Tema 4

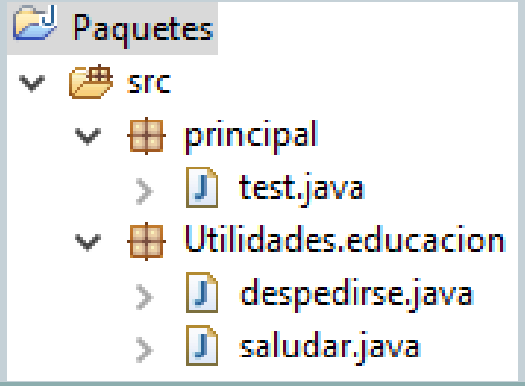
PROGRAMACIÓN ORIENTADA A OBJETOS. CLASES Y OBJETOS

1. CREACIÓN DE PAQUETES

- Un paquete es un conjunto de clases relacionadas entre sí.
- Un paquete puede contener a su vez subpaquetes.
- Java mantiene su biblioteca de clases en una estructura jerárquica.
- Cuando nos referimos a una clase de un paquete (salvo que se haya importado el paquete) hay que referirse a la misma especificando el paquete (y subpaquete si es necesario) al que pertenece (Ejemplo: java.io.File).

1. CREACIÓN DE PAQUETES

- Los paquetes permiten reducir los conflictos en los nombres puesto que dos clases que se llaman igual, si pertenecen a paquetes distintos, no deberían dar problemas.
- Los paquetes permiten proteger ciertas clases no públicas al acceso desde fuera del mismo.
- Crea la siguiente estructura:



1. CREACIÓN DE PAQUETES

- El código de cada clase es:

```
package Utilidades.educacion;

public class saludar {

    public void saludo(){
        System.out.println("Hola");
    }

}
```

```
package Utilidades.educacion;

public class despedirse {

    public void despedida(){
        System.out.println("Adiós");
    }

}
```

```
package principal;

import Utilidades.educacion.*;

public class test {

    public static void main(String[] args) {
        saludar s = new saludar();
        despedirse d = new despedirse();

        s.saludo();
        d.despedida();

    }

}
```

2. CLASES

- En la POO las clases permiten a los programadores abstraer el problema a resolver ocultando los datos y la manera en la que estos se manejan para llegar a la solución (se oculta la implementación).
- En un POO es impensable que desde el mismo programa se acceda directamente a las variables internas de una clase si no es a través de los métodos getters y setters.

2. CLASES

- Por lo tanto, en la definición de las clases deberemos de cuidar lo siguiente:
  - No se deberá tener acceso directo a la estructura interna de las clases. El acceso a los atributos será a través de getters y setters.
  - En el supuesto que haya que modificar el código sin modificar el interfaz con otras clases o programas, esto debería poder hacerse sin tener ninguna repercusión con otras clases o programas. Se busca que las clases tenga un alto grado de independencia.

2. CLASES

- 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).

2. CLASES

- 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.

2. CLASES

- **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.

2. CLASES

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

3. CABECERA Y CUERPO DE UN MÉTODO

- Un método tiene dos **partes claramente diferenciadas, la cabecera y el cuerpo.**
- La **cabecera incluye:**
  - **Accesibilidad del método:** public, private, protected y package.
  - **Tipo de valor a devolver:** para que el método devuelva un valor, habrá que especificar qué tipo de valor se devuelve (*byte, short, int, long, float, double, boolean o char); en el caso de que no devuelva valor, se especificara void.*
  - **Nombre del método:** el identificador con el que se invocará (usará) el método.
  - **Parámetros**: los valores que se usaran en el código que conforma el cuerpo del método, necesarios para su ejecución. Los parámetros aparecen siempre entre paréntesis.
  - **Tipo de excepción:** si el método puede lanzar excepciones hay que indicarlo aquí; lo estudiaremos más adelante en otra unidad.

3. CABECERA Y CUERPO DE UN MÉTODO

- Por lo tanto, la **cabecera** de un método tiene el siguiente aspecto:

```
acceso      tipo  nombre      [parámetros]      excepciones
```

- Las palabras relativas a la accesibilidad del método y al tipo de excepción no son obligatorias.
- También puede haber métodos que no requieran parámetros, en cuyo caso pondremos los paréntesis vacios ().

3. CABECERA Y CUERPO DE UN MÉTODO

- La segunda parte claramente diferenciada de un método es el **cuerpo**, conformado por una serie de instrucciones entre llaves (bloque).
- Dentro de este bloque es posible declarar variables que llamaremos automáticas o locales y que existirán únicamente mientras estemos dentro del método.
- En cuanto salimos del método todas las variables locales se destruyen.

4. MÉTODOS ACCESORES Y MUTADORES

- Un **método mutador** o **modificador** sirve para controlar las modificaciones que se realizan en un atributo de una clase.
- Son también conocidos por su termino en ingles: métodos ***setter***.
- Los métodos *setter* suelen acompañarse de un método **accesor o selector**, también conocido como método ***getter***, encargado de devolver el valor de un atributo *o* miembro privado de una clase.
- Estos métodos se usan especialmente en la programación orientada a objetos con el fin de mantener el principio de **encapsulación**

4. MÉTODOS ACCESORES Y MUTADORES

- Según este principio, los **miembros de una clase tendrán acceso privado con el fin de esconderlos y protegerlos del acceso de los usuarios de esa clase, y únicamente podrán ser transformados por un miembro público (el método mutador)** que podrá obtener el nuevo valor en forma de parámetro y de manera opcional podrá validarlo, para así modificar el miembro privado de la clase.
- Puedes ampliar información sobre los *getters y setters* en la pagina oficial de Java de Oracle: http://docs.oracle.com/javaee/7/tutorial/doc/cdibasic010.htm

5. SOBRECARGA DE MÉTODOS

- Utilizando clases de la biblioteca de Java a menudo aparecen clases que implementan reiteradamente el mismo método.
- En este ejemplo, la clase **PrintStream** utilizada en **System.out**, implementa como vemos diez formas diferentes del método println().

```
1.  public void println();
2.  public void println(boolean x)
3.  public void     println(char x)
4.  public void     println(char[] x)
5.  public void     println(double x)
6.  public void     println(float x)
7.  public void     println(int x)
8.  public void     println(long x)
9.  public void     println(Object x)
10. public void     println(String x)
```

5. SOBRECARGA DE MÉTODOS

- La diferencia está en su **signatura**, lo que son su número y tipo de parámetros.
- Mirando bien, println() no tiene ningun parametro, y el resto, aunque todos ostentan un parámetro, son en su totalidad de distintos tipos (boolean, char, double ...).
- Un método está **sobrecargado** cuando se define varias veces un mismo método en una clase con distinto número de parámetros, o bien lo encontramos definido con igual numero de parámetros pero de tipo diferente en alguno de los casos.

5. SOBRECARGA DE MÉTODOS

- Los métodos sobrecargados pueden diferenciarse además de lo anterior en el tipo de valor que devuelven, pero el compilador Java no admite dos métodos declarados que solo sean distintos en el tipo del valor retornado; deben ser distintos también en la lista de parámetros.
- Lo que importa es su signatura, lo que viene a ser su número y el tipo de parámetros.
- La sobrecarga de métodos permite que no sea imprescindible nombrar de forma diferente métodos cuya función es fundamentalmente la misma, como es el caso del método println, que permite al método modificar su comportamiento según el número de parámetros con el que sea llamado.

5. SOBRECARGA DE MÉTODOS

- Para ejemplificar este tema procederemos a sobrecargar el método **setDomicilio** para tener la oportunidad de invocarlo:
  - 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.

5. 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);
	}
```

5. SOBRECARGA DE MÉTODOS

- Por cada llamada al método **setDomicilio** en un programa, el compilador Java tiene que resolver cual de los métodos con el nombre setDomicilio es invocado, cosa que consigue cotejando las signaturas de sus diferentes definiciones.
- Aquí podemos ver las posibles maneras de invocar al método setDomicilio:

```
public class Test {

public static void main(String args[]) {

        Persona p1 = new Persona();

        p1.setDomicilio(new Dirección("C\ Condesa","León", "LEÓN", 24004));
        p1.setDomicilio("C\ Condesa",","León");
        p1.setDomicilio(("C\ Condesa","León", "LEÓN");

       p1.setDomicilio("C\ Condesa","León", "LEÓN", 24004);

        }

}
```

6. CONSTRUCTORES EN JAVA

- Se le llama constructor a un método especial con el mismo nombre que la clase y que se utiliza para inicializar los atributos de un objeto concreto cuando este es creado.
- Si no se define un constructor par la clase, Java creará uno por defecto.
- Java, igual que con las variables, cuando va a crear un objeto comienza por reservar el espacio necesario para el en la memoria.

6. CONSTRUCTORES EN JAVA

- En esta etapa crea un constructor público por defecto de cada objeto sin parámetros de entrada y que asigna estos valores según su tipo:

```
Enteros → 0
Reales → 0.0
Char →'0'
Boolean → false
Objetos  → null
```

6. CONSTRUCTORES EN JAVA

- Para la definición de un método constructor es necesario rellenar **el tipo de acceso,** el **nombre** de la **clase, los parámetros que soporta, si lanza excepciones, y un cuerpo** o bloque de código en el que ejecutaremos las inicializaciones de atributos de clase que nos sean necesarias.

```
Acceso nombreClase (parámetros) excepciones {
        ...
}
```

6. CONSTRUCTORES EN JAVA

- Aquí podemos ver un ejemplo de definición de **método constructor:**

```
public class Perro {
	private String nombreMascota;

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

- Por lo tanto, existen dos tipos de constructores, que son **constructor por defecto (creado por Java) y el constructor definido.**

6. CONSTRUCTORES EN JAVA

- Si incorporamos uno o varios constructores definidos a una clase, el constructor por defecto de Java ya no será accesible en favor de los nuevos.
- La denominación de un constructor definido es igual al nombre de la clase. No devuelve valores y no es posible declararlo como *static, final, abstract o synchronized*.
- *Normalmente* los constructores se declaran como públicos (public) para permitir su utilización por cualquier otra clase.

6.1 SOBRECARGA DEL CONSTRUCTOR

- Se puede incorporar un número múltiple de constructores definidos para una clase para que el objeto pueda ser inicializado de múltiples maneras.
- Al sobrecargar un constructor variaremos el tipo y el numero de parámetros que recibe.
- Ejemplo:

```
public class Perro {

	private String nombreMascota;

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

	public Perro() {
		nombreMascota = "";
	}
}
```

6.1 SOBRECARGA DEL CONSTRUCTOR

- Los casos en los que se dispone de más de un constructor para una clase se dice que este esta sobrecargado.
- Cuando creamos un objeto con ***new***, Java elige el constructor mas adecuado dependiendo de los parámetros utilizados.

7. 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**.**

7. 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;
```

7. 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;
}
```

7. 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.

7. 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;
}
```

8. 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.

8.1 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;
    }
```

8.1 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();

}
```

8.2 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

7.2 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();
    }

}
```

7.2 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.

RESUMEN

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

8. PASO DE OBJETOS A MÉTODOS y DEVOLUCIÓN DE OBJETOS

- Un método puede recibir parámetros que sean objetos.
- Al mismo tiempo, un método puede devolver objetos.
- Ejemplo:

```
public class Block {

    private int a, b, c;
    private int volume;

    public Block(int a, int b, int c) {
            this.a=a;
            this.b=b;
            this.c=c;
            volume = a * b * c;
        }
    public boolean sameBlock(Block ob) {
        if ((ob.a == a) && (ob.b == b) && (ob.c == c)) {
            return true;
        }
        else {
            return false;
        }
    }
```

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

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

8. PASO DE OBJETOS A MÉTODOS y DEVOLUCIÓN DE OBJETOS

- Ejemplo:

```
public class Prueba {

    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();

    }

}
```

ACTIVIDAD

- Realiza los ejercicios 1, 2 y 3 de la hoja de ejercicios.

9. OBJETOS

- Un objeto es un caso particular de una clase.
- En Java, la unidad de programación es la clase de la cual en algún momento se instancia (es decir, se crea) un objeto.
- Habitualmente un programa utiliza varios objetos de la misma clase.

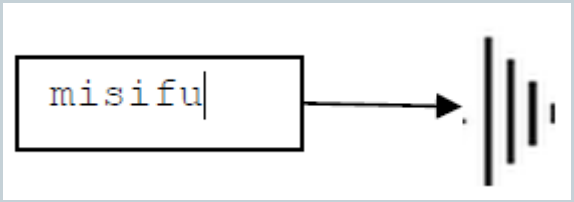
9.1 CICLO DE VIDA DE UN OBJETO

- 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).

9.1 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.

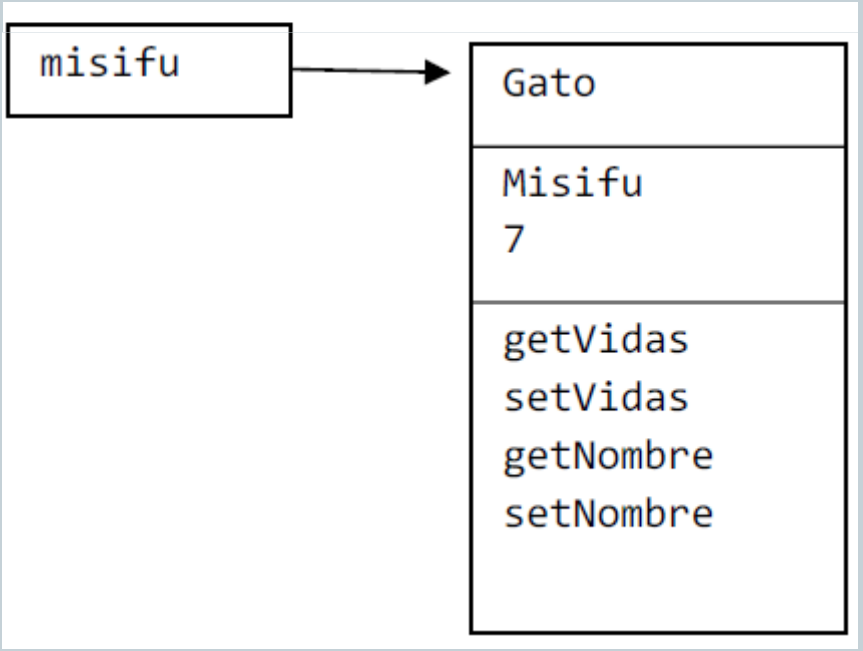
```
Gato misifu; // Inicialmente a null
```



9.1 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.

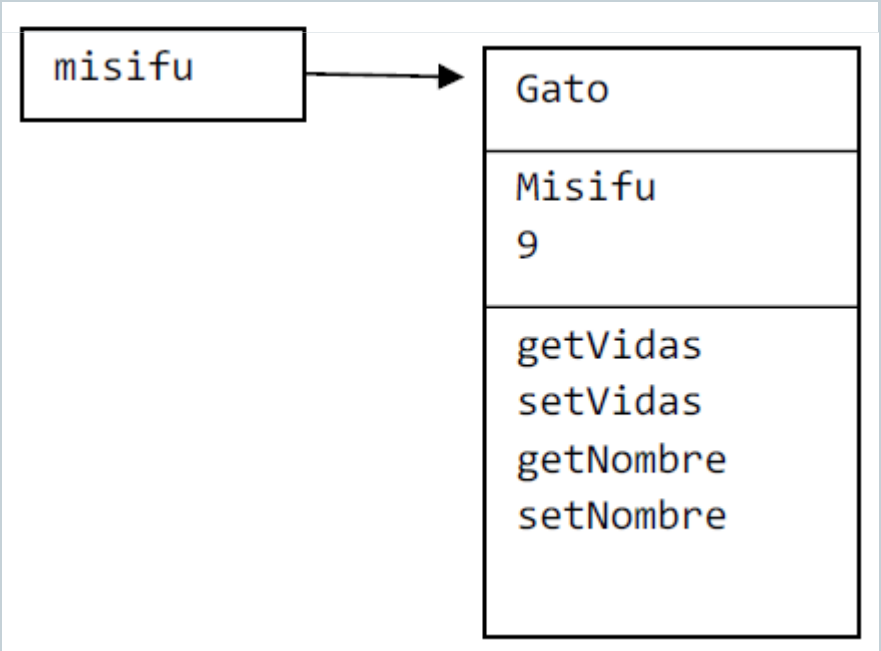
```
Gato misifu = new Gato ("Misifu",7);
```



9.1 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.

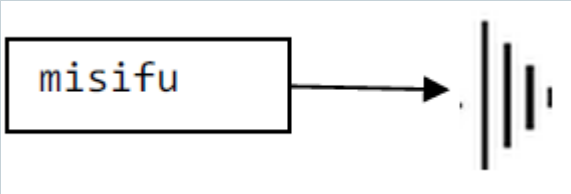
```
misifu.setVidas(9); // En Reino unido los gatos tienen 9 vidas
```



9.1 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
```



9.1 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);
```

9.1 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.

9.2. INICIALIZADORES STATIC

- Los **inicializadores static** son un bloque de código que se ejecutará una vez solamente cuando se utilice la clase.
- A diferencia del constructor que se llama cada vez que se crea un objeto de dicha clase, el inicializador solamente se ejecuta la primera vez que se utiliza la clase.
- Siguen las siguientes reglas:
  - No devuelven ningún valor
  - Son métodos sin nombre
  - Ideal para inicializar objetos

9.2. INICIALIZADORES STATIC

- Siguen las siguientes reglas:
  - Permiten gestionar excepciones
  - Se puede crear más de un inicializador static y se ejecutarán según el orden en el que se han definido.
  - Se pueden utilizar para invocar métodos nativos o para inicializar variables static
- Ejemplo:

```
public class testInicializador {

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

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

    public testInicializador(){
        System.out.println("Llamada al constructor");
    }
}
```

9.2. INICIALIZADORES STATIC

- Ejemplo:

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

}
```

- Este programa mostrará por pantalla lo siguiente:

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

9.3 CONSTRUCTOR DE COPIA

- 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.
- Ciertos constructores especiales denominados **constructores de copia** inicializan un objeto reproduciendo en el los valores de otro objeto distinto, perteneciente a la misma clase.
- Este constructor copia tendrá solo un parámetro: un **objeto** de la misma clase.

9.3 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 copia, los objetos de esa clase tienen la oportunidad de usarlo para confeccionar un objeto de la misma clase con iguales atributos.

9.3 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 copia "**se reproducirán**" los miembros del objeto gatito en el objeto minino.

10. Instanciación de objetos. Declaración y creación

- Si queremos crear un objeto de una clase deberemos hacerlo de manera **explícita** utilizando el **operador new.**

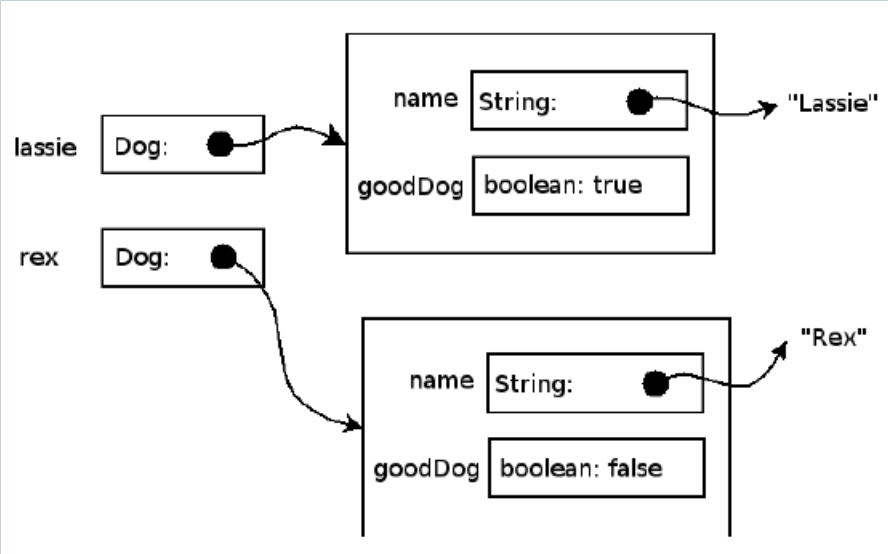
```
Dog lassie = new Dog ("Lassie",true);
Dog rex = new Dog ("Rex",false);
```

- Las sentencias anteriores instancian (crean) dos objetos de la clase Dog, aunque realmente realizan tres acciones que podemos programar por separado; **declaración, instanciación e inicialización.**
  - **Dog lassie:** declaración de variable, únicamente informa al compilador de que el nombre **lassie** se usara para referirse a un objeto cuyo tipo es **Dog.**
  - El operador **new** procede a instanciar la clase Dog (crea un nuevo objeto Dog).

10. Instanciación de objetos. Declaración y creación

- **Dog ("Lassie", true)** se encarga de inicializar el objeto.

- 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:



10. Instanciación de objetos. Declaración y creación

- Si en algún caso tenemos que una referencia no se refiere a **ningún objeto** (como puede ocurrir en el paso antes de crear, si por el momento solo lo hemos declarado) se dice que apunta a **null.**
- Un objeto puede tener varias referencias. Esto quiere decir que dos o más referencias pueden apuntar al mismo objeto.
- En programación, cuando esto ocurre se denomina utilizar **alias.**
- En el siguiente ejemplo nos extenderemos en el **concepto de alias,** siguiendo con la clase Dog, con los atributos name y goodDog.

10. Instanciación de objetos. Declaración y creación

- La clase Dog encapsula la estructura de datos conformada por un “String y un boolean: name y goodDog”; y para posibilitar el acceso a esta estructura provee la interfaz pública que forman los métodos:

```
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;
	}
```

```
	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;
	}
}
```

10. Instanciación de objetos. Declaración y creación

```
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());
```

10. Instanciación de objetos. Declaración y creación

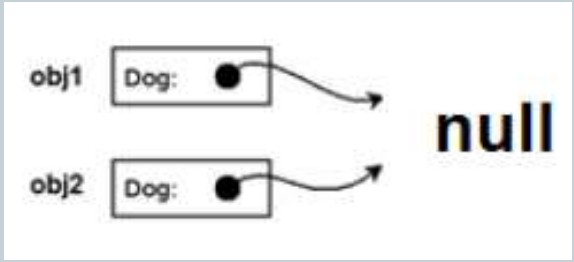
```
        obj2.setName("Rintintin");
        obj2.setGoodDog(true);

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

    }
}
```

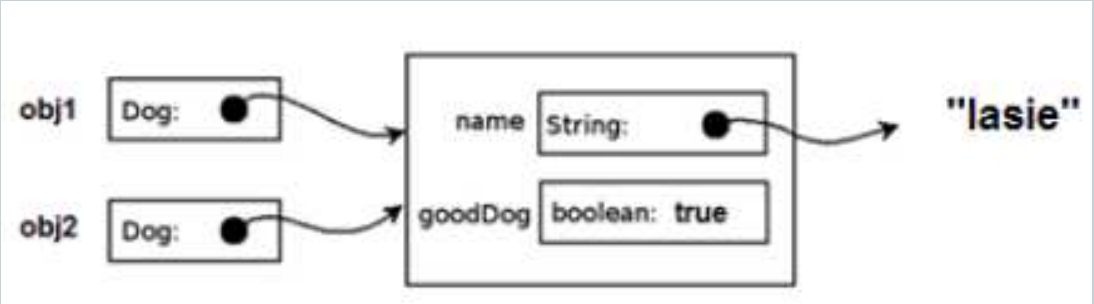
10. Instanciación de objetos. Declaración y creación

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



10. Instanciación de objetos. Declaración y creación

- 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.**

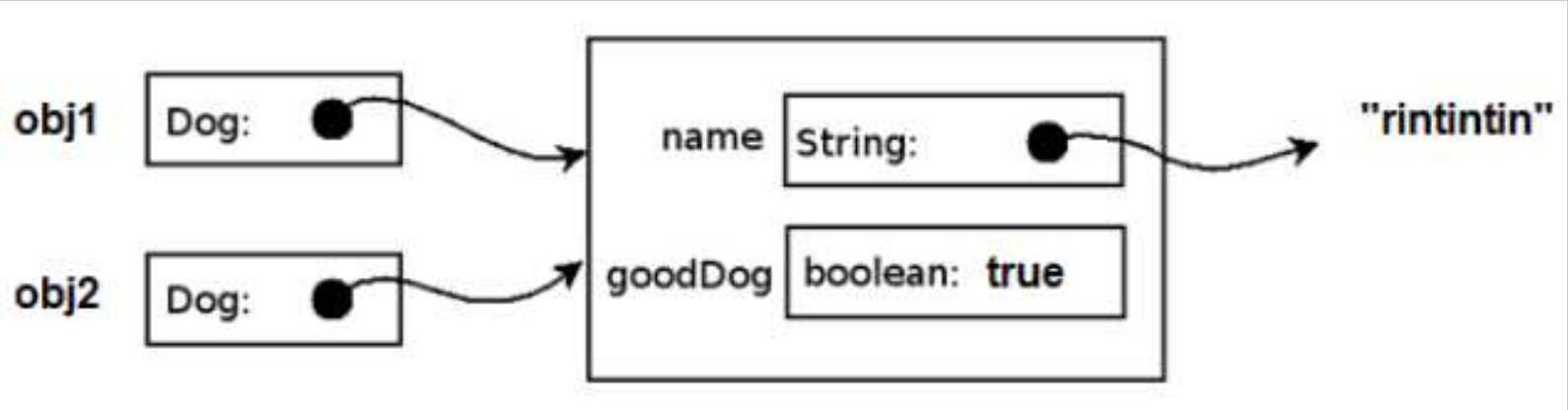


10. Instanciación de objetos. Declaración y creación

- Como conclusión del ejemplo, preguntémonos ahora: 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? La respuesta es que cuando se asigna obj1 a obj2, se crea una referencia nueva al mismo objeto referenciado por obj1. Por ello, modificar el objeto al que se refiere obj1 es equivalente a modificar el objeto al que se refiere obj2 porque obj1 y obj2 están referenciando al mismo objeto.

10. Instanciación de objetos. Declaración y creación

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



10. Instanciación de objetos. Declaración y creación

- Si en realidad necesitáramos que obj1 y obj2 señalaran a objetos separados, habríamos de utilizar new con las dos referencias con el fin de crear entonces 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.**

11. Comparación de objetos

- Cuando comparamos referencias de objetos en Java con el operador ==, se coteja su valor (la dirección de memoria del objeto).
- La comparación devolverá verdadero únicamente si ambas referencias señalan al mismo objeto.
- Sin embargo, los resultados de las comparaciones de objetos mediante el operador == no son válidas, ya que si intentamos comparar dos objetos con distinta referencia, el resultado será falso; pero también en el caso de dos objetos con todos los atributos iguales, ya que tienen distintas referencias.

11. Comparación de objetos

- Si nuestra intención es comparar 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, de manera que el resultado es true si los dos objetos son iguales (de igual tipo y con datos iguales) y false si se trata del caso opuesto.
- Si las clases no son estándar, como sería el caso de Dog, deberemos redefinir el método equals() para conseguir cotejarlos de manera correcta.

11. Comparación de objetos

- Veamos en este ejemplo la **redefinición de un método.**
- El método equals() de la clase Object se define así:

```
public boolean equals(Object obj) {
  return (this == obj);
}
```

11. Comparación de objetos

- Primeramente redefiniremos el método **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;
}
```

11. Comparación de objetos

- La primera sentencia **@Override** sirve para que el compilador confirme que estamos en el proceso de redefinir un método heredado y no en la creación de uno nuevo.
- Comprobamos si el objeto ostenta la misma referencia; si es así, se trata del mismo objeto y como consecuencia se devuelve true.
- Si difieren en la referencia, en caso de que el objeto sea de tipo Dog, procedemos a asegurarnos de que sus valores son los mismos; efectuamos conversión o casting de Object a Dog (profundizaremos en instanceof, la clase Object y el casting en unidades posteriores) y guardamos la referencia en la variable d.
- Si name y isGoodDog resultan iguales, devolvemos true.

11. Comparación de objetos

- **this.name.equals(d.getNombre())** efectúa la comprobación de que los nombres son iguales; es posible usar el método equals de la clase String por ser una clase estándar.
- Para cotejar si **es bueno,** es suficiente con hacer uso del operador de comparación, por ser boolean un dato primitivo this.goodDog==d.isGoodDog() y finalmente sumamos el operador logico && para que ambas condiciones se cumplan.
- Únicamente llegaremos a la ultima sentencia si el objeto tiene diferente referencia, no es de tipo Dog o alguno de los atributos no es el mismo, por lo que se devolverá **false.**

11. Comparación de objetos

- En el código de prueba que sigue, 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.

11. Comparación de objetos

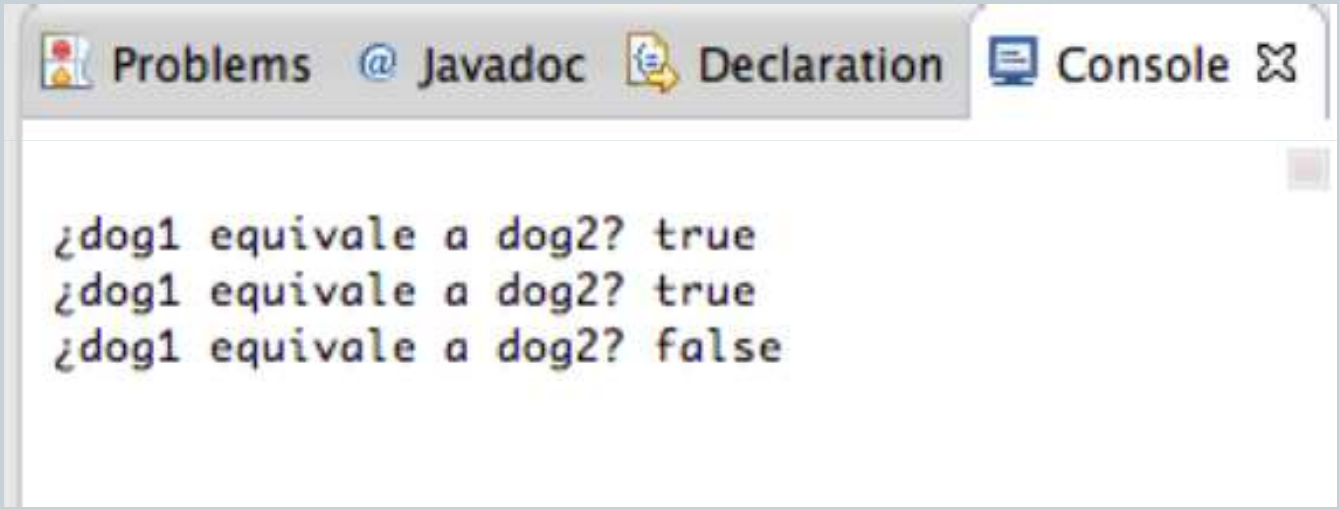
```
Dog dog1 = new Dog ("Lasie",true);
Dog dog2 = dog1;
boolean comparacion;
comparacion = dog1.equals(dog2); // devuelve true
System.out.println("¿dog1 equivale a dog2? "+comparacion);

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);
```

11. Comparación de objetos

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



ACTIVIDAD

- Realiza los ejercicios del 4 al 12 de la hoja de ejercicios

12. MÉTODOS. PARÁMETROS Y VALORES DE RETORNO

- Los métodos son los encargados de definir las operaciones que es posible realizar con sus atributos.
- Si nuestra intención es invocar un método podemos elegir entre tres mecanismos diferentes para ello:
  - Si estamos fuera de la clase en la que se define el método, deberemos incluir el nombre de un objeto que tenga el método, un punto y además el nombre del método que deseamos llamar:

```
objeto.metodo(argumentos)
```

12. MÉTODOS. PARÁMETROS Y VALORES DE RETORNO

- Si hablamos de un método static o método de clase pondremos el nombre de la clase que define el método, un punto, y añadiremos el nombre del método que vamos a llamar:

```
Clase.metodo(argumentos)
```

- Será suficiente con el nombre, en caso de que hagamos uso del método en la misma clase que lo define:

```
metodo(argumentos)
```

12.1 Paso de parámetros. Por valor y por referencia

- Denominamos **argumentos** de la llamada a los valores que utilizamos en la invocación del método, los que se usan para cargar los parámetros.
- Para explicar como funciona la ejecución de un método con los argumentos de la llamada, veamos el método denominado mueve con los parámetros nuevox, nuevoy y b.

```
public void mueve(float nuevox, float nuevoy, boolean b) {
        //…
}
```

12.1 Paso de parámetros. Por valor y por referencia

- Usaremos para llamar a este método desde un objeto los argumentos 3.15, Math.sqrt(22.11*x)+9 y true.
- Consecuentemente, al parámetro nuevox se le da el valor 3.15, al parámetro b el valor true, etc.

```
p.mueve(3.15, Math.sqrt(22.11*x)+9 ,true);
```

12.1 Paso de parámetros. Por valor y por referencia

- Por lo tanto:
  - **Parámetro:** en la definición de un método, es la denominación de cada valor que será utilizado en el cuerpo del método.
  - **Argumento:** valor que se utiliza en la invocación al método.

- **Aclaración:** En ciertos libros, de manera diferente a como hacemos nosotros, los parámetros son denominados **parámetros formales** y los argumentos se llaman **parámetros reales**.

12.1.1 Por valor

- Los tipos primitivos invariablemente efectúan paso por valor; es decir, hacen una reproducción del valor que tiene la variable del método para que el método utilice esta “copia” en lugar del original.
- Así se previene el riesgo de modificación del valor original al actuar el método sobre un parámetro de tipo primitivo.

12.1.1 Por valor

- En el siguiente ejemplo de paso por valor, podemos ver que la línea sombreada invoca al método cumpleaños y se encarga de copiar (reproducir) el valor del argumento edad en el parámetro años del método; decimos que el argumento ha sido **pasado por valor**.

```
static void cumpleaños(int años){
            años++;
      }

      public static void main(String args[]) {

            int edad = 18;
            cumpleaños(edad);

            System.out.println(edad);
      }
```

12.1.1 Por valor

- Aunque el método ejecute cambios sobre años, estos dejaran intacta la variable original.
- La consola ostentara su valor original de edad, el resultado 18.
- Si necesitamos que estas variables que pasamos como parámetros modifiquen su valor tras ejecutar el método en el caso de que este las haya cambiado, podremos hacerlo con la sentencia **return**.
- Que será necesario hacer para que un método pueda cambiar el valor original del argumento que le es pasado? Hacer uso de un paso de **parámetros por referencia**.

12.1.2 Por referencia

- Los objetos siempre se pasan por referencia.
- Con esto queremos decir que Java no hace una replica del objeto sobre el parámetro del método; en cambio, proporciona al método la localización de ese objeto con el fin de que le sea posible accederlo.
- Pasa como valor la referencia, es decir, la dirección de memoria donde esta el objeto original.

12.1.2 Por referencia

- Una posible solución para poder modificar el valor sería la creación de una clase de nombre **Edad**, que dispondría de métodos mutador y seleccionador (*setter y getter*).

```
public class Edad {
	prívate int años;

	public Edad(int años) {

		this.años = años;
	}

	public int getAños() {
		returnaños;
	}

	public void setAños(int años) {
		this.años = años;
	}

}
```

12.1.2 Por referencia

- De esta manera, al ejecutar el método cumpleaños se modificaría el valor de edad desde la referencia guardada en años, imprimiendo en consola 19.

```
Static void cumpleaños(Edad años){
        int valor = años.getAños();
        valor++;
       años.setAños(valor);
}


public static void main(String args[]) {

        Edad edad = new Edad(18);
        cumpleaños(edad);

        System.out.println(edad.getAños());


}
```

RESUMEN

- Las variables pasadas como **parámetro** se cambian a través de su interfaz pública en el paso de **parámetros por referencia**, a causa de que el método opera usando las direcciones de memoria de los parámetros.
- En el paso de **parámetros por valor** los parámetros se copian o reproducen en las variables del método, y las variables pasadas como parámetro no se modifican.

12.1.3 Número variable de parámetros

- 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 numero 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)
```

12.1.3 Número variable de parámetros

- En el siguiente ejemplo veremos un número variable de argumentos en un método.
- 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+" ";
                }
```

12.1.3 Número variable de parámetros

- 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());

        }

}
```

12.1.3 Número variable de parámetros

- 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();
}
```

12.1.3 Número variable de parámetros

- 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();
}
```

12.1.3 Número variable de parámetros

- 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 13 al 16 de la hoja de ejercicios

13. Envoltorios y autoboxing

- En el lenguaje Java los tipos predefinidos siempre son tipos **primitivos, nunca clases.**
- Hay ciertas clases predefinidas llamadas **envoltorios (*wrappers, en inglés),*** preparadas para estos tipos simples.
- Estas clases envoltorio son las que **se encargan de proveer, según el caso,** de:
  - Métodos de clase necesarios en la conversión de cadenas de texto al tipo adecuado.
  - Métodos de clase para la impresión con varios formatos.
  - Constantes estáticas con el valor mínimo y máximo de un tipo, etc.

13. Envoltorios y autoboxing

- Estos son los envoltorios **definidos en Java:**

Envoltorio	Tipo predefinido
Boolean	boolean
Character	char
Integer	int
Long	long
Float	float
Double	double

- Los tipos primitivos, que no se mencionan aquí, **byte y short** no disponen de envoltorios predefinidos. En sus casos se usa la clase **Integer.**

13. Envoltorios y autoboxing

- Ejemplos:

```
Character miLetra = new Character('H');
Integer miEntero = new Integer(1024);
Float miRealCorto = new Float("12.3E-2");
```

- El nombre del método para retornar el valor primitivo resulta de la concatenación del tipo simple con la palabra **Value,** esto es: **charValue()** en el envoltorio **Character, intValue()** en el envoltorio **Integer**, etc.

```
int n = miEntero.intValue();
```

13. Envoltorios y autoboxing

- El método **equals()** efectúa comparaciones del valor entre envoltorios:

```
Integer miEntero = new Integer(1024);
Integer otroEntero = new Integer(1024);


System.out.println("miEntero equivale a otroEntero "
                   +miEntero.equals(otroEntero));
// Devuelve: miEntero equivale a otroEntero true
```

13. Envoltorios y autoboxing

- Desde Java 1.5 una nueva característica se suma, **AutoBoxing y UnBoxing,** dirigida a hacer posible que el compilador Java transforme automáticamente tipos primitivos (básicos) en su **clase envoltorio correspondiente (*wrappers) y al revés.***
- ***El resultado*** es la generación del mismo código, pero autoboxing se salta las áridas instancias y conversiones que dificultan su lectura.

13. Envoltorios y autoboxing

- Veremos de esta manera la diferencia entre un código que utiliza autoboxing, y otro que no lo usa y necesita de la creación de instancias y conversiones.

Con Autoboxing	Sin Autoboxing
`int i`	`int i`
`Integer j`	`Integer j`
`i = 1`	`i = 1`
`j = 2`	`j = new Integer(2)`
`i = j`	`i = j.intValue()`
`j = i`	`j = new Integer(i)`

- Es preferible la utilización de **tipos primitivos** si no son necesarios los métodos de los envoltorios

14. Destrucción de objetos y liberación de memoria

- En Java, a diferencia de C++ y en otros lenguajes de programación orientada a objetos, **no existen destructores** a disposición del programador para liberar recursos.
- Para simplificar la gestión de la memoria es la misma máquina virtual de Java quien elimina los objetos.
- Si un objeto pierde sus referencias, y por tanto deja de ser accesible por el programa la máquina virtual de Java procede a eliminarlo.

14. Destrucción de objetos y liberación de memoria

- Ejemplo:

```
Dog lassie = new Dog ("Lasie",true);
lassie = null;
```

- El objeto se queda sin referencia, “lassie”, una vez que en la segunda sentencia esta apunta a null.
- Ya no será posible llamar al objeto, y la maquina virtual de Java evitará que siga ocupando memoria, reciclándolo o lo que es lo mismo, eliminándolo.

14. Destrucción de objetos y liberación de memoria

- **Garbage collector** (recolector de basura) es la denominación del sistema de liberación de memoria de Java.
- Se activa únicamente cuando encuentra una situación que lo requiera, como puede ser si hay escasez de memoria.
- Sin embargo, se puede proponer su activación a Java mediante la llamada **System.gc()**, por ejemplo en algún momento que se han dado gran cantidad de objetos por inservibles, para forzar la liberación de recursos.

14.1 Finalizadores en Java

- Para liberar automáticamente la memoria de objetos inservibles, el recolector de basura procede a ejecutar el **finalizador del objeto**, que se distingue por no tener valor de retorno ni argumentos, no poder ser static y por su nombre obligatorio **finalize().**
- El compilador proporciona un finalizador por **defecto a través de la clase Object** cuando no especificamos uno.
- Esta seria la sintaxis correspondiente:

```
protected void finalize throws Throwable (){ /* sin código */ }
```

14.1 Finalizadores en Java

- Es necesario **reescribir** el anterior método si queremos definir un finalizador en una clase.
- Una clase solo puede definir un finalizador, al contrario de lo que sucede con los constructores.
- En el cuerpo, además, es posible incluir la operación que deseemos en relación con el objeto a finalizar.

```
@Override
protected void finalize throws Throwable (){
        System.out.println("Sayonara baby");
}
```

14.1 Finalizadores en Java

- 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);
    }


}
```

14.1 Finalizadores en Java

- 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);

         }
    }

}
```

14.1 Finalizadores en Java

- Los finalizadores se suelen usar con el fin de liberar memoria, cerrar ficheros, conexiones, etc.
- Dado que el momento en el que los finalizadores se ejecutan no se puede elegir, es el recolector de basura quien procede a imponer el momento.
- Sin embargo, hay recursos cuya liberación es aconsejable efectuar explícitamente sin esperar al recolector de basura.
- No debe cerrarse en un finalizador, por ejemplo, una conexión a una base de datos.

14.1 Finalizadores en Java

- Se puede invocar explícitamente un finalizador así:

```
objeto.finalize()
```