirinclinacion();

BajarInclinacion();

ParadaEmergencia();

Ejemplo.

```
public class Cintacorrer extends Maquina {
  private ProgramaCinta programas[];
  private int velocidad;
  private int inclinacion;
  private int minutos_sesion;
  private ProgramaCinta programa_actual;
  public Cintacorrer() {
    this.inclinacion = 0;
    this.velocidad = 0;
    this.setEstado(EstadoMaquina.APAGADA);
    this.programa_actual = null;
    System.out.println("Soy el constructor de la cinta");
  }....
```

```
public class Principal {
    public static void main (String args[]){
        Cintacorrer cinta= new Cintacorrer();
        System.out.println(cinta.pintardisplay());
    }
}
```

Al ejecutar se imprime por pantalla:

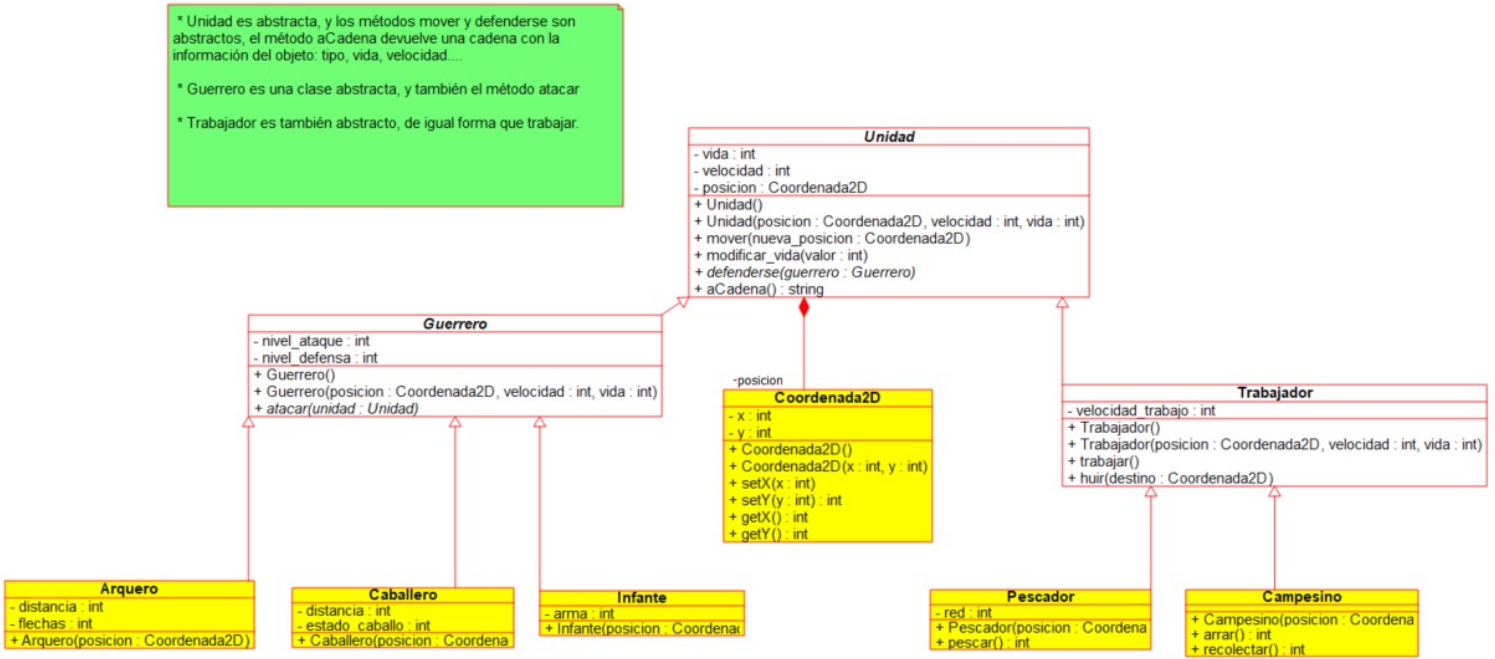
Soy el constructor por defecto de máquina

Soy el constructor de la cinta

Máquina: software:null horas funcionamiento:0 estado:APAGADA

Ejercicio.

Se ha decidido implementar parte del juego de estrategia diseñado anteriormente a partir de la solución de clase, el diagrama de clases es el siguiente:



Ejercicio.

Crear las diferentes clases siguiendo las intrucciones y en el programa principal:

- **Crear un objeto de tipo Unidad ¿Qué sucede?**
- **Crear un objeto de tipo Guerrero ¿Qué sucede? ¿Cuál es la razón?**
- **En el constructor de cada clase imprimir un mensaje con “Soy el constructor de la clase:”+clase.**
- **Crear un Arquero y un caballero. Ver los mensajes que se imprimen, explicar qué sucede en esas llamadas.**
- **Definir una variable de tipo Unidad y sobre esta hacer un new de Pescador.**
- **Ejecutar el método aCadena. ¿Qué sucede? Explicar la razón.**
- **Ejecutar ahora el método atacar sobre otra unidad ¿Qué sucede? ¿Cuál es la razón?**
- **Definir una variable de tipo Guerrero y crear un Campesino sobre esta. ¿És posible? Explicar.**
- **Sobre la variable Guerrero anterior asignar un nuevo Arquero, llamar al método aCadena. ¿Qué sucede?**
- **Ejecutar en la variable de tipo Guerrero el método atacar a una unidad.**
- **Sobre la variable anterior asignar un nuevo caballero y volver a atacar. ¿És posible hacer esto? Indicar la razón.**

IMPORTANTE.

Es conveniente llamar al constructor de la clase base con super() o en caso de querer llamar a un constructor sobrecargado, super y los parámetros, a pesar de que se llama al constructor por defecto.

Super ha de ser la primera instrucción del constructor de la clase que hereda, en caso de que no se incluya esta llamada, Java lo incluye de forma automática y en caso de que la clase base no disponga de ese contructor dará error.

```
Exception in thread "main" java.lang.RuntimeException: Uncompilable source code
- Erroneous sym type: pedro.ieslaencanta.com.maquinagimnasio.Maquina.<init>

at
pedro.ieslaencanta.com.maquinagimnasio.Cintacorrer.<init>(Cintacorrer.java:19)

at Principal.main(Principal.java:15)
```

Restricciones a la herencia.

- **Se puede limitar la posibilidad de heredar de una clase concreta con la palabra reservada final en la definición de la clase.**
- **Implica que tampoco se podrán reescribir los métododos de la clase padre.**

A partir de Java 15, se definen las clases "sealed" en las que se puede indicar de forma explícita las clases a las que se permite heredad.

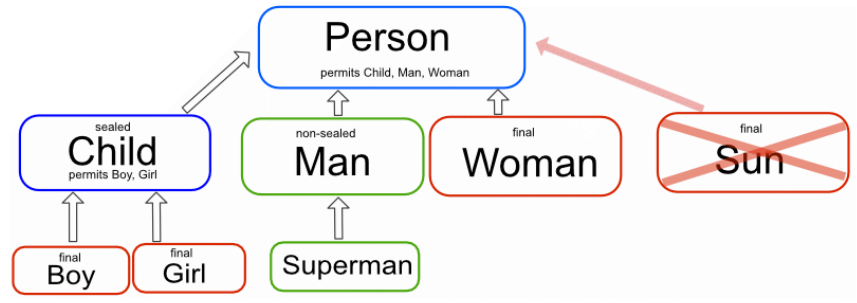
```
Public sealed class Shape permits Circle, Rectangle, Square{…..
```

```
Public class Circle extends Shape{…..
```

~~`Public class Poligon extends Shape {----`~~

Ejercicio.

- **Cambiar JDK a 15 o superior.**
- **Permitir que solo Arquero, Caballero e Infante hereden de Guerrero.**
- **Permitir que solo Pescador y Campesino hereden de Trabajador.**
- **Introducir un nuevo tipo de trabajador: Leñador y ver qué sucede.**
- **¿Como hacer para que leñador puede ser un trabajador?**
- **Crear ahora la clase Pescador_rio y Pescador_mar que heredan de Pescador, compilar.**
- **Limitar que no se pueda heredar de Pescador, volver a compilar y explicar lo sucedido.**



The `sealed class Person` can define which other classes are authorized to extend it.
`Person permits Man, Woman, Child` means that the class Person can be extended only by the classes `Man` , `Woman` and `Child` .

The extended classes need to have one of the following modifiers:

sealed : the class can be extended by the defined classes using `permits`

non-sealed : any class can extend this class

final : the class cannot be extended

Interfaces.

- **Java implementa herecia simple. Sólo se puede heredar de una clase.**
- **Necesidad heredad de más de una clase.**
- **Interfaces→ Contrato que obliga a una clase a implementar ciertos métodos públicos (no tienen sentido los privados/protegidos/defecto).**
- **Soluciona en parte los problemas de herencia simple.**
- **Solo se define el método y el tipo devuelto y los parámetros, NO el cuerpo.**
- **A partir de Java 8, posible definir métodos en las interfaces con implementación, aunque no tiene mucho sentido ya que no puede hacer referencia a atributos (la interaz no posee atributos).**

```
default String mensaje(){ return "reproductor…."};
```

- **También métodos estáticos con su implementación.**

```
public static int metodo_estadico(int valor){
     return valor*2;
 }
```

Interfaces.

- **Una clase puede implementar más de una interfaz.**

```
public class Ship extends Element implements Drawable, Movable{
  ...
}
```

La clase Ship hereda de Element e implementa las interfaces Drawable y Movable, con lo que ha de definir los métodos de las dos interfaces.

```
public interface Drawable{
  public void paint (GraphicContext gc);
}
```

¿Qué métodos ha de implementar Ship para cumplir con el contrado de la interfaz Drawable?

Interfaces. Básicas.

- Base para el uso de librerías de terceros.
- Interfaz Comparable: Permite ordenar de forma automática arrays de objetos de la misma clase.
  - La clase ha de implementar la interfaz Comparable.
  - Documentación de la interfaz y el método en la API de Java.
    - ¿Cuántos métodos define la interfaz?
    - ¿Qué métodos ha de definir la clase que implemente la interfaz Comparable?
- Otras interfaces muy utilizadas en Java:
  - Serializable. Convierte un objeto a flujo de bytes. Almacenamiento y transmión por red
  - Cloneable. Para clonar objetos.
  - Observable. Eventos (temas posteriores)
  - Iterable. Recorrer colecciones de objetos con foreach,por ejemplo vectores.
  - Runnable. Permitir que un objeto se transforme en un hilo independiente en el programa.

Method Summary

All Methods | Instance Methods | Abstract Methods

Modifier and Type	Method and Description
int	`compareTo(T o)` Compares this object with the specified object for order.

Method Detail

compareTo

`int compareTo(T o)`

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

The implementor must ensure `sgn(x.compareTo(y)) == -sgn(y.compareTo(x))` for all x and y. (This implies that `x.compareTo(y)` must throw an exception iff `y.compareTo(x)` throws an exception.)

The implementor must also ensure that the relation is transitive: `(x.compareTo(y)>0 && y.compareTo(z)>0)` implies `x.compareTo(z)>0`.

Finally, the implementor must ensure that `x.compareTo(y)==0` implies that `sgn(x.compareTo(z)) == sgn(y.compareTo(z))`, for all z.

It is strongly recommended, but *not* strictly required that `(x.compareTo(y)==0) == (x.equals(y))`. Generally speaking, any class that implements the `Comparable` interface and violates this condition should clearly indicate this fact. The recommended language is "Note: this class has a natural ordering that is inconsistent with equals."

In the foregoing description, the notation `sgn(`*expression*`)` designates the mathematical *signum* function, which is defined to return one of `-1`, `0`, or `1` according to whether the value of *expression* is negative, zero or positive.

Parameters:

`o` - the object to be compared.

Returns:

a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Throws:

`NullPointerException` - if the specified object is null

`ClassCastException` - if the specified object's type prevents it from being compared to this object.

Interfaces. Ejercicio.

- **Heredar de la clase String, por ejemplo con la clase**
- **Definir el método distance (int x, int y) y el método distance(Point2D p) que calcula la distancia entre 2 puntos.**
- **Definir un vector de Point2D (al menos 3) desordenados.**
- **Implementar en la clase Point2D la interfaz Comparable.**
- **Usar el método Array.Sort para ordenar el vector.**

```
public class Point2D {
    private int x;
    private int y;
    public Point2D(int x,int y){
        this.x=x;
        this.y=y;
    }
    public Point2D(){
        this.x=-1;
        this.y=-1;
    }
    public double distance(int x,int y){
        return Math.sqrt((this.y-y)*(this.y-y)+
(this.x-x)*(this.x-x));
    }
    public double distance(Point2D p){
        return Math.sqrt((this.y-p.y)*(this.y-
p.y)+(this.x-p.x)*(this.x-p.x));
    }
}
```

Interfaces. Ejercicio.

- **Sobreescribir el método toString para poder ver la ordenación por la salida estándar.**
- **La salida del programa anterior es:**

```
P[0]:X:1 Y:1
P[1]:X:2 Y:2
P[2]:X:5 Y:5
P[3]:X:10 Y:10
```

```
 public static void main(String[] args) {
        Point2D puntos[]= new Point2D[4];
        puntos[0]= new Point2D(2,2);
        puntos[1]= new Point2D(10,10);
        puntos[2]= new Point2D(5,5);
        puntos[3]=new Point2D(1,1);
        Arrays.sort(puntos);
       for(int i=0;i<puntos.length;i++){
           System.out.println("P["+i+"]:"+puntos[i]+"
");
        }
    }
```

Interfaces. Definiendo interfaces.

- **Se pueden crear nuevas interfaces según las necesidades.**
- **La sintaxis es:**

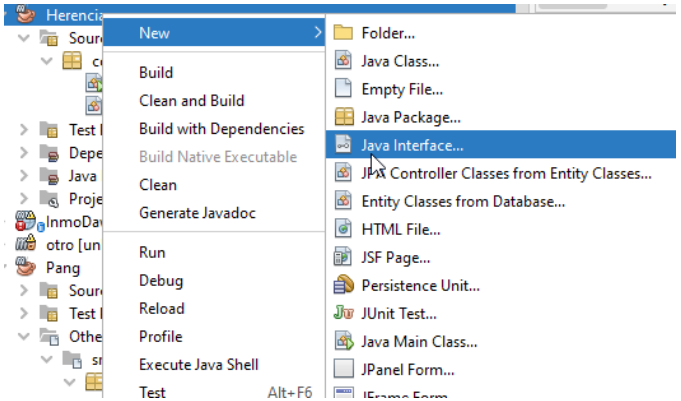
```
Public interface NOMBRE_INTERFAZ {

    public tipo_devuelto nombre_método1(parámetros);

    public tipo_devuelto nombre_método2(parámetros);

    …

}
```



- **En NetBeans:**
  - **Botón derecho sobre el proyecto→ New → Java Interface**

Interfaces. Definiendo interfaces. Ejercicio

- **Sobre el ejercicio Point2D:**
  - **Definir la interfaz “Printable”, con únicamente un método public void print().**
  - **Hacer que la clase Point2D implemente además de Comparable, la interfaz Printable.**
  - **En el “for” no usar System.out.println… sino usar la interfaz “Printable” para mostrar los datos del punto.**
  - **Si se tienen diferentes figuras como Rutas, círculos, cuadrados...¿Qué han de cumplir para que se puedan imprimir de la misma forma que Point2D?**

```
public static void main(String[] args) {
        Point2D puntos[]= new Point2D[4];
        puntos[0]= new Point2D(2,2);
        puntos[1]= new Point2D(10,10);
        puntos[2]= new Point2D(5,5);
        puntos[3]=new Point2D(1,1);
        Arrays.sort(puntos);
       for(int i=0;i<puntos.length;i++){
           puntos[i].print();
       }
    }
```

Clases y métodos abstractos

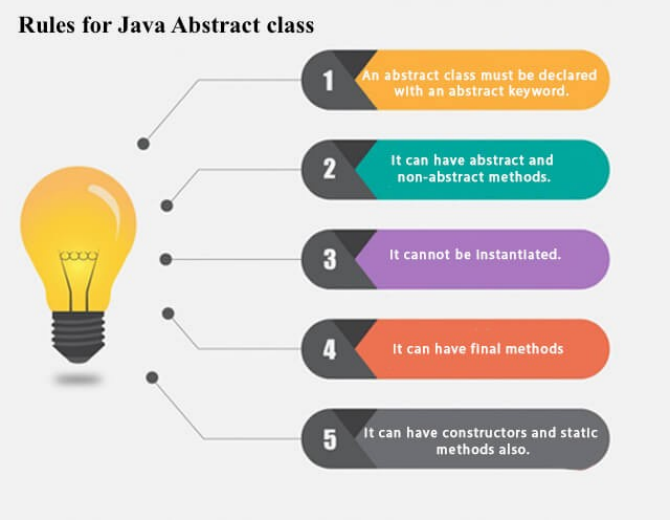
- **Principios POO.**
  - **Abstracció.**
  - **Reutilización de código.**
  - **Herencia.**
- **Cumplir con lo principios→ Clases que no tiene sentido lógico que se puedan instanciar.**
- **Ejemplos:**
  - **Vehiculo, coche, moto, bicicleta. No tiene sentido tener un objeto de tipo vehiculo.**
  - **Figura, círculo, rectangulo, triángulo. No tiene tampoco sentido tener un objeto de tipo figura.**
  - **Aeronave, avión, dron, helicoptero. Tampoco tiene sentido tener una aeronave.**

No tiene lógica tener un objeto de los tipos anteriores, pero *si se pueden tener variables y atributos de ese tipo* (variables polimorficas, se trata posteriormente).

Clases y métodos abstractos

- **Un método es abstracto si se declara con la palabra reservada abstract y no tiene implementación.**
- **Una clase es abstracta si se declara como abstracta o alguno de sus métodos es abstracto.**
- **El primer descenciente NO ABSTRACTO que herede de una o más clases abstractas ha de implementar TODOS los métodos abstractos.**

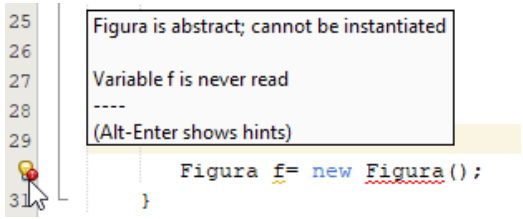
```
public abstract class Figura {
    protected String name;
    protected int x, y;
    public Figura(){
     this.x=-1;
     this.y=-1;
     this.name="Figura";
    }
    public void moverA(int x, int y){
     this.x=x;
     this.y=y;
    }
    public abstract String aSVG();
    public abstract void pintar();
}
```



Clases y métodos abstractos

- Al intentar instanciar un objeto de esa clase abstracta, el compilador devuelve un error.
- En caso de usar un IDE, muestra el error.

```
com/mycompany/herencia/Herencia.java:[30,18] com.mycompany.herencia.Figura is abstract; cannot be instantiated
1 error
```



Vídeo de la UPV sobre herencia en Java.

Clases y métodos abstractos.

Heredar de clases abstractas.

- Si la que hereda es abstracta, no se implementa nada.
- Si la que hereda NO es abstracta, ha de implementar TODOS los métodos abstractos de sus antesores. (Puede existir una jerarquía de herencia).

```
public class Rectangulo extends  Figura{
    protected int alto;
    protected int ancho;
    public Rectangulo(){
        super();
         this.name="Rectangulo";
    }
    public Rectangulo(int alto,int ancho, int x,int y){
        super();
        this.alto=alto;
        this.ancho=ancho;
        this.x=x;
        this.y=y;
       this.name="Rectangulo";
    }
    @Override
    public String aSVG(){
        return "<rect x='"+this.x+"'  y='"+this.y+"' width='"+this.ancho+"0'
height='"+this.alto+"' style='fill:rgb(0,0,255);stroke-
width:3;stroke:rgb(0,0,0)' />";
    }
    @Override
    public void pintar() {
          }
}
```

Polimorfismo.

Llamada a métodos que se denominan igual en diferentes tipos de objetos, **pero que su comportan de diferente.**

- **Su implementación no es la misma**.
- **Comparten un punto común en la herencia o interfaces.**
- **La variable o atributo a de ser el punto común**

```
Figura f;


f=new Circulo(44,56,55);
System.out.pritnln("El área es "+f.calcularArea());
f=new Cuadrado(66,77,20);
System.out.pritnln("El área es "+f.calcularArea());
f=new Elipse(46,177,20,30);
System.out.pritnln("El área es "+f.calcularArea());
```

¿Qué método se ejecuta en cada momento?

Polimorfismo. Clase object.

En Java toda clase hereda de la clase base Object.

Útil para gestionar objetos de forma sencilla en la VM.

La clase object implementa los siguiente métodos:

Object clone() **Boolean equals()**

Void finalice() **Class<?> getClass()**

Int hashCode() Void notify()

Void notifyAll() **String toString()**

Void wait() Void wait (long timeout)

Void wait(long timeout, int nanos)

Method Summary

Methods

Modifier and Type	Method and Description
protected Object	clone() Creates and returns a copy of this object.
boolean	equals(Object obj) Indicates whether some other object is "equal to" this one.
protected void	finalize() Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class<?>	getClass() Returns the runtime class of this Object.
int	hashCode() Returns a hash code value for the object.
void	notify() Wakes up a single thread that is waiting on this object's monitor.
void	notifyAll() Wakes up all threads that are waiting on this object's monitor.
String	toString() Returns a string representation of the object.
void	wait() Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
void	wait(long timeout) Causes the current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.
void	wait(long timeout, int nanos) Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.

Polimorfismo. Clase object. Método clone

Permite hacer copias de los objetos.

Por defecto es un método protected, si se quiere hacer público se ha de sobreescribir.

Existe la interfaz cloneable, su uso no es obligatorio, pero si recomentable. **Es una interfaz vacia.**

Cuando se clona, por defecto, dependiendo del tipo de atributo se ha de tener en cuenta:

- **Los tipos primitivos se duplican, es decir si se tiene un atributo x con valor 5, al clonar existe otra x diferente en el objeto clonado con valor 5, si se cambia en uno de ellos no se cambia en el otro, ya que son diferentes en memoria.**
- **En los tipos complejos, objetos, se copia la referencia al objeto, por lo que un cambio en un atributo de tipo complejo en el clonado o original, se "ve" igual por el otro, ya que son el mismo.**

A este tipo de clonado se le denomina copia superficial.

Polimorfismo. Clase object. Método clone

Ejemplo clonado:

```
public static void main(String[] args) {
        Point2D p= new Point2D(5,6);
        Point2D pc= (Point2D) p.clone();
    }
```

El código no funciona. ¿Cuál es la razón?

```
public class Point2D implements Cloneable{
    private int x;
    private int y;
    public Point2D() {
        this.x = -1;
        this.y = -1;
    }
    public Point2D(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int getX() {  return x;}
    public void setX(int x) {   this.x = x;  }
    public int getY() {    return y;   }
    public void setY(int y) { this.y = y;  }


}
```

Polimorfismo. Clase object. Método clone

Ejemplo clonado:

```
public static void main(String[] args) throws
CloneNotSupportedException {
        Point2D p = new Point2D(5, 6);
        Point2D pc = (Point2D) p.clone();
        System.out.println("P->X:" + p.getX() + " Y:" + p.getY());
        System.out.println("PC->X:" + pc.getX() + " Y:" + pc.getY());
        pc.setX(46);
        System.out.println("Clonado:");
        System.out.println("P->X:" + p.getX() + " Y:" + p.getY());
        System.out.println("PC->X:" + pc.getX() + " Y:" + pc.getY());

    }
```

Salida:

```
P->X:5 Y:6
PC->X:5 Y:6
Clonado:
P->X:5 Y:6
PC->X:46 Y:6
```

```
public class Point2D implements Cloneable{
    private int x;
    private int y;
    public Point2D() {
        this.x = -1;
        this.y = -1;
    }
    public Point2D(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int getX() {  return x;}
    public void setX(int x) {   this.x = x;  }
    public int getY() {    return y;   }
    public void setY(int y) { this.y = y;  }
    @Override
    public Object clone() throws
CloneNotSupportedException{
        return super.clone();
    }
}
```

Polimorfismo. Clase object. Método clone

Ejemplo clonado 2:

```
public static void main(String[] args) throws CloneNotSupportedException {
    Triangulo t = new Triangulo();
    Triangulo tc = (Triangulo) t.clone();
    System.out.println("T[" + 0 + "]:" + "X:" + t.getVertice(0).getX() + " Y:" +
t.getVertice(0).getY());
    System.out.println("TC[" + 0 + "]:" + "X:" + tc.getVertice(0).getX() + " Y:" +
tc.getVertice(0).getY());
    t.getVertice(0).setX(7);
    System.out.println("Copia superficial:");
    System.out.println("T[" + 0 + "]:" + "X:" + t.getVertice(0).getX() + " Y:" +
t.getVertice(0).getY());
    System.out.println("TC[" + 0 + "]:" + "X:" + tc.getVertice(0).getX() + " Y:" +
tc.getVertice(0).getY());
    }
```

Salida:

```
T[0]:X:1 Y:1
TC[0]:X:1 Y:1
Copia superficial:
T[0]:X:7 Y:1
TC[0]:X:7 Y:1
```

```
public class Triangulo implements Cloneable{
    private Point2D puntos[];
    public Triangulo(){
        this.puntos= new Point2D[3];
        this.puntos[0]= new Point2D(1,1);
        this.puntos[1]= new Point2D(2,2);
        this.puntos[2]= new Point2D(3,1);
    }
    @Override
    public Object clone() throws CloneNotSupportedException{
        return super.clone();
    }
    public Point2D getVertice(int i){
        return this.puntos[i];
    }
}
```

Polimorfismo. Clase object. Método clone

Ejemplo clonado 2:

```
public static void main(String[] args) throws CloneNotSupportedException {
    Triangulo t = new Triangulo();
    Triangulo tc = (Triangulo) t.clone();
    System.out.println("T[" + 0 + "]:" + "X:" + t.getVertice(0).getX() + " Y:" +
t.getVertice(0).getY());
    System.out.println("TC[" + 0 + "]:" + "X:" + tc.getVertice(0).getX() + " Y:" +
tc.getVertice(0).getY());
    t.getVertice(0).setX(7);
    System.out.println("Copia superficial:");
    System.out.println("T[" + 0 + "]:" + "X:" + t.getVertice(0).getX() + " Y:" +
t.getVertice(0).getY());
    System.out.println("TC[" + 0 + "]:" + "X:" + tc.getVertice(0).getX() + " Y:" +
tc.getVertice(0).getY());
    }
```

Salida:

```
T[0]:X:1 Y:1
TC[0]:X:1 Y:1
Copia superficial:
T[0]:X:7 Y:1
TC[0]:X:7 Y:1
```

```
public class Triangulo implements Cloneable{
    private Point2D puntos[];
    public Triangulo(){
        this.puntos= new Point2D[3];
        this.puntos[0]= new Point2D(1,1);
        this.puntos[1]= new Point2D(2,2);
        this.puntos[2]= new Point2D(3,1);
    }
    @Override
    public Object clone() throws CloneNotSupportedException{
        return super.clone();
    }
    public Point2D getVertice(int i){
        return this.puntos[i];
    }
}
```

Efecto **aliasing** de **copia superficial**: modificación de objetos referenciados desde diferentes puntos sin desearlo.

Polimorfismo. Clase object. Método clone

Ejemplo clonado 3:

```
public static void main(String[] args) throws CloneNotSupportedException {
    Triangulo t = new Triangulo();
    Triangulo tc = (Triangulo) t.clone();
    System.out.println("T[" + 0 + "]:" + "X:" + t.getVertice(0).getX() + " Y:" +
t.getVertice(0).getY());
    System.out.println("TC[" + 0 + "]:" + "X:" + tc.getVertice(0).getX() + " Y:" +
tc.getVertice(0).getY());
    t.getVertice(0).setX(7);
    System.out.println("Copia superficial:");
    System.out.println("T[" + 0 + "]:" + "X:" + t.getVertice(0).getX() + " Y:" +
t.getVertice(0).getY());
    System.out.println("TC[" + 0 + "]:" + "X:" + tc.getVertice(0).getX() + " Y:" +
tc.getVertice(0).getY());
    }
```

Salida:

```
T[0]:X:1 Y:1
TC[0]:X:1 Y:1
Copia superficial:
T[0]:X:7 Y:1
TC[0]:X:1 Y:1
```

```
public class Triangulo implements Cloneable {
    private Point2D puntos[];
    public Triangulo() {
        this.puntos = new Point2D[3];
        this.puntos[0] = new Point2D(1, 1);
        this.puntos[1] = new Point2D(2, 2);
        this.puntos[2] = new Point2D(3, 1);
    }
    @Override
    public Object clone() throws CloneNotSupportedException {
        Triangulo t = new Triangulo();
        t.setVertice(0, (Point2D) this.getVertice(0).clone());
        t.setVertice(1, (Point2D) this.getVertice(1).clone());
        t.setVertice(2, (Point2D) this.getVertice(2).clone());
        return t;
    }
    public Point2D getVertice(int i) {
      return this.puntos[i];
    }
    public void setVertice(int i, Point2D p) {
      this.puntos[i] = p;
    }
}
```

Copia profunda. El programador/a a de decidir si quiere superficial, profunda o híbrida en función necesidades.

Polimorfismo. Clase object. Método equals

Indica si un objeto es igual a otro.

El operador == indica si referencian al mismo objeto, no si son iguales.

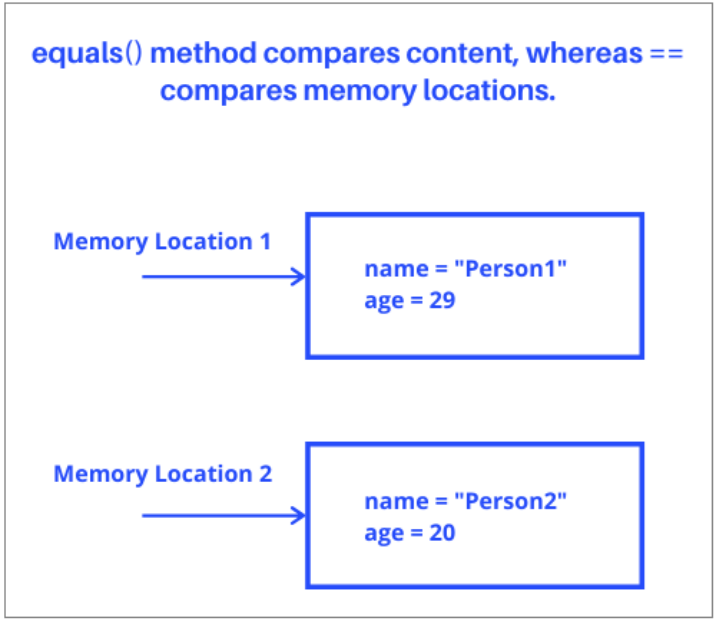
Por defecto comprueba el número “hash” devuelto por el método “hascode”.

Ejemplo con objetos con los mismos valores:

```
public static void main(String[] args) throws CloneNotSupportedException {
        Point2D p = new Point2D(5, 6);
        Point2D p2 = new Point2D(5,6);
        System.out.println("Si se usa el operador == ->"+(p==p2));
        System.out.println("Al comparar usando equals ->"+ p.equals(p2));
    }
```

Salida:

```
Si se usa el operador == ->false
Al comparar usando equals ->false
```



Polimorfismo. Clase object. Método equals

Sobreescribir el método equals.

```
public static void main(String[] args)  {
        Point2D p = new Point2D(5, 6);
        Point2D p2 = new Point2D(5,6);
        System.out.println("Si se usa el operador == ->"+
(p==p2));
        System.out.println("Al comparar usando equals ->"+
p.equals(p2));
    }
```

Salida:

```
Si se usa el operador == ->false
Al comparar usando equals ->true
```

```
public class Point2D implements Cloneable {
    private int x;
    private int y;
    public Point2D() {
        this.x = -1;
        this.y = -1;
    }
    public Point2D(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int getX() {   return x; }
    public void setX(int x) {  this.x = x; }
    public int getY() {  return y;}
    public void setY(int y) {this.y = y; }
    @Override
    public boolean equals(Object o) {
        boolean vuelta = false;
        if (!(o instanceof Point2D)) {
            vuelta = false;
        } else {
            Point2D p = (Point2D) o;
            if (this.x == p.getX() && this.y == p.getY()) {
                vuelta = true;
            }
        }
        return vuelta;
    }
}
```

Polimorfismo. Clase object. Método equals

Ejercicio:

- Crear la clase Point2D.
- Crear la clase Triangulo.
- Sobrecargar el método equals de triángulo.
- Probar el método equals.

```
public class Triangulo  {
    private Point2D puntos[];
    public Triangulo() {
        this.puntos = new Point2D[3];
        this.puntos[0] = new Point2D(1, 1);
        this.puntos[1] = new Point2D(2, 2);
        this.puntos[2] = new Point2D(3, 1);
    }
    @Override
    public boolean equals(Object o){
        return false;
    }
    public Point2D getVertice(int i) {  return this.puntos[i]; }
    public void setVertice(int i, Point2D p) {this.puntos[i] = p; }
}
```

Polimorfismo. Clase object. Método toString

Usado cuando se concatena un objeto en una cadena.

Por defecto devuelve el paquete.nombreclase@hashobjeto.

```
public static void main(String[] args) {
        Point2D p = new Point2D(5, 6);
        System.out.println(p);
    }
```

Salida:

```
com.mycompany.clonado.Point2D@2c7b84de
```

Al sobrescribir:

```
X:5,Y:6
```

```
public class Point2D {
     private int x;
    private int y;
    public Point2D() {
        this.x = -1;
        this.y = -1;
    }
    public Point2D(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int getX() {
        return x;
    }
    public void setX(int x) {
        this.x = x;
    }
    public int getY() {
        return y;
    }
    public void setY(int y) {
        this.y = y;
    }
    @Override
    public String toString(){
        return "X:"+this.x+",Y:"+this.y;
    }
}
```

Polimorfismo. Clase object. Método GetClass

Obtener información sobre la **clase,** no sobre el objeto.

Nombre clase.

Clases de las que hereda,

Obtener método constructor.

Listado de métodos.

Interfaces.

Atributos...

Crear nuevos objetos a partir de getClass:

```
Point2D p = new Point2D(5, 6);
Point2D  m=p.getClass().getDeclaredConstructor().newInstance();
System.out.println(m);
```

Ver los métodos de una clase:

```
Point2D p = new Point2D(5, 6);
      Class<? extends Point2D> c = p.getClass();

      for (int i = 0; i < c.getMethods().length; i++) {
          System.out.println("Método:" + c.getMethods()[i].getName());
      }
```

Salida:

```
Método:toString
Método:getX
Método:getY
Método:setX
Método:setY
Método:wait
Método:wait
Método:wait
Método:equals
Método:hashCode
Método:getClass
Método:notify
Método:notifyAll
```

Polimorfismo. Clase object. Método HashCode

Obtiene a partir del objeto y aplicando una función “HASH” un entero.

Dado dos objetos que son “iguales” han de devolver la misma función “HASH”.

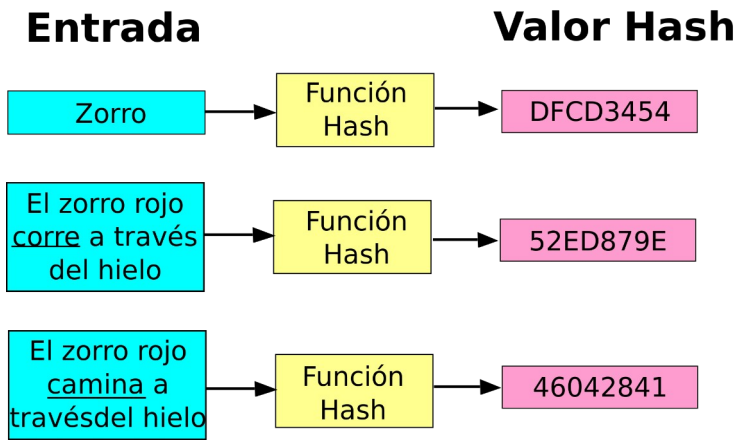
Optimiza la comparación, “equals”.

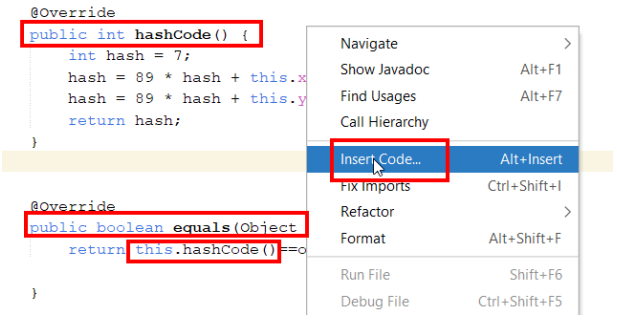
Difícil obtener un método único.

```
@Override
   public int hashCode(){
       //¿problemas?
       return this.x*1000+this.y;
   }
```

Los IDE’s generan métodos “hash” y “equals” de forma automática.

```
@Override
   public int hashCode() {
       int hash = 7;
       hash = 89 * hash + this.x;
       hash = 89 * hash + this.y;
       return hash;
   }
```





Polimorfismo. Variables polimorficas.

- Definir variables o atributos de la clase base o interface.
- Referenciar a objetos de clases que heredan o implementan clases/interfaces de las variables o atributos anteriores.
- Poseer conjunto de variables del tipo base/interfaz pero que son de clases derivadas/implementadas.
- Sólo se puede acceder a métodos de la clase base/interfaz de la que se declaro la variable/atributo/vector.
- Cada método se llama igual, pero su implementación puede ser diferente.

Ejemplo:

Se tiene un vector de la clase abstracta "Elemento" que posee un método "move()", en ese vector se almacenan: Jugador, EnemigoTonto y EnemigoListo, que heredan de "Elemento" y por tanto han de implementar el método "move" pero cada uno de ellos implementará de forma diferente el método "move".

Cuando se recorre el vector para mover, se llama en cada objeto almacenado a "move", siendo indiferente la implementación. Cada elemento ejecutara su código propio, es decir se moverá de forma diferente.

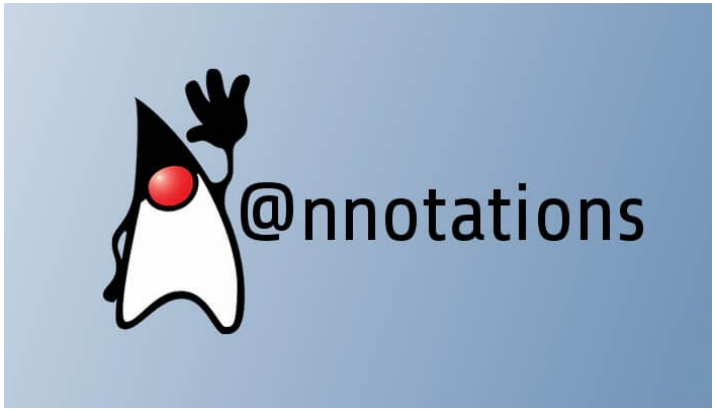
Polimorfismo. Llamar a métodos clase padre.

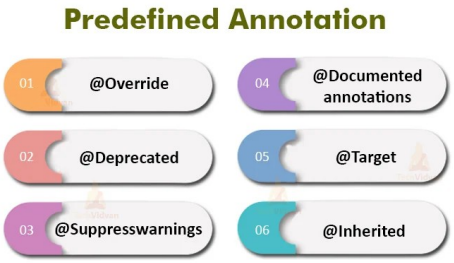
Constructor: super() super(parámetros) Otros métodos: super.nombre(parámetros)

TEMA 5. Herencia. Java→Anotaciones. @

Metadatos.

- Sintáxis: @Nombreanotación.
- Añade informoración que se puede usar en:
  - En el código. SOURCE.
  - Tiempo compilación. CLASS
  - Tiempo ejecución. RUNTIME
- Formato:
  - @nombreanotacion(parámetro=”valor”, parámetro=”valor”)
- Usados en lenguajes más populares: Java, C# ([ ]), PHP…






```
0 references | 0 changes | 0 authors, 0 changes
public class Persona
{
    [Required]
    0 references | 0 changes | 0 authors, 0 changes
    private string Name { get; set; }
    [Required]
    0 references | 0 changes | 0 authors, 0 changes
    private string LastName { get; set; }
    [Required]
    0 references | 0 changes | 0 authors, 0 changes
    private string Address { get; set; }
}
```

```
<?php

namespace App\Controller;


use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\Routing\Annotation\Route;

class HomeController extends Controller
{
    /**
     * @Route('/')
     */
    public function home(Request $request)
    {
        return $this->render( view: 'home/home.html.twig');
    }
}
```

Metadatos.

- Ejemplos anotaciones java.lang:
  - @Deprecated, indica que ese método está en desuso y puede ser que en próximas versiones ya no se encuentre disponible.
  - @Override, informa al compilador de que ha de sobreescribir el método en la clase que hereda.
  - @SafeVarargs, es posible pasar un conjunto indeterminado de parámetros a un método, esto provoca un aviso por parte del compilador, para evitar que se produzca este aviso se usa esta anotación.
  - @FunctionalInterface. Interfaces funcionales y funciones lambda (siguiente tema). Esta anotación indica que la interfaz posee un solo método abstracto y que es una interfaz funcional.
  - @SuppressWarnings. Desactiva para ese método o variables algunos avisos de compilación. Se le puede dar parámetros: all, boxing, cast...

```
@SafeVarargs
    public final void safe(T... toAdd) {
        for (T version : toAdd) {
            versions.add(version);
        }
    }
```

```
@SuppressWarning("unused") public void foo() {
  String s;
 }
```

TEMA 5. Herencia. Java→Anotaciones. @

Posible crear anotaciones propias, usa a su vez la anotación @interface.

Estas anotaciones poseen normas como:

- Los valores devueltos han de ser primitivos, cadenas o arrays.
- Los parámetros de los métodos pueden tener valores por defecto.

Toma de decisiones en:

- Codificación.
- Compilación.
- Ejecución.

Muchos frameworks definen sus propias anotaliciones, por ejemplo

Spring (Web)

- @Required
- @Qualifier
- @Configuration
- @Bean

```
@MetadataDefinition
    @Name("com.oracle.Severity")
    @Label("Severity")
    @Description("Value between 0 and 100
that indicates " +
        "severity. 100 is most severe.")
    @Retention(RetentionPolicy.RUNTIME)
    @Target({ ElementType.TYPE })
    public @interface Severity {
       int value() default 50;
    }
```

Ejemplo.

Generación de forma automática de formulario a partir de una clase sencilla.

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.SOURCE)
public @interface HTMLEntity {
    public String cssfile() default "";
}
```

```
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.SOURCE)
public @interface HTMLField {
    public String regularexpresion() default "";
    public boolean required() default false;
}
```

```
public class BuilderProcessor extends AbstractProcessor {
    @Override
    public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv) {
        Set<? extends Element> annotatedElements = roundEnv.getElementsAnnotatedWith(HTMLEntity.class);
        annotatedElements.forEach(htmlentity -> {
            try {
                this.writeBuilderFile(htmlentity);
            } catch (IOException ex) {
                Logger.getLogger(BuilderProcessor.class.getName()).log(Level.SEVERE, null, ex);
            }
        });
        return true;
    }
```

Ejemplo.

Generación de forma automática de formulario a partir de una clase sencilla.

```
private void writeBuilderFile(Element element) throws IOException {
        //abrir fichero
        FileObject builderFile = processingEnv.getFiler().createResource(
                StandardLocation.CLASS_OUTPUT,
                "", element.getSimpleName() + ".html");
        PrintWriter out = new PrintWriter(builderFile.openWriter());
        List<? extends Element> htmlfields = element.getEnclosedElements();
        //insertar los elementos hmtl
        out.print("<html>\n"); out.print("\t<head>\n");
        out.print("\t\t<title> Forumulario " + element.getSimpleName() + "</title>\n");
         out.print("\t</head>\n");out.print("\t<body>\n");  out.print("\t\t<form>\n");
        htmlfields.forEach(h -> {
            HTMLField a = h.getAnnotation(HTMLField.class);
            if (a != null) {
                boolean requerido = a.required();
                String regex = a.regularexpresion();
                out.print("\t\t<label for='" + h.getSimpleName() + "'/>\n");
                if (h.asType().toString().equals(String.class.getCanonicalName())) {
                    out.print("<input type='text' id='" + h.getSimpleName() + "' " + (requerido ? "required" : "") + " " +
(!regex.equals("") ? "pattern='" + regex + "'" : "") + ">");
                }
```

Ejemplo.

Generación de forma automática de formulario a partir de una clase sencilla.

```
@HTMLEntity(cssfile = "./css/style.css")
public class Usuario {
    @HTMLField(required = true)
    private String nombre;
    @HTMLField(required = false)
    private int edad;
}
<html>
    <head><title> Forumulario Usuario</title></head>
    <body>
        <form>
            <label for='nombre'/>
            <input type='text' id='nombre' required >
            <label for='edad'/>
            <input type='number' id='edad' >
        </form>
    </body>
</html>
```

```
@HTMLEntity(cssfile = "./css/style.css")
public class Coche {
    @HTMLField(required = true, regularexpresion = "[a-z]+")
    String color;
    @HTMLField(required = true)
    String marca;
    @HTMLField(required = true)
    int puertas;
}
    <html>
        <head><title> Formulario Coche</title> </head>
        <body>
            <form>
                <label for='color'/>
                <input type='text' id='color' required
pattern='[a-z]+'>
                <label for='marca'/>
                <input type='text' id='marca' required >
                <label for='puertas'/>
                <input type='number' id='puertas' required>
            </form>
        </body>
    </html>
```