Tema 3

INTRODUCCIÓN A LA PROGRAMACIÓN ORIENTADA A OBJETOS

1. INTRODUCCIÓN

- La programación orientada a objetos es un paradigma de programación totalmente diferente al método clásico de programación, el cual utiliza objetos y su comportamiento para resolver problemas y generar programas y aplicaciones informáticas.
- Con la POO aumenta la **modularidad** de los programas y la **reutilización** de los mismos.
- Además, se diferencia de la programación clásica porque utiliza técnicas nuevas como el **polimorfismo**, el **encapsulamiento**, la **herencia**, etc.

2. INTRODUCCIÓN AL CONCEPTO DE CLASE

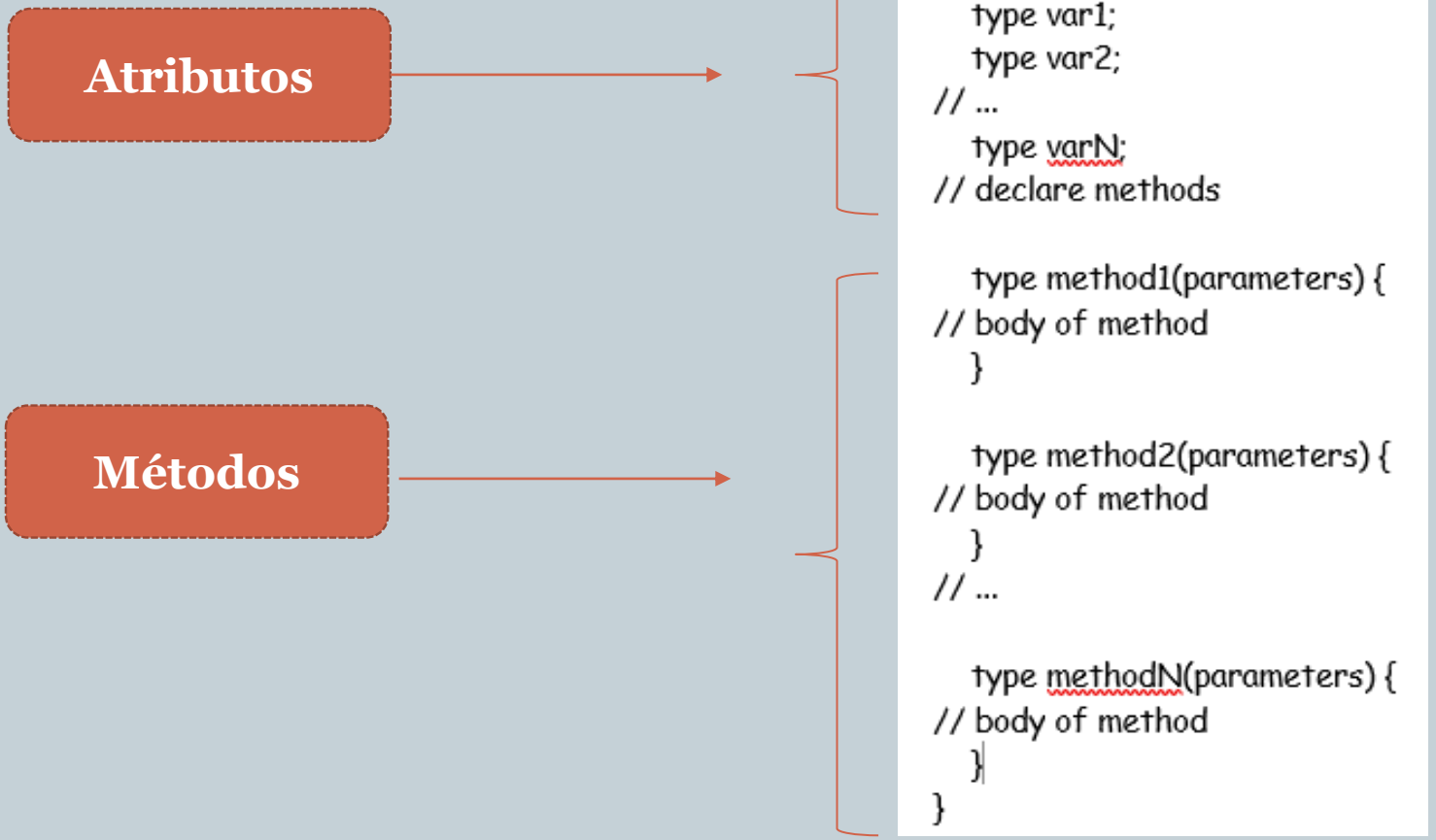
- Una **clase** es una plantilla que define la forma de un objeto.
- En la clase se definen las propiedades de los objetos que pertenecen a esa clase (**atributos**) y el comportamiento de los mismos (**métodos**).
- Java usa esa especificación de la clase para construir objetos.

3. INTRODUCCIÓN AL CONCEPTO DE OBJETO

- Un objeto es una **instancia** de una clase.
- Así, por ejemplo, podríamos definir la clase pájaro y mi loro Felipe pertenecería a dicha clase.
- Todos los objetos de la clase pájaro se identificarán, por ejemplo, con los atributos: nombre, color de plumaje, edad y si son domésticos o no.

En un símil con la costura, las clases son los patrones y los objetos son las prendas.

4. ESTRUCTURA DE UNA CLASE



5. DEFINIENDO UNA CLASE

- Veamos un ejemplo de una clase que encapsula información sobre vehículos, coches, camiones y carabanas.
- Llamaremos **Vehicle** a la clase y almacenará tres **atributos**: el número de pasajeros que es capaz de transportar, la capacidad del depósito de gasolina y el consumo medio.

```
class Vehicle {
    int passengers; // number of passengers
    int fuelcap; // fuel capacity in gallons
    int mpg; // fuel consumption in miles per gallon
}
```

5. DEFINIENDO UNA CLASE

- La definición de una clase crea un nuevo tipo de dato, en este caso, llamado Vehicle.
- Ahora podremos usar esta clase para declarar objetos de tipo vehículo.
- Recuerda que la declaración de una clase es solo un tipo de descripción, no crea un objeto.
- Por lo tanto, el código anterior no crea objetos de tipo Vehicle.
- Para crear un objeto de tipo Vehicle, usaremos la siguiente sintaxis:

```
//crea un objeto Vehicle llamado minivan
Vehicle minivan = new Vehicle();
```

5. DEFINIENDO UNA CLASE

- Cada objeto de tipo **Vehicle** contendrá su copia propia de las variables de instancia: número de pasajeros, capacidad del depósito de gasolina y el consumo medio.
- Para acceder a estas variables, usaremos el operador .
- Ejemplo: **minivan.fuelcap = 16;**

5. DEFINIENDO UNA CLASE

- Un programa Java que usa la clase Vehicle y que se llama **VehicleDemo.java** es el siguiente:

```
class VehicleDemo {

    public static void main(String args[]) {
        Vehicle minivan = new Vehicle();
        int range;

        // assign values to fields in minivan
        minivan.passengers = 7;
        minivan.fuelcap = 16;
        minivan.mpg = 21;

        // compute the range assuming a full tank of gas
        range = minivan.fuelcap * minivan.mpg;

        System.out.println("Minivan can carry " + minivan.passengers
                + " with a range of " + range);
    }
}
```

5. DEFINIENDO UNA CLASE

- Como ya hemos comentado, cada objeto tiene su propia copia de las variables de instancia definidas en su clase:

```
class TwoVehicles {

    public static void main(String args[]) {
        Vehicle minivan = new Vehicle();
        Vehicle sportscar = new Vehicle();
        int range1, range2;

        // assign values to fields in minivan
        minivan.passengers = 7;
        minivan.fuelcap = 16;
        minivan.mpg = 21;

        // assign values to fields in sportscar
        sportscar.passengers = 2;
        sportscar.fuelcap = 14;
        sportscar.mpg = 12;

        // compute the ranges assuming a full tank of gas
        range1 = minivan.fuelcap * minivan.mpg;
        range2 = sportscar.fuelcap * sportscar.mpg;
        System.out.println("Minivan can carry " + minivan.passengers
                + " with a range of " + range1);
        System.out.println("Sportscar can carry " + sportscar.passengers
                + " with a range of " + range2);
    }
}
```

6. CÓMO SE CREAN LOS OBJETOS

- Para crear un objeto se siguen los siguientes pasos:
  - Se declara la variable que va a alojar al objeto y que será del mismo tipo que la clase a la que pertenece el objeto.
  - Se reserva espacio en memoria para almacenar el objeto haciendo uso del operador new.
- El operador new reserva (en tiempo de ejecución) memoria dinámica para alojar el objeto y devuelve una referencia al mismo.
- Esta referencia es la dirección de memoria donde se va a almacenar al objeto.

```
Vehicle minivan; // declare reference to object
minivan = new Vehicle(); // allocate a Vehicle object
```

ACTIVIDAD 1

- Crea una clase llamada *Persona* con los siguientes atributos:
  - *nombre*
  - *edad*
  - *dni*
  - *sexo*
  - *peso*
  - *altura*
- Crea dos objetos de la clase Persona, inicializa sus atributos e imprime sus valores posteriormente
- **Extra**: busca cómo puedes darle un valor por defecto a los atributos de las clases (y compruébalo).

7. MÉTODOS

- Son subrutinas que operan sobre los datos definidos en la clase.
- Permiten que otras partes del programa interactúen con una clase a través de ellos.
- La estructura general de un método es:

```
return-type nameOfMethod( parameter-list ) {
    // body of method
}
```

7. MÉTODOS

- Ejemplo: añadimos un método a la clase Vehicle llamado range, que no devuelve nada y calcula

```
class Vehicle {
    int passengers; // number of passengers
    int fuelcap; // fuel capacity in gallons
    int mpg; // fuel consumption in miles per gallon

// Display the range.

    void range() {
        System.out.println("Range is " + fuelcap * mpg);
    }
}
```

7.1 DEVOLUCIÓN DE VALORES

- Un método puede devolver cero o un valor al punto desde donde fue llamado.
- Si el método no devuelve nada, en ***return-type*** se indica **void**.
- Sin embargo, es común que los métodos devuelvan o bien un valor calculado en el cuerpo del mismo ó que simplemente informen sobre el éxito o fracaso de una operación (true/false).
- En este caso, en ***return-type*** se indica el tipo de dato del dato que va a devolver.

7.1 DEVOLUCIÓN DE VALORES

- Ejemplo:

```
// Use a return value.
class Vehicle {
    int passengers; // number of passengers
    int fuelcap; // fuel capacity in gallons
    int mpg; // fuel consumption in miles per gallon

// Return the range.
    int range() {
        return mpg * fuelcap;
    }
}
```

7.2 PASO DE PARÁMETROS

- Es posible pasar uno o más valores de entrada a un método cuando éste es invocado.
- El valor que se le pasa al método en su llamada se le denomina ***argumento***.
- Dentro del método, la variable que recibe el valor del argumento se le llama ***parámetro***.
- Los parámetros son declarados dentro de los paréntesis que siguen al nombre del método.
- La forma de declarar los parámetros es la misma que hemos usado para declarar variables (**tipo nombre**).
- **Los parámetros sólo existen dentro del método**

7.2 PASO DE PARÁMETROS

- Ejemplo:

```
class Factor {
    boolean isFactor(int a, int b) {
        if ((b % a) == 0) {
            return true;
        } else {
            return false;
        }
    }
}
```

7.2 PASO DE PARÁMETROS

- Si añadimos un método parametrizado a la clase ***Vehicle***:

```
class Vehicle {

    int passengers; // number of passengers
    int fuelcap; // fuel capacity in gallons
    int mpg; // fuel consumption in miles per gallon

    // Return the range.
    int range() {
        return mpg * fuelcap;
    }

    // Compute fuel needed for a given distance.
    double fuelneeded(int miles) {
        return (double) miles / mpg;
    }
}
```

ACTIVIDAD 2

- Crea una clase *Calculadora:*
  - Tendrá un método *calcular* que recibe una operación (“x”, “/”, “+”, “-”) y dos operandos y almacena el resultado en un atributo *resultado*
  - Tendrá otro método *mostrar* que muestra el resultado por pantalla
- Crea un programa que utilice la clase anterior y pida al usuario una operación y dos operandos y muestre por pantalla el resultado
- **Extra:**
  - Devuelve un error en caso de que la operación no pueda ser realizada
  - Muestra el resultado de la siguiente forma:
    - *El resultado de la operación: 4 x 3 = 12*
  - Muestra todos los resultados que tienen decimales con exactamente 3 decimales
  - Crea un método independiente para cada operación y un método principal que los invoque. Realiza la multiplicación y la división sin usar / o *

8. CONSTRUCTORES

- Un constructor inicializa un objeto cuando éste es creado.
- Tiene el mismo nombre que la clase y es sintácticamente similar a un método.
- Sin embargo, los constructores **no devuelven valores**.
- Normalmente, se usan los constructores para inicializar las variables de instancia definidas en la clase.
- Todas las clases tienen mínimo un constructor, el cual puede ser definido por el programador o no, ya que Java automáticamente prove un constructor que inicializa todas las variables de instancia a los valores por defecto (cero para valores numéricos, null para los objetos y false para los booleanos).
- Sin embargo, si el programador define un constructor, el constructor por defecto es ignorado.

8. CONSTRUCTORES

- Ejemplo:

```
class MyClass {
    int x;
    MyClass() {
        x = 10;
    }
}

class ConstructorDemo {
    public static void main(String args[]) {
        MyClass t1 = new MyClass();
        MyClass t2 = new MyClass();
        System.out.println(t1.x + " " + t2.x);
    }
}
```

8.1 CONSTRUCTORES PARAMETRIZADOS

- Muchas veces es necesario pasarle parámetros al constructor.
- Para ello, se usa la misma sintaxis que la ya vista para los métodos.

```
class MyClass {
    int x;

    MyClass() {
        x = 10;
    }

    MyClass(int i) {
        x = i;
    }
}
```

8.1 CONSTRUCTORES PARAMETRIZADOS

- Añadimos un constructor a la clase ***Vehicle***:

```
class Vehicle {

    int passengers; // number of passengers
    int fuelcap; // fuel capacity in gallons
    int mpg; // fuel consumption in miles per gallon

    Vehicle(int p, int f, int m) {
        passengers = p;
        fuelcap = f;
        mpg = m;
    }

    // Return the range.
    int range() {
        return mpg * fuelcap;
    }

    // Compute fuel needed for a given distance.
    double fuelneeded(int miles) {
        return (double) miles / mpg;
    }
}
```

9. RECOLECTOR DE BASURA

- Como hemos comentado, los objetos se crean dinámicamente en tiempo de ejecución cuando se utiliza el operador **new**.
- En ese momento, se reserva un espacio de memoria para alojar el objeto.
- Sin embargo, la memoria no es infinita y puede llegar a agotarse y hacer que la creación de un objeto falle si no hay espacio suficiente para alojar el objeto deseado.
- Por esta razón, un componente clave de cualquier asignación de memoria dinámica es la recuperación de memoria cuando los objetos dejan de ser utilizados.
- En algunos lenguajes de programación, el programador es el encargado de liberar la memoria manualmente.
- Sin embargo, en Java existe el **recolector de basura** (*garbage collection)*.

9. RECOLECTOR DE BASURA

- El recolector de basura de Java libera automáticamente la memoria de los objetos que dejan de ser utilizados, haciendo transparente este proceso al programador.
- El funcionamiento es el siguiente: cuando deja de haber referencias a un objeto existente, se assume que no va a ser necesitado más veces y la memoria ocupada por el mismo es liberada.
- Esa memoria liberada puede ser utilizada posteriormente para alojar a otro objeto.

10. EL MÉTODO FINALIZE()

- Es posible definir un método que será llamado justo antes de la destrucción final del mismo por el recolector de basura.
- Este método es el método **finalize( )** y puede ser utilizado para asegurarnos que un objeto finaliza de manera limpia.
- Por ejemplo, se debe usar para asegurarnos que un fichero abierto por un objeto es cerrado por el mismo antes de su destrucción.

10. EL MÉTODO FINALIZE()

- Para añadir un finalizador a la clase, basta con definer el método finalize( ).
- La máquina virtual de Java llamará a este método antes de eliminar definitivamente el objeto y ejecutará el código existente dentro del mismo.
- La cabecera es la siguiente:

```
protected void finalize( ){
// finalization code here
}
```

10. EL MÉTODO FINALIZE()

- Ejemplo:

```
public class EjemploFinalize {

    private int x;

    public EjemploFinalize(int x) {
        this.x = x;
        System.out.println("Creado objeto "+x);
    }


    // called when object is recycled
    protected void finalize() {
        System.out.println("Finalizing " + x);
    }


}
```

10. EL MÉTODO FINALIZE()

- Ejemplo:

```
public class Prueba {

    public static void main(String[] args) {
        int count;

    /* Now, generate a large number of objects. At some point, garbage collection will occur.
    Note: you might need to increase the number of objects generated in order to force
    garbage collection. */

         for (count = 1; count < 1000000000; count++) {
              EjemploFinalize ob = new EjemploFinalize(count);

         }
    }

}
```

Actividad

- Realiza los ejercicios del 1 y 2 de la hoja de ejercicios: “Tema 3 – Introducción a la programación orientada a objetos - Ejercicios”

11. Acceso a los miembros de una clase

- En JAVA hay varios niveles de acceso a los miembros de una clase.
- Cuando especificamos el nivel de acceso a un atributo o a un método de una clase, lo que estamos especificando es el nivel de accesibilidad que va a tener ese atributo o método que puede ir desde el acceso más restrictivo (**private**) al menos restrictivo (**public**).

11. Acceso a los miembros de una clase

- Dependiendo de la finalidad de la clase, utilizaremos un tipo de acceso u otro:
  - **public (acceso público):** un miembro público puede ser accedido desde cualquier otra clase o subclase que necesite utilizarlo. Una interfaz de una clase estará compuesta por todos los miembros públicos de la misma.
  - **private (acceso privado):** un miembro privado puede ser accedido solamente desde los métodos internos de su propia clase. Otros acceso será denegado.
  - **protected (protegido):** el acceso a estos miembros es igual que el acceso privado. No obstante, para las subclases o clases del mismo paquete (package) a la que pertenece la clase, se considerarán estos miembros como públicos.

11. Acceso a los miembros de una clase

- **No especificado (package):** los miembros no etiquetados podrán ser accedidos por cualquier clase perteneciente al mismo paquete.

- Para un mayor control de acceso se recomienda etiquetar los miembros de una clase como public, private y protected.

11. Acceso a los miembros de una clase

Modificador de acceso	Public	Protected	Private	Sin especificar (package)
¿El método o atributo es accesible desde la propia clase?	Sí	Sí	Sí	Sí
¿El método o atributo es accesible desde otras clases en el mismo paquete?	Sí	Sí	No	Sí
¿El método o atributo es accesible desde una subclase en el mismo paquete?	Sí	Sí	No	Sí
¿El método o atributo es accesible desde subclases en otros paquetes?	Sí	Sí	No	No
¿El método o atributo es accesible desde otras clases en otros paquetes?	Sí	No	No	No

11. Acceso a los miembros de una clase

- Ejemplo:

```
class MiClase {

    private int alpha; // private access
    public int beta; // public access
    int gamma; // default access

    void setAlpha(int a) {
        alpha = a;
    }

    int getAlpha() {
        return alpha;
    }
}

class Acceso {
    public static void main(String args[]) {
        MiClase ob = new MiClase();
        ob.setAlpha(-99);
        System.out.println("ob.alpha is " + ob.getAlpha());
        ob.beta = 88;
        ob.gamma = 99;
    }
}
```

Cuando los atributos de una clase se han declarado **private**, es necesario implementar los métodos **getter** y **setter** para permitir el acceso y manipulación de dicho atributo desde fuera de la clase → **Encapsulación**

12. Referencias y autoreferencias

- Es muy común que los parámetros de un constructor coincidan con los atributos de la clase, y que los constructores se llamen entre si, para poder realizarlo correctamente se utilizan las autoreferencias.
- Si operamos desde **fuera de un objeto**, es posible usar sus elementos (atributos y métodos) usando la referencia al objeto.
- Si operamos desde **la definición de una clase**, tenemos la oportunidad de usar sus elementos sin referencia, o bien usando **this** como referencia.
- Esto nos resultara muy útil en los casos en los que haya variables automáticas con el mismo nombre**.**

12. Referencias y autoreferencias

- En el siguiente ejemplo de referencias disponemos de dos constructores para la clase Perro; en el primero de ellos el nombre de la variable automática es distinto al del atributo, así que no usaremos this como referencia.

```
public Perro(String nombre) {
        nombreMascota = nombre;
```

12. Referencias y autoreferencias

- En este otro ejemplo, la variable automática ostenta idéntico nombre que el atributo, por lo que para asegurarnos de que el compilador Java las distingue debemos usar la referencia **this**, que indicara cómo asignar el valor del atributo de la clase.

```
public Perro(String nombreMascota) {
        this.nombreMascota = nombreMascota;
}
```

12. Referencias y autoreferencias

- Cuando existe más de un constructor, es normal usar como parte de la construcción la llamada a un constructor común a todos.
- Se puede hacer con la referencia this() y encerrando entre parentésis los parámetros del constructor que pretende invocar.
- En este ejemplo de la clase **Persona**, disponemos de dos constructores, de los que el primero será el común, y el segundo llama al primero mediante this() y los parámetros String e int, para luego seguir inicializando los atributos.
- Solo existe la condición de que debe incluirse en la primera línea del constructor.

12. Referencias y autoreferencias

```
public class Persona {


        // atributos
        private String nombre;
        private int edad = 18;
        private String email;
        private Dirección domicilio;

        // constructores
```

```
public Persona(String nombre, int edad, String email) {

        this.nombre = nombre;
        this.edad = edad;
        this.email = email;
}

public Persona(String nombre, int edad, String email,
              Dirección domicilio) {

        this(nombre,edad, email);
        this.domicilio = domicilio;
}
```

Actividad

- Realiza los ejercicios del 3, 4 y 5 de la hoja de ejercicios: “Tema 3 – Introducción a la programación orientada a objetos - Ejercicios”

13. MÉTODOS/ATRIBUTOS DE INSTANCIA Y DE CLASE

- Podemos dividir los métodos/atributos en dos bloques:
  - **Métodos/atributos de instancia:** son aquellos utilizados por la instancia.
  - **Métodos /atributos de clase:** son aquellos comunes para una clase. Un método/atributo por clase.

13.1 MÉTODOS/ATRIBUTOS DE INSTANCIA

- Los métodos de instancia son los más comunes.
- Cada instancia u objeto tendrá sus propios métodos/atributos independientes del mismo método/atributo de otro objeto de la misma clase.

```
public class rectangulo {

    private int ancho=0;
    private int alto=0;

    rectangulo(int an, int al){
        ancho=an; //se puede omitir el this
        this.alto=al;
    }

    public int getAncho(){
        return this.ancho;
    }

    public int getAlto(){
        return alto; //se puede omitir el this
    }

    public rectangulo incrementarAncho(){
        ancho++; //se puede omitir el this
        return this;
    }

    public rectangulo incrementarAlto(){
        this.alto++;
        return this;
    }
```

13.1 MÉTODOS/ATRIBUTOS DE INSTANCIA

- Para referenciar a un método de instancia se creará una instancia (objeto) de la clase y se llamará al método correspondiente del objeto.

```
public static void main(String args[]){
    rectangulo r =  new rectangulo(5,6);
    r.incrementarAlto();

}
```

13.2 MÉTODOS/ATRIBUTOS ESTÁTICOS O DE CLASE

- Un método static no tiene referencia **this**
- Un método static no puede acceder a miembros que no sean static
- Un método no static puede acceder a miembros static y no static

13.2 MÉTODOS/ATRIBUTOS ESTÁTICOS O DE CLASE

- 1º error: los métodos static no tienen referencia this
- 2º y 3º error: un método static no puede acceder a miembros que no sean static.

```
package principal;

import Utilidades.educacion.*;

public class test {
    //Atributos
    public int dato=0;
    public static int datostatico=0;

    //Métodos
    public void metodo(){
        this.datostatico++;
    }

    public static void metodostatico(){
        this.datostatico++;
        datostatico++;
    }

    public static void main(String[] args) {
        dato++;
        datostatico++;
        metodostatico();
        metodo();
    }

}
```

13.2 MÉTODOS/ATRIBUTOS ESTÁTICOS O DE CLASE

- Un ejemplo de los métodos de clase son las funciones de la librería java.lang.Math las cuales pueden ser llamadas anteponiendo el nombre de la clase Math.
- Un ejemplo de llamada a una de las funciones es:

  Math.cos(angulo);

- Se antepone el nombre de la clase al del método.

13.2 MÉTODOS/ATRIBUTOS ESTÁTICOS O DE CLASE

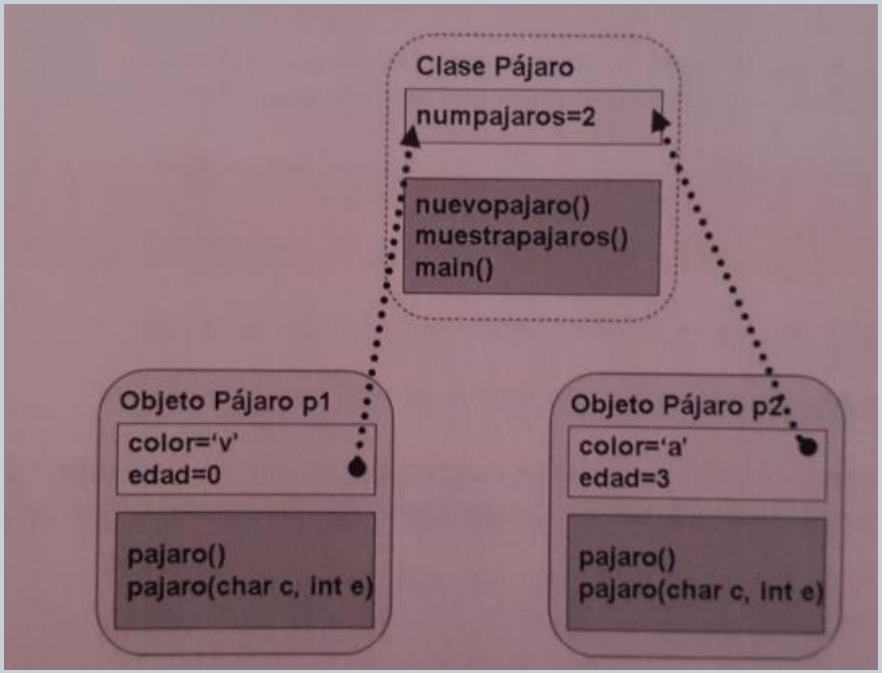
- Por lo tanto, cuando un método o atributo se define como **static**, quiere decir que se va a crear para esa clase solo una instancia de ese método o atributo.
- En el siguiente ejemplo, se ve como se ha creado un atributo numpajaros que contará el número de pájaros que se van generando.
- Si ese atributo fuese no fuese estático, sería imposible contar los pájaros, puesto que en cada instancia del objeto se crearía una variable numpajaros.
- De la misma manera, los métodos *nuevopajaro()*, *muestrapajaro()* y *main()* son estáticos.

13.2 MÉTODOS/ATRIBUTOS ESTÁTICOS O DE CLASE

```
package pajaro;

public class Pajaro {

    private static int numpajaros=0;
    private char color;
    private int edad;

    public Pajaro(){
        color='v';
        edad=0;
        nuevoPajaro();
    }

    public Pajaro(char c, int e){
        color=c;
        edad=e;
        nuevoPajaro();
    }

    static void nuevoPajaro(){
        numpajaros++;
    }

    static void muestraPajaros(){
        System.out.println(numpajaros);
    }

    public static void main(String[] args) {
        Pajaro p1, p2;
        p1=new Pajaro();
        p2= new Pajaro('a',3);

        Pajaro.muestraPajaros();
    }
}
```

13.2 MÉTODOS/ATRIBUTOS ESTÁTICOS O DE CLASE

El atributo numpajaros y los métodos nuevopajaro(), muestrapajaro() y main() se comparten por todos los objetos creados de la clase pájaro.



RESUMEN

Método	Llamada	Declaración	Acceso
Clase	Clase.metodo(parametros)	Static	Miembros de clase
Instancia	Instancia.metodo(parametros)		Miembros de clase y de instancia

Actividad

- Realiza los ejercicios del 6 al 11 de la hoja de ejercicios: “Tema 3 – Introducción a la programación orientada a objetos - Ejercicios”

14. Objetos como parámetros y retorno

- Los objetos pueden ser pasados como parámetros a un método.
- De la misma forma, un método puede devolver un objeto.
- Veamos un ejemplo:

```
class Block {
    int a, b, c;
    int volume;

    Block(int a, int b, int c) {
        this.a=a;
        this.b=b;
        this.c=c;
        volume = a * b * c;
    }

    boolean sameBlock(Block ob) {
        if ((ob.a == a) & (ob.b == b) & (ob.c == c)) {
            return true;
        } else {
            return false;
        }
    }

    boolean sameVolume(Block ob) {
        if (ob.volume == volume) {
            return true;
        } else {
            return false;
        }
    }

    Block initializeBlock()
    {
        return new Block(0,0,0);
    }
}
```

14. Objetos como parámetros y retorno

- La prueba de la clase anterior sería:

```
class PassOb {
    public static void main(String args[]) {
        Block ob1 = new Block(10, 2, 5);
        Block ob2 = new Block(10, 2, 5);
        Block ob3 = new Block(4, 5, 5);
        System.out.println("ob1 same dimensions as ob2: "
                + ob1.sameBlock(ob2));
        System.out.println("ob1 same dimensions as ob3: "
                + ob1.sameBlock(ob3));
        System.out.println("ob1 same volume as ob3: "
                + ob1.sameVolume(ob3));
        ob3=ob3.initializeBlock();

    }
}
```

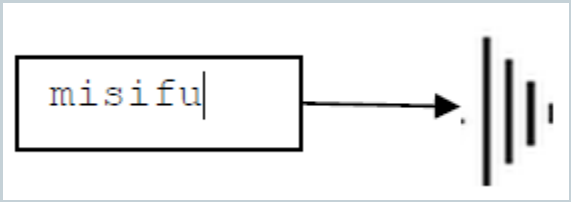
15. Ciclo de vida de un objeto

- Los objetos están dotados de un ciclo de vida.
- Cuando un objeto pierde sus referencias, deja de ser accesible por el programa.
- Desde ese instante, la maquina virtual de Java tiene oportunidad de liberar sus recursos (memoria que ocupa).

15. Ciclo de vida de un objeto

- Las **fases** del ciclo de vida de los objetos:
  - **Definición.** Su finalidad es la creación de una referencia para el objeto (o declaración) en una variable del tipo de la Clase.

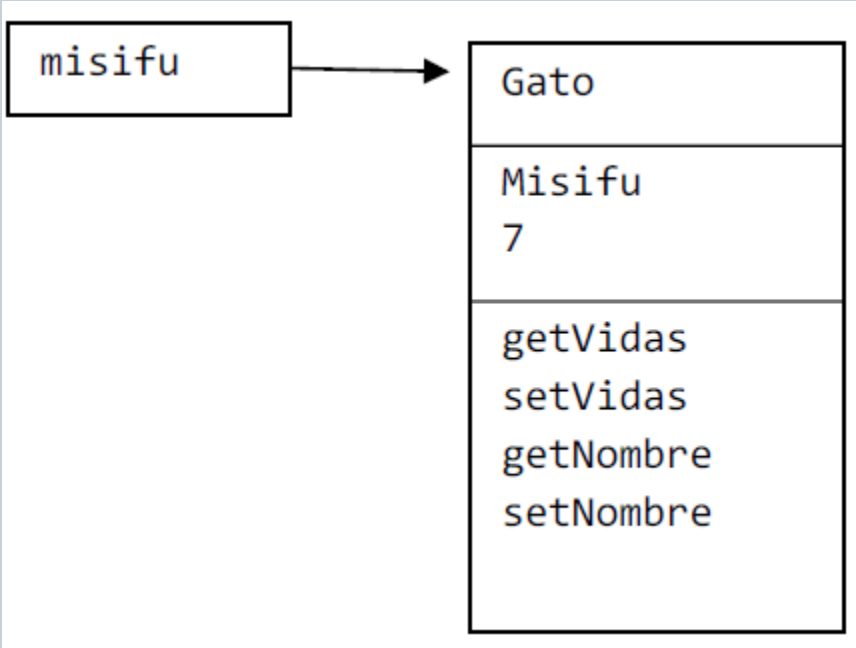
```
Gato misifu; // Inicialmente a null
```



15. Ciclo de vida de un objeto

- **Creación.** En este caso, se busca crear el objeto a través del operador new (instanciación) y mediante la invocación a un método constructor (inicialización) para después almacenar la referencia del objeto en la variable.

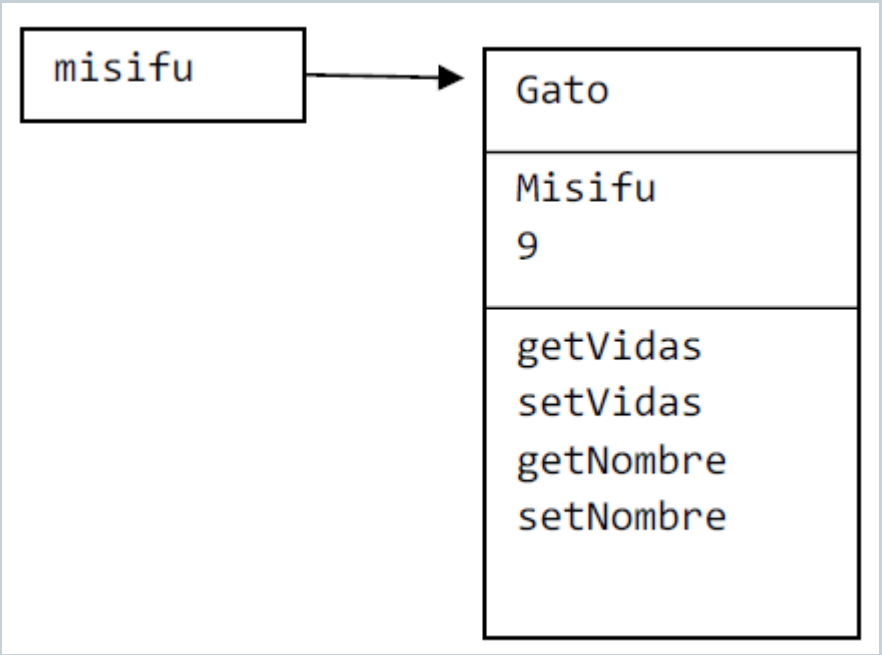
```
Gato misifu = new Gato ("Misifu",7);
```



15. Ciclo de vida de un objeto

- **Uso.** Se trata de dar uso al objeto mediante su referencia: invocar sus métodos, manejar sus datos, proceder a pasarlo como parámetro, etc.

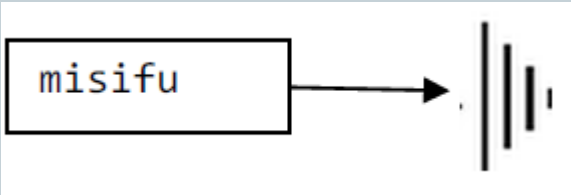
```
misifu.setVidas(9); // En Reino unido los gatos tienen 9 vidas
```



15. Ciclo de vida de un objeto

- **Desaparición.** Es olvidar el objeto cuando no haga falta; si llega un momento en el que no tiene referencias, la máquina virtual de Java se encarga de destruirlo. También existe la posibilidad de forzar esa destrucción si asignamos el valor null a la referencia.

```
misifu =  null; // forzamos la destrucción del objeto
```



15. Ciclo de vida de un objeto

- Si usamos por error un atributo o método de una referencia con el valor null, una excepción **NullPointerException** será lanzada por el sistema y a continuación el programa se cerrará.

```
Gato misifu;
misifu.setVidas(9); // Lanza una excepción NullPointerException
```

- Por suerte, siempre se puede comparar una referencia con el valor null, para saber si una referencia tiene asignada una instancia.
- Así, por ejemplo una manera de evitar la excepción anterior sería ésta:

```
if (misifu!=null) misifu.setVidas(9);
```

15. Ciclo de vida de un objeto

- Usando las expresiones **p!= null** y **p == null** seremos capaces de precisar si es posible o no usar una referencia.

ACTIVIDAD

- Realiza los ejercicios del 1 al 5 de la hoja de ejercicios: “Tema 3 – Introducción a la programación orientada a objetos - Ejercicios II”.

17. Constructor de copia

- **Repaso**: Un constructor es un método especial con el nombre igual que el de la clase, que se usa para inicializar los atributos de un objeto determinado cuando se crea.
- Un **constructor de copia** inicializa un objeto reproduciendo en él los valores de otro objeto distinto de la misma clase.
- Este constructor copia tendrá solo un parámetro: un **objeto** de la misma clase.

17. Constructor de copia

- Un constructor de copia para la clase Gato sería el siguiente:

```
Gato (Gato gatito){

        this.nombre = gatito.getNombre();
        this.vidas = gatito.getVidas();
}
```

- Cuando una clase dispone de un constructor de copia, los objetos de esa clase tienen la oportunidad de usarlo para confeccionar un objeto de la misma clase con iguales atributos.

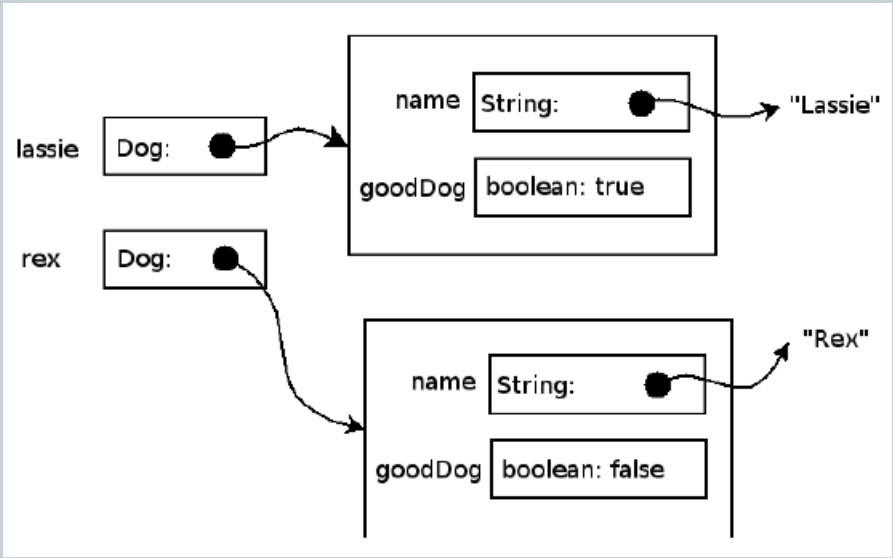
17. Constructor de copia

```
public static void main(String[] args) {
        // Comienzo del programa
        Gato gatito, minino;
        gatito = new Gato("Lucas",9); // crea un objeto Gato
        minino = new Gato(gatito);

        // se visualiza Lucas
        System.out.println(gatito.getNombre());
                // se visualiza Lucas
                System.out.println(minino.getNombre());

        }
}
```

- Al usar el constructor de copia "**se reproducirán**" los miembros del objeto gatito en el objeto minino.

18. Asignación de un objeto a otro

- Al asignar un objeto a otro, o bien al pasar objetos como argumentos a métodos, lo que se copia son referencias, no el contenido de los objetos, como vemos aquí:



18. Asignación de un objeto a otro

- Dada la clase **Dog**:

```java
public class Dog {

	private String name;
	private boolean goodDog;

	public Dog() {
		this.name = "";
		this.goodDog = false;
	}

	public Dog(String name, boolean goodDog) {
		super();
		this.name = name;
		this.goodDog = goodDog;
	}

	public boolean isGoodDog() {
		return goodDog;
	}
```

```java
	public void setGoodDog(boolean goodDog) {
		this.goodDog = goodDog;
	}

	public String getNombre() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	public String toString() {
		return name + ", perro " + goodDog;
	}
}
```

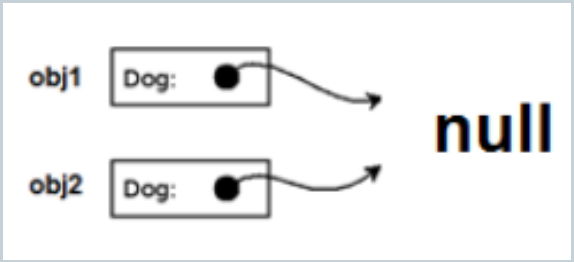
18. Asignación de un objeto a otro

Y el siguiente método **main**:

```
public static void main(String[] args) {
	// Comienzo del programa
	Dog obj1, obj2;
	obj1 = new Dog(); // crea un objeto Dog
	obj1.setName("Lasie");
	obj1.setGoodDog(true);

	obj2 = obj1;

	obj1.setName("Golfo");
	obj1.setGoodDog(false);
	// se visualiza Golfo, perro malo
	System.out.println(obj1.toString());
	// se visualiza Golfo, perro malo
	System.out.println(obj2.toString());

		obj2.setName("Rintintin");
		obj2.setGoodDog(true);

		// se visualiza perro bueno
		System.out.println(obj1.toString());
		// se visualiza perro bueno
		System.out.println(obj2.toString());

	}
}
```

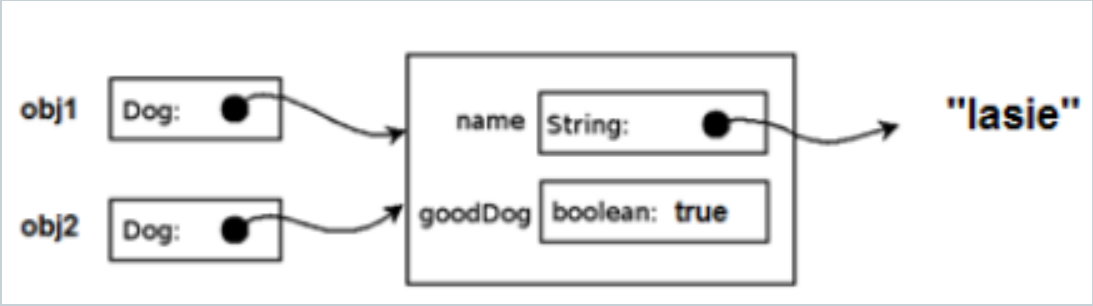
18. Asignación de un objeto a otro

- En el main la parte primera se encarga de declarar dos variables obj1 y obj2 de tipo Dog; en esta primera sentencia únicamente se ha declarado la referencia, con lo que obj1 y obj2 aún no apuntan a objeto ninguno, es decir, apuntan a null.



18. Asignación de un objeto a otro

- Más tarde, inicializamos el objeto **obj1** usando los valores “Lasie” y “bueno”.
- **obj1** apunta ya al objeto que se ha creado, y asignamos entonces el valor de la referencia de **obj1 a obj2.**

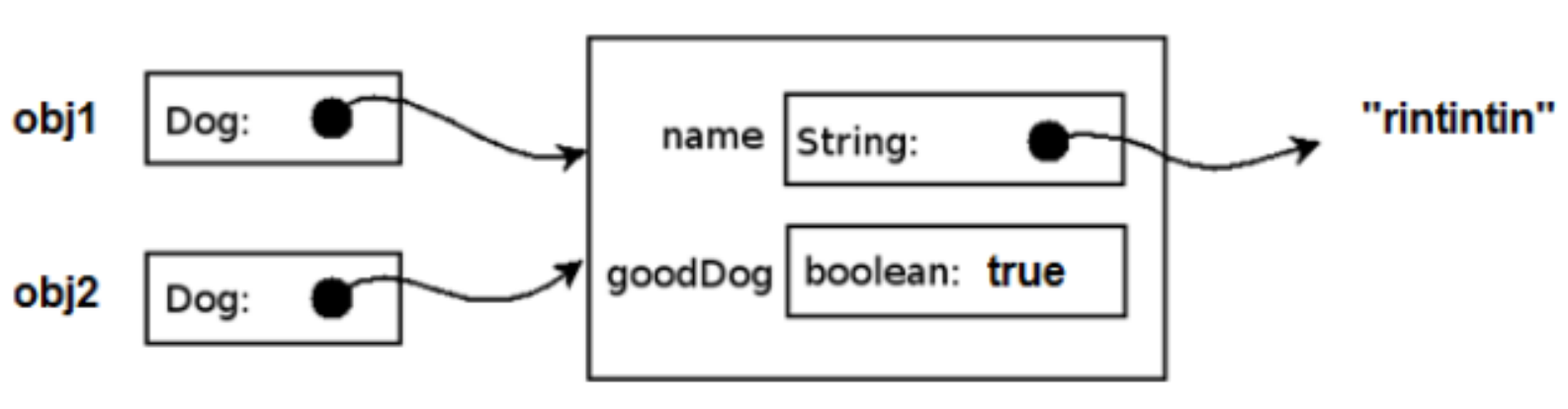


18. Asignación de un objeto a otro

- Conclusión: si con posterioridad se asigna a obj1 un valor adicional “Golfo” y “malo” . Cuál pensamos que será el valor de obj2?
  - Podemos comprobar que es igual que el de obj1.
- ¿Qué ha pasado entonces? Cuando se asigna obj1 = obj2, se crea una referencia nueva al objeto referenciado por obj1. Por ello, modificar obj1 es equivalente a modificar obj2, ya que están referenciando al mismo objeto.

18. Asignación de un objeto a otro

- De igual manera, si después asignamos a obj2 un valor diferente como “rintintin” y “bueno”, ¿cual será el valor de obj1?
  - El mismo que obj2; dado que ambos objetos apuntan al mismo objeto, los cambios realizados en obj2 lógicamente afectan también a obj1.



18. Asignación de un objeto a otro

- Si necesitamos que obj1 y obj2 señalen a objetos separados, tenemos que utilizar new para crear dos objetos separados.

- **IMPORTANTE:** Al asignar un objeto a otro, o al pasar objetos como argumentos a métodos, no copiamos el contenido de los objetos sino **referencias.**

ACTIVIDAD

- Realiza el ejercicio 6 de la hoja de ejercicios: “Tema 3 – Introducción a la programación orientada a objetos - Ejercicios II”.

19. Comparación de objetos

- Cuando comparamos referencias de objetos en Java con el operador ==, se compara su dirección de memoria.
- La comparación devolverá verdadero únicamente si ambas referencias señalan al mismo objeto.
- Esto no siempre es útil, ya que si comparamos dos objetos con distinta referencia, el resultado será falso; aunque los dos objetos tengan todos los atributos iguales.

19. Comparación de objetos

- Si nuestra intención es comparar atributos de los objetos será necesario un método específico.

- Todos los objetos disponen del método **equals()** heredado de su superclase **Object**, que posibilita comparar los contenidos de dos objetos diferentes:
  - **true** si los dos objetos son iguales (de igual tipo y con datos iguales)
  - **false** si se trata del caso opuesto.

- Si las clases son creadas por nosotros (como Dog), debemos **redefinir** el método equals()**.**

19. Comparación de objetos

- Ejemplo: redefinimos **equals() dentro de la clase Dog.**

```
@Override
public boolean equals(Object o){
	if (this == o) // si tienen la misma referencia
		return true;
	if (o instanceof Dog){ // si el objeto es de tipo Perro
	Dog d = (Dog) o;
	if (this.name.equals(d.getNombre()) &&
	    this.goodDog==d.isGoodDog())
		return (true);
	}

	return false;
}
```

19. Comparación de objetos

- **@Override** sirve para redefinir un método.
- 1- Comprobamos si el objeto tiene la misma referencia; si es así, se trata del mismo objeto y como consecuencia se devuelve true.
- 2- Si difieren en la referencia, nos aseguramos de que sus valores son los mismos; efectuamos conversión o casting de Object a Dog y guardamos la referencia en la variable d.
- 3- Si name y isGoodDog resultan iguales, devolvemos true.

19. Comparación de objetos

- **this.name.equals(d.getNombre())** comprueba si los nombres son iguales; es posible usar el método equals de la clase String por ser una clase estándar.

- Para compara *goodDog*, se puede usar ==, porque boolean es un dato primitivo this.goodDog==d.isGoodDog().

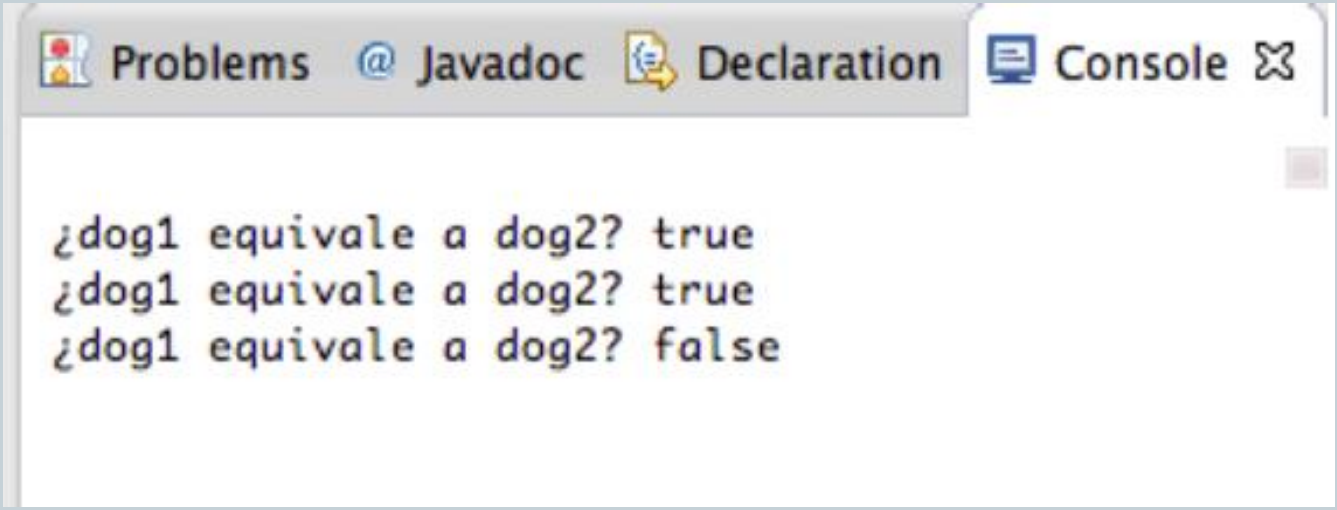
- Si el objeto tiene diferente referencia, no es de tipo Dog o alguno de los atributos no es el mismo, se devolverá **false.**

19. Comparación de objetos

```
Dog dog1 = new Dog ("Lasie",true);
Dog dog2 = dog1;
boolean comparacion;
comparacion = dog1.equals(dog2); // devuelve true
System.out.println("¿dog1 equivale a dog2? "+comparacion);

dog2 = new Dog ("Lasie",true);
comparacion = dog1.equals(dog2); // devuelve true
System.out.println("¿dog1 equivale a dog2? "+comparacion);

dog2.setName("Golfo");
comparacion = dog1.equals(dog2); // devuelve false
System.out.println("¿dog1 equivale a dog2? "+comparacion);
```

19. Comparación de objetos

- Al ejecutar el código, **mostrará los siguientes mensajes:**



19. Comparación de objetos

- En el código anterior, primeramente instanciamos un objeto Dog y almacenamos la referencia en dog1.
- Asignamos la referencia de dog1 a dog2.
- Al invocar a equals() devolverá true, ya que tienen idéntica referencia.
- Más tarde instanciamos un nuevo objeto y almacenamos la referencia en dog2.
- Al llamar a equals(), también devolverá **true**, porque los valores de sus atributos son iguales aunque no tienen la misma referencia.
- El ultimo paso es cambiar el valor del nombre en la referencia de dog2. Si llamamos a equals una vez mas, este devolverá false, ya que no tienen la misma referencia ni los mismos valores.

ACTIVIDAD

- Realiza los ejercicios del 7 al 10 de la hoja de ejercicios: “Tema 3 – Introducción a la programación orientada a objetos - Ejercicios II”.

20. SOBRECARGA DE MÉTODOS

- Se le llama **sobrecarga** a definir varios métodos con el mismo nombre en una clase.

- El número y/o tipo de los parámetros de los métodos **tiene que ser distinto** para que JAVA sepa cuál invocamos.

- Ejemplo - método **setDomicilio()**:
  - Con dos argumentos, dos String con la calle y la ciudad.
  - Tres argumentos, tres Strings con la calle, ciudad y provincia.
  - Cuatro argumentos, tres Strings para la calle, ciudad y provincia y un int para el codPostal.

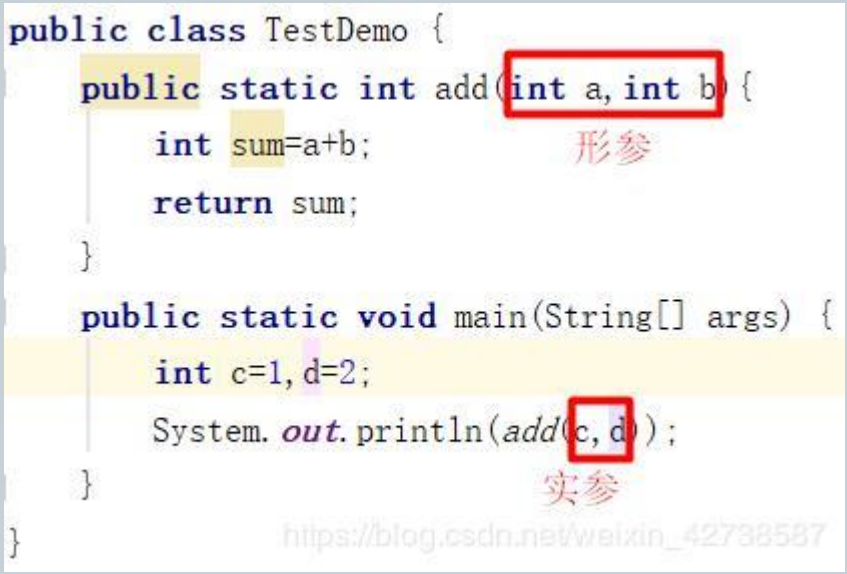
20. SOBRECARGA DE MÉTODOS

```
public void setDomicilio(String calle , String ciudad) {
	this.domicilio.setCalle(calle);
	this.domicilio.setCiudad(ciudad);
}

public void setDomicilio(String calle , String ciudad, String provincia) {
	setDomicilio(calle,ciudad);
	this.domicilio.setProvincia(provincia);
}

public void setDomicilio(String calle , String ciudad, String provincia, int
codPostal) {
	setDomicilio(calle,ciudad,provincia);
	this.domicilio.setCodPostal(codPostal);
	}
```

21. MÉTODOS. PARÁMETROS Y VALORES DE RETORNO

- Formas de invocar a un método:
  - Desde fuera de la clase: `objeto.metodo(argumentos)`
  - Desde dentro de la clase: `metodo(argumentos)`
  - Métodos estáticos: `Clase.metodo(argumentos)`

21.1 Paso de parámetros. Por valor y por referencia



- En la definición de un método se establecen sus **parámetros formales**
- Al invocar al método, se asignan los **parámetros reales**

21.2 PASO DE PARÁMETROS POR VALOR / REFERENCIA

- En JAVA, cuando se pasa un parámetro de un tipo primitivo (int, double, char, boolean, etc.) se copia su valor en el parámetro formal del método
  - Esto se llama paso de parámetros por valor

- En JAVA, cuando se pasa un parámetro de un tipo complejo (objetos, arrays, etc.) se apunta el parámetro formal a la misma zona de memoria
  - Esto se llama paso de parámetros por referencia

21.2 PASO DE PARÁMETROS POR VALOR / REFERENCIA

- Ejemplos:
  - obj.calcula(5); void calcula(int x){ … }
    - La variable ***x*** pasa a valer 5 dentro del método
    - Al terminar la ejecución del método, x se destruye
    - Los cambios realizados a *x* NO afectan al programa principal que invoca a calcula( )
  - obj.compra(ferrari); void compra(Coche coche){ … }
    - El objeto ***coche*** pasa a apuntar a la posición de memoria de Ferrari
    - Al terminar la ejecución, coche se destruye, pero Ferrari sigue apuntando al mismo sitio
    - Los cambios realizados a *coche* SÍ afectan al programa principal que invoca a compra()

21.3 Número variable de parámetros (ampliación)

- En Java el compilador verifica el número y tipo de los parámetros pasados a un método.
- En ciertos casos se hace necesario poder especificar un método y pasarle un número **variable** de argumentos.
- Esta circunstancia es admisible siempre que sean del mismo tipo, añadiendo tres puntos tras el tipo de parámetro, de esta manera:

```
nombreDelMetodo (tipoParametro ...  nombreParametro)
```

21.3 Número variable de parámetros (ampliación)

- Una vez definido un método que recibe uno o más nombres de tipo **String**, el contenido del atributo nombre es **borrado** por el método; luego se procede a recorrer los argumentos obtenidos y **encadenarlos** al atributo nombre, separando con un espacio en blanco.

```
publicvoid setNombre(String... nombre) {
        this.nombre=+""
        for(String n: nombre)
        this.nombre += n+" ";
                }
```

21.3 Número variable de parámetros (ampliación)

- Así tenemos que las llamadas posteriores a **setNombre** son válidas, y les pasamos distintos números de argumentos:

```
public class Test {

	publicstaticvoid main(String args[]) {

		Persona p1 = new Persona();
		Persona p2 = new Persona();

		p1.setNombre("Jorge","Álvarez","Castrillejo");
		p2.setNombre("John","Smith");

		System.out.println(p1.getNombre());
		System.out.println(p2.getNombre());

	}

}
```

21.3 Número variable de parámetros (ampliación)

- Si el método, aparte del parámetro de número variable, tiene parámetros "normales", el parámetro de número variable debe ser el último parámetro declarado en el método.
- Ejemplo:

```
public static void vTest2(String msg, int... v) {
    System.out.println(msg + v.length);
    System.out.println("Contents: ");
    for (int i = 0; i < v.length; i++) {
        System.out.println(" arg " + i + ": " + v[i]);
    }
    System.out.println();
}
```

21.3 Número variable de parámetros (ampliación)

- Los métodos con número variable de parámetros también pueden ser sobrecargados:

```
public static void vaTest(int... v) {
    System.out.println("vaTest(int ...): "
            + "Number of args: " + v.length);
    System.out.println("Contents: ");
    for (int i = 0; i < v.length; i++) {
        System.out.println(" arg " + i + ": " + v[i]);
    }
    System.out.println();
}

public static void vaTest(boolean... v) {
    System.out.println("vaTest(boolean ...): "
            + "Number of args: " + v.length);
    System.out.println("Contents: ");
    for (int i = 0; i < v.length; i++) {
        System.out.println(" arg " + i + ": " + v[i]);
    }
    System.out.println();
}
```

```
public static void vTest(String msg, int... v) {
    System.out.println(msg + v.length);
    System.out.println("Contents: ");
    for (int i = 0; i < v.length; i++) {
        System.out.println(" arg " + i + ": " + v[i]);
    }
    System.out.println();
}
```

21.3 Número variable de parámetros (ampliación)

- En este caso, se podrían producir los siguientes errores de compilación:

```
public static void main(String args[]) {
     vaTest(1, 2, 3); // OK
     vaTest(true, false, false); // OK
     vaTest(); // Error: Ambiguous!
}
```

ACTIVIDAD

- Realiza los ejercicios del 11 y 12 de la hoja de ejercicios: “Tema 3 – Introducción a la programación orientada a objetos - Ejercicios II”.

- Ampliación: Realiza los ejercicios del 13 al 17 de la hoja de ejercicios: “Tema 3 – Introducción a la programación orientada a objetos - Ejercicios II”.

Normas de codificación

- http://javafoundations.blogspot.com/2010/07/java-estandares-de-programacion.html
- https://amap.cantabria.es/amap/bin/view/AMAP/CodificacionJava