

# UT06. PROGRAMACIÓN ORIENTADA A OBJETOS. CLASES

# Trabajando con métodos

---

- **Parámetros actuales:** Son los argumentos que aparecen en la llamada a un método.
- **Parámetros formales:** Son los argumentos que aparecen en la cabecera del método. Reciben los valores que se envían en la llamada al método.
- Los parámetros actuales y los formales deben coincidir en número, orden y tipo.

# Trabajando con métodos

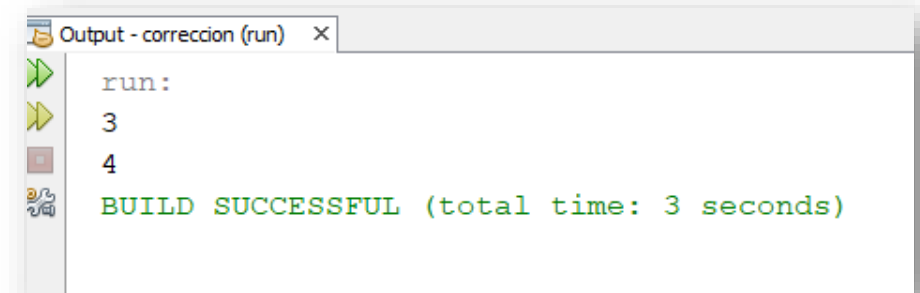
---

- Paso de parámetros por **valor** y por **referencia**.
  - Paso de **parámetros por valor** (o por copia): los parámetros se copian en las variables del método. Las variables pasadas como parámetro **no** se modifican.
  - Paso de **parámetros por referencia**: sí se pueden modificar puesto que el método trabaja con las direcciones de memoria de los parámetros.
- Cuando se pasa en Java como parámetro
  - un objeto, se pasa la **referencia** al mismo → cualquier cambio que se haga en el parámetro va a hacerse en la referencia y quedará registrado en el objeto incluso al salir del método.
  - un tipo primitivo, se pasa el **valor** → cualquier cambio que se haga en el parámetro no afectará en su correspondiente parámetro actual

# Trabajando con métodos

```
public class TestParam{  
    public static void cambiar (int x){  
        x++;  
    }  
  
    public static void cambiar2 (int[] par){  
        par[0]++;  
    }  
  
    public static void main (String[] args){  
        int x = 3;  
        int[] arrx = {3};  
  
        cambiar(x);  
        System.out.println (x);  
        cambiar2(arrx);  
        System.out.println (arrx[0]);  
    }  
}
```

¿Qué visualiza?



The screenshot shows an IDE output window titled "Output - correccion (run)". It displays the output of the Java program, which is the value of x (3) and the value of the first element of the array arrx (4). The output is as follows:

```
run:  
3  
4  
BUILD SUCCESSFUL (total time: 3 seconds)
```

# Trabajando con métodos

---

- Métodos **recursivos**
  - Cuando se llama a sí mismo.
  - Uso:
    - Cuando la resolución de un problema es más sencilla
    - Cuando no es infinita, hay un caso resoluble más básico o más sencillo.
  - En cada llamada recursiva nos acercamos más a la solución
  - La recursividad **no** es eficiente.
  - Es sencilla de programar y de entender
  - Siempre hay un método equivalente iterativo

# Trabajando con métodos

- Métodos recursivos

- Ejemplo de recursividad

- potencia(x, y)

- **Fórmula** o método que reduzca la complejidad y nos acerque a la solución

- $x^y = x * x^{y-1} \rightarrow 2^4 = 2 * 2^3$

- **Caso base** que solucione el problema

- $x^0 = 1 \rightarrow 2^0 = 1$

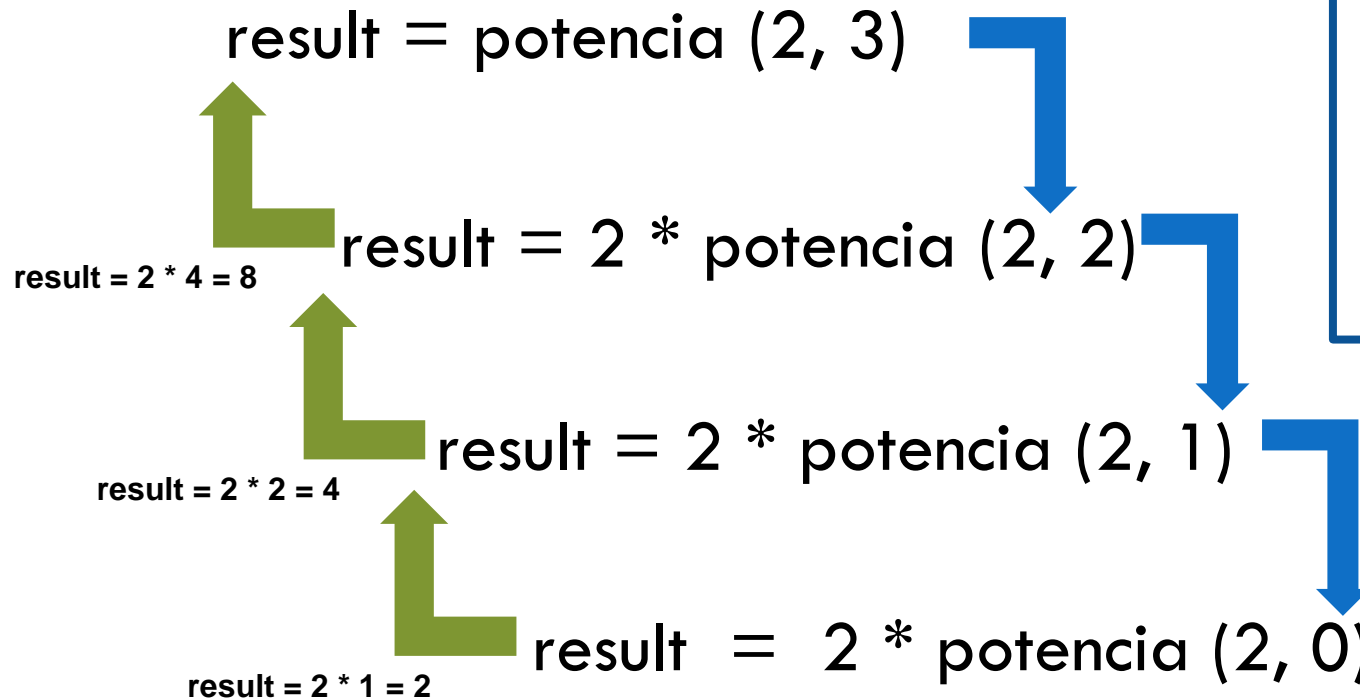
- **Código**

```
public static int potencia(int x, int y){
    int result;

    if(y==0)
        result = 1;
    else
        result = x * potencia(x, y-1);
    return result;
}
```

# Trabajando con métodos

- Ejemplo:  $2^3$

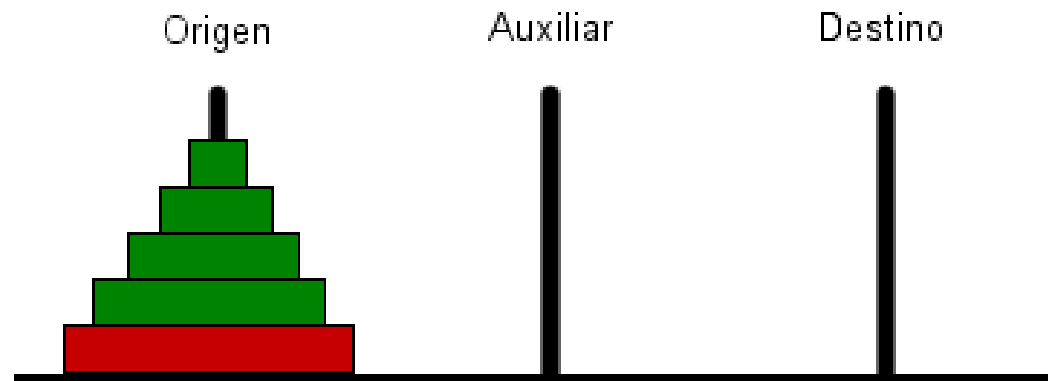


```
public static int potencia(int x, int y){  
    int result;  
  
    if(y==0)  
        result = 1;  
    else  
        result = x * potencia(x, y-1);  
    return result;  
}
```

# Trabajando con métodos

- **Ejercicios.**

1. Solucionar el factorial de un número de forma recursiva
2. Resolver la serie de Fibonacci de manera recursiva, teniendo en cuenta que cada término es la suma de los dos anteriores. El primer término es el 0 y el segundo el 1.
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89...
3. Resolver el problema de las Torres de Hanoi de forma recursiva





# Los constructores

---

- Se llama de forma automática siempre que se crea un objeto.
- Aunque se pueda llamar a otros métodos de la clase desde un constructor, no es recomendable hacerlo (los atributos pueden no estar en un **estado consistente**, se está creando el objeto)
- Tipos:
  - Por defecto
  - Definido

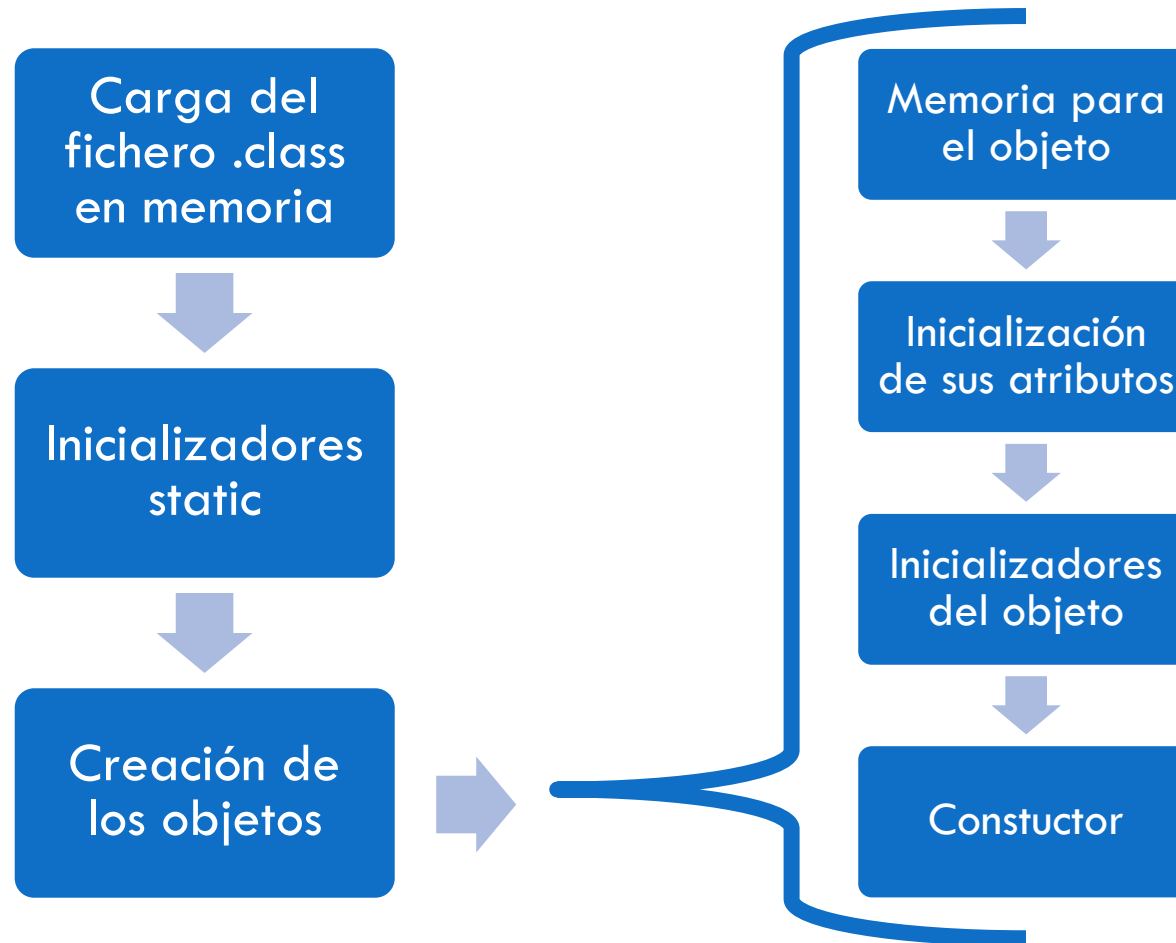
# Los constructores

---

- Por defecto
  - No se especifica en el código.
  - Se **ejecuta** siempre de manera automática.
    - Llama al **constructor sin argumentos** de su **superclase**
    - **Inicializar** las variables de instancia con los valores por defecto.
    - Si la superclase no tuviese un constructor sin argumentos ➔ **error**.
  - **Inicializa el objeto** con los valores especificados o predeterminados del sistema.
- Definido
  - Si se define constructor, ya no se puede utilizar el constructor por defecto
  - Puede ser más de uno ➔ está **sobrecargado**
  - Tiene el **mismo nombre** de la clase.
  - **Nunca devuelve** un valor
  - No puede ser declarado como **static**, **final**, **native**, **abstract** o **synchronized**.
  - Suelen ser **public**

# Los constructores

- Pasos en la creación de los objetos.



# Métodos sobrecargados

---

- **Métodos sobrecargados**
  - Se distinguen por su **firma**: combinación del **nombre** del método, el **número** y **tipo** de sus parámetros.
  - No puede haber en una misma clase dos métodos que se llamen igual y que además tengan el mismo número y tipo para sus parámetros.
  - Pueden tener el mismo o distinto tipo de valor de retorno, pero no pueden coincidir en su firma.

# Los constructores

- **Sobrecarga del constructor**
  - Inicializar un objetos de múltiples formas.
  - Variación en el número de y tipo de parámetros

```
public class Rectangulo{  
    private int ancho;  
    private int alto;  
  
    public Rectangulo(){  
        this.ancho = this.alto = 0;  
    }  
  
    public Rectangulo(int an, int al){  
        ancho=an;  
        alto=al;  
    }  
  
    public Rectangulo(int dato){  
        this.alto = this.ancho = dato;  
    }  
    //crear objetos con cada uno de los  
    constructores
```

# Los constructores

---

- **this** y **this()**
  - **this**
    - Hace referencia a los atributos de la propia clase
  - **this()**
    - El método **this()** se usa para hacer referencia, dentro de un constructor de una clase, **a otro constructor sobrecargado** de la misma clase, aquel que coincida con la lista de parámetros de la llamada.

# Constructores

```
public class Rectangulo{  
    private int ancho;  
    private int alto;  
    private String nombre;  
  
    public Rectangulo(int an, int al){  
        ancho=an;  
        alto=al;  
    }  
  
    public Rectangulo(int an, int al, String nombre ){  
        this(an, al); //Invoca al constructor anterior  
  
        this.nombre=nombre; //Evita ambigüedades entre atributo y parámetro  
    }  
}
```

# Los constructores

- **Asignación de objetos**

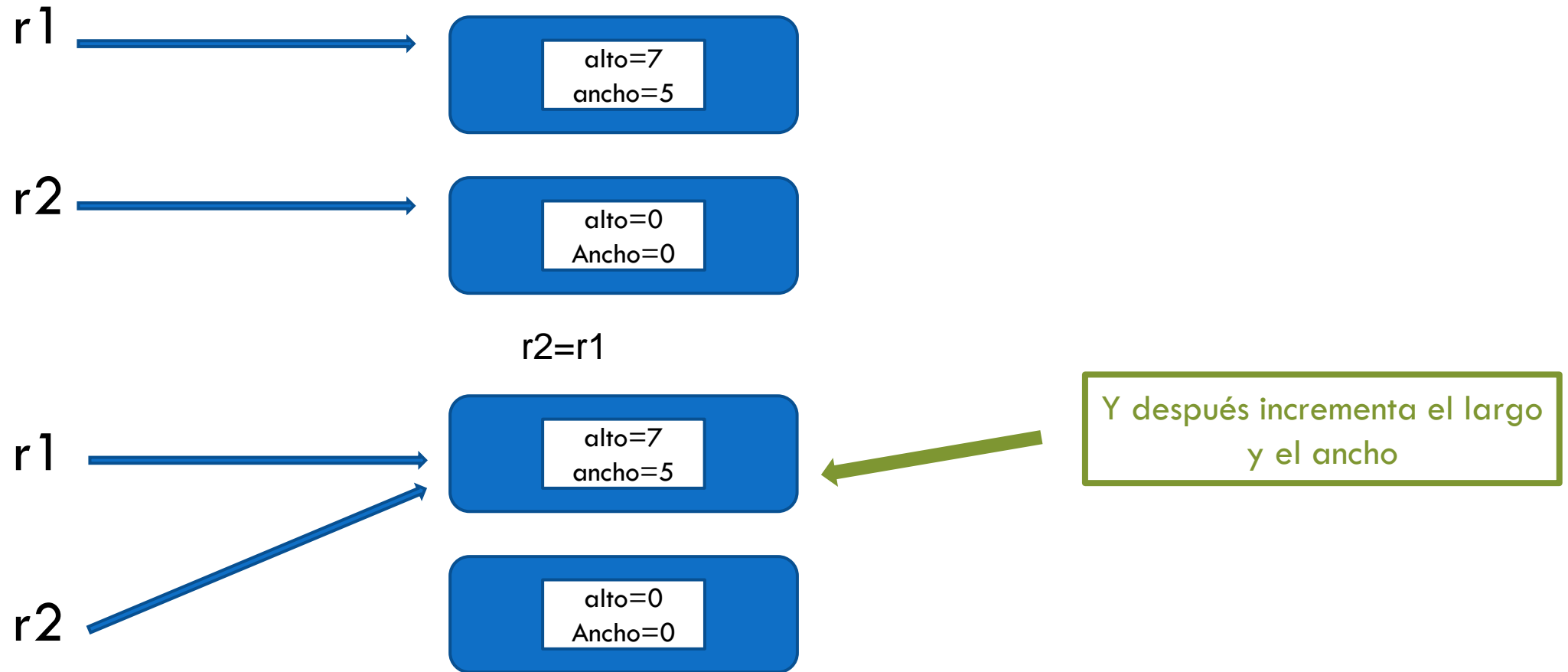
- Referencia = localización en memoria
- Cuando trabajamos con objetos, trabajamos con referencias
- ¿Qué mostrará el siguiente código por pantalla?

```
Rectangulo r1 = new Rectangulo(5, 7);  
Rectangulo r2 = new Rectangulo();  
r2 = r1;  
r2.incrementarAncho();  
r2.incrementarAlto();  
System.out.println("Alto: " + r1.getAlto());  
System.out.println("Ancho: " + r1.getAncho());
```

Opción1	Opción2
Alto: 7 Ancho: 5	Alto: 8 Ancho: 6



# Los constructores



Respuesta: Opción2

# Los constructores

- **Constructor copia**

- Un objeto se inicializa asignándole valores de otro objeto diferente de la misma clase.
- Solo un parámetro: un objeto de la misma clase.

```
public Rectangulo (Rectangulo r){  
    this.ancho = r.ancho;  
    this.alto = r.alto;  
}  
.....  
Rectangulo r1 = new Rectangulo(5, 7);  
Rectangulo r2 = new Rectangulo(r1);  
r2.incrementarAncho();  
r2.incrementarAlto();  
System.out.println("Alto: " + r1.getAlto());  
System.out.println("Ancho: " + r1.getAncho());
```

¿Qué mostrará el código anterior?

```
Alto: 5  
Ancho: 7  
BUILD SUCCESSFUL (total time: 0 seconds)
```

# Los constructores

---

- Inicializadores **static**
  - Bloque de código que se ejecutará una vez solamente cuando se utilice la clase
  - No devuelve ningún valor
  - Son métodos sin nombre
  - Ideal para inicializar objetos o elementos complicados
  - Permiten gestionar excepciones con **try...catch**
  - Se pueden crear más de un **inicializador static**, ejecutándose en el orden en el que se han definido
  - Se pueden utilizar para invocar métodos nativos o inicializar **variables static**.
  - A partir de Java 1.1 existen los inicializadores de objeto utilizados en las clases anónimas y no tienen el modificador **static**

# Los constructores

## Ejemplo de inicializador

```
public class TestInicializador{
    static{
        System.out.println("Llamada al inicializador");
    }

    static{
        System.out.println("Llamada al segundo inicializador");
    }

    TestInicializador(){
        System.out.println("Llamada al constructor");
    }

    public static void main(String[] args){
        TestInicializador t1 = new TestInicializador();
        TestInicializador t2 = new TestInicializador();
        TestInicializador t3 = new TestInicializador();
    }
}
```

Testpotencia... FactorialRec... FibonacciRe... MaximoCom... FactorialIter... Racional.java

jGRASP Messages Run I/O Interactions

----jGRASP wedge2: pid for process is 7976.

Llamada al inicializador  
Llamada al segundo inicializador  
Llamada al constructor  
Llamada al constructor  
Llamada al constructor

# Ejercicio Examen

---

- Realizar una clase, de nombre **Examen** para guardar información de un examen es: el nombre de la asignatura, el aula, la fecha y la hora. Para guardar la fecha y la hora hay que realizar dos clases: fecha y hora.
- La clase **Fecha** guarda día, mes y año. Todos los valores se reciben en el constructor por parámetro. Además, esta clase debe tener un método que devuelva cada uno de los atributos y un método **toString()**.
- La clase **Hora** guarda hora y minuto. También recibe los valores para los atributos por parámetro en el constructor, tiene métodos que devuelven cada uno de los atributos y un método **toString()**
- Escribe una aplicación que cree un objeto de tipo **Examen**, lo muestre por pantalla, modifique su fecha y hora y lo vuelva a mostrar por pantalla.

# Los destructores

- En Java no existen destructores
- El sistema gestiona elimina objetos de memoria cuando le asignamos el valor **null** a la referencia.
- **Garbage collector**
  - Se activa automáticamente cuando falta memoria
  - Se puede sugerir la activación mediante **System.gc()**
- Finalizador
  - Lo usa el sistema para finalizar objetos cuando se va a liberar memoria
  - No tiene valor de retorno, argumentos, ni static
  - Suele llamarse **finalize()** → **protected void finalize(){SOP("Adios");}**
  - Suelen usarse para cerrar ficheros, conexiones, liberar memoria,...
  - Una forma de sugerir que Java ejecute el recolector de basura y después lo ejecute  
**System.runFinalization();**  
**System.gc();**

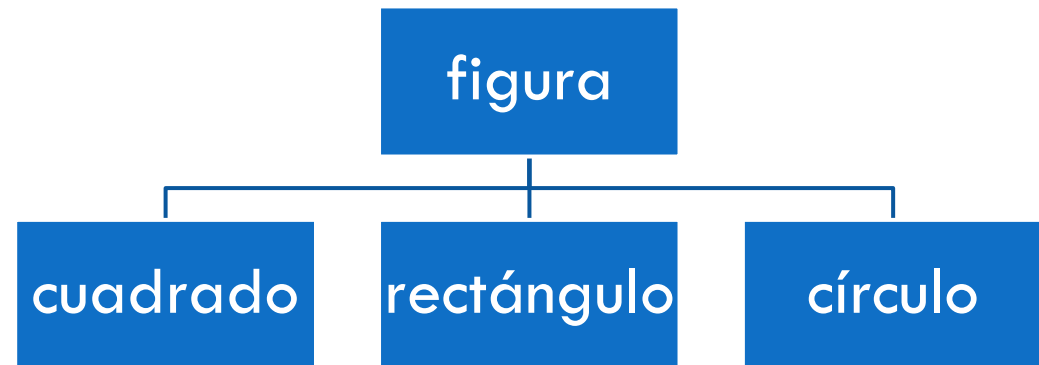
# Encapsulación y visibilidad. Interfaces

---

Se verá más adelante, en la unidad de “POO. Clases avanzadas”

# Herencia

- Es la base de la reutilización del código
- Una clase derivada hereda todos los miembros y métodos de su antecesor.
- También se puede redefinir (**@override**) los miembros para adaptarlos o ampliarlos
- No se permite herencia múltiple: heredar de varias clases





# Herencia

- Para indicar que una clase hereda de otra se usa la cláusula **extends** detrás del nombre de la clase

```
public class Rectangulo extends Figura{...}  
public class Circulo extends Figura{...}  
public class Cuadrado extends Figura{...}
```

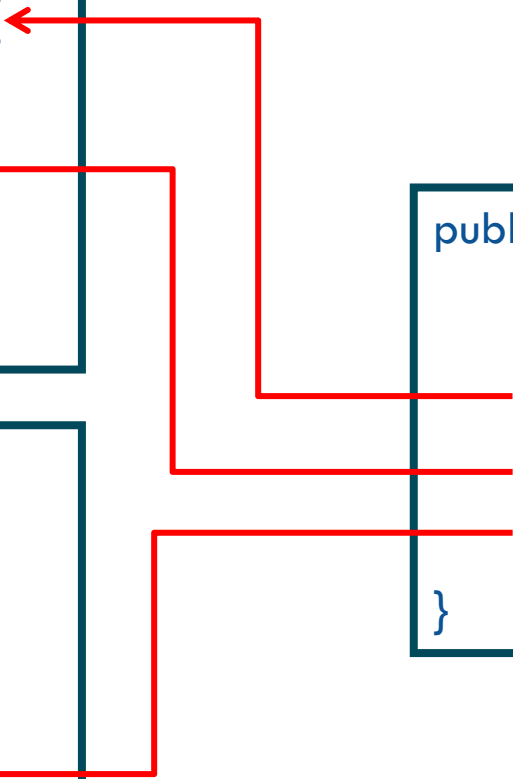
- Toda clase hereda de forma implícita de **Object**.
- Las clases heredan de sus antecesoras (padres) pero no lo heredan de otras subclases (hermanos)

# Ejemplo herencia

```
public class Figura {  
    private String color;  
    public void setColor (String color){  
        this.color = color;    }  
    public String getColor (){  
        return this.color;    }  
}
```

```
public class Cuadrado extends Figura{  
    private int lado;  
    public Cuadrado (int lado){  
        this.lado=lado;    }  
    public int getArea (){  
        return this.lado * this.lado;    }  
}
```

```
public static void main (String[] args) {  
    Cuadrado c = new Cuadrado (5);  
    c.setColor("Verde");  
    System.out.println(c.getColor());  
    System.out.println(c.getArea());  
}
```



# Herencia

- Una subclase puede **sobreescribir**
  - Un método de la superclase.
  - Un atributo de la superclase.
- Se puede hacer **más público**, pero **no más privado**.

Derecho de la clase	Puede ser en clase derivada a
privado	privado protegido de paquete público
de paquete	de paquete protegido público
protegido	protegido público
público	público

# Herencia

---

- **super**

- Hace referencia a la superclase.

```
super.imprime(); //llama al método imprime() de la superclase
```

- Para usar el constructor de la superclase en la subclase.
  - Si hubiera más de uno, solo se referencia a uno y **siempre será la primera sentencia**

```
public class Persona{  
    private String nombre;  
    private String apellidos;  
    private int añoDeNac;
```

```
    public Persona(String nombre, String apellidos, int añoNac){  
        this.nombre=nombre;  
        this.apellidos=apellidos;  
        this.añoDeNac=añoNac;  
    }  
}
```

```
public class Alumno extends Persona{  
    private String grupo;  
    private Horario horario;
```

```
    public Alumno(String nombre, String apellidos, int añoDeNac){  
        super(nombre, apellidos, añoDeNac);  
    }  
}
```

# Herencia

---

- **final**

- Impide

- Para una variable, que cambie su contenido (**constante**).
    - Para un método, que se sobrecargue.
    - Para el parámetro de un método, que el método cambie su valor.
    - Para una clase, que se herede de ella.

# Ejercicios I

---

- Escribir una clase **Multimedia** para almacenar información de objetos de tipo multimedia (películas, discos, mp3...). Esta clase contiene título, autor, formato y duración como atributos. El formato puede ser uno de los siguientes: wav, mp3, midi, avi, mov, mpg, cdAudio, y dvd. El valor de todos los atributos se pasa por parámetro en el momento de crear el objeto. Esta clase tiene, además, un método para devolver cada uno de los atributos y un método **toString()** que devuelve la información del objeto. Por último, un método **equals()** que recibe un objeto de tipo **Multimedia** y devuelve true en caso de que el título y el autor sean iguales a los del objeto que llama al método y false en caso contrario.

# Ejercicios II

---

- Escribir una clase **Película** que herede de la clase **Multimedia** anterior. La clase Película tiene, además de los atributos heredados, un actor principal y una actriz principal. Se permite que uno de los dos sea nulo, pero no los dos. La clase debe tener dos métodos para obtener los dos nuevos atributos y debe sobrescribir el método `toString()` para que devuelva, además de la información heredada, la nueva información.
- Escribir una clase **Disco** que herede de la clase `Multimedia` ya realizada. La clase Disco tiene, aparte de los elementos heredados, un atributo para almacenar el género al que pertenece (rock, pop, funk, etc). La clase debe tener un método para obtener el nuevo atributo y debe sobrescribir el método `toString()` para que devuelva, además de la información heredada, la nueva información.



# Ejercicios III

---

- Escribir una clase **Coche** de la que van a heredar **CocheCambioManual** y **CocheCambioAutomático**. Los atributos de los coches son la matrícula, la velocidad y la marcha. Para este ejercicio no se permite la marcha atrás, por tanto no se permiten ni velocidad negativa, ni marcha negativa. En el constructor se recibe el valor de la matrícula por parámetro y se inicializa el valor de la velocidad y la marcha a 0. Además tendrá los siguientes métodos:
  - **getMatricula** que devuelve el valor de la matrícula.
  - **getMarcha** devuelve el valor de la marcha.
  - **getVelocidad** devuelve el valor de la velocidad.
  - **acelerar** recibe por parámetro un valor para acelerar el coche.
  - **frenar** recibe por parámetro un valor para frenar el coche.
  - **toString** devuelve en forma de String la información del coche.
  - **cambiaMarcha** recibe por parámetro la marcha a la que se tiene que cambiar el coche. Este método es `protected`, para que puedan acceder a él las clases que heredan del Coche, pero no las clases de otros paquetes.

# Ejercicio III

---

- La Clase **CocheCambioManual** sobrescribir el método **cambiaMarcha()** y lo hace público, para que pueda ser llamado desde cualquier clase. No permite que se cambie a una marcha negativa.
- La clase **CocheCambioAutomático** sobrescribe los métodos **acelerar()** y **frenar()** para que cambie automáticamente de marcha conforme se va acelerando y frenando.
- Escribir una aplicación que pida por teclado la matrícula de un coche y pregunte si el coche es con cambio automático o no. Posteriormente, debe crear un coche con las características indicadas por el usuario y mostrarlo. Acelerar el coche 60km/h, si el coche es con cambio manual, cambiar la marcha a tercera y volverlo a mostrar