Apuntes

de

Java

Tema 2: Programación orientada a objetos

TEMA 2: PROGRAMACIÓN ORIENTADA A OBJETOS

2.1.- ¿QUÉ ES LA POO?

Originalmente, la POO nació para sistemas de simulación, ya que:

- Pretende modelar el comportamiento de "objetos" reales
- Cada objeto real es un objeto en el programa
- Los objetos tienen características o propiedades y un comportamiento
- Interiormente están formados por un mecanismo que no necesitamos conocer una vez que está formados

Esta separación, abstracción, entre la visión externa y la interna marca el encapsulamiento. Un objeto integra datos, funciones e implementación.

Por ejemplo, del vídeo de casa no nos preocupa su electrónica, simplemente su mando a distancia. En las instrucciones pone "No abrir salvo Servicio Técnico". Esto se conoce como **ocultación**. Es similar al concepto de TAD.

2.2.- ALGUNOS ELEMENTOS FUNDAMENTALES DEL MODELO DE OBJETOS

Abstracción

Abstracción="Separar mentalmente"="representación de las características esenciales de algo sin incluir antecedentes o detalles irrelevantes"

Una abstracción denota las características esenciales de un objeto que lo distinguen de todos los demás tipos de objeto y proporciona así fronteras conceptuales nítidamente definidas respecto a la perspectiva del observador.

Una abstracción que describa un conjunto de objetos en términos de una estructura de datos y encapsulada u oculta y las operaciones sobre esa estructura la denominaremos *tipo abstracto de datos (TAD)*

Todos los lenguajes de programación permiten cierto grado de abstracción, el lenguaje ensamblador es una abstracción de la máquina, y lenguajes como C o Pascal son abstracciones del ensamblador. La aproximación que hace la OO va más allá y ofrece herramientas que permiten representar elementos en el dominio del problema. Esta representación es lo suficientemente general como para no estar limitado a algunos tipos de problemas concretos.

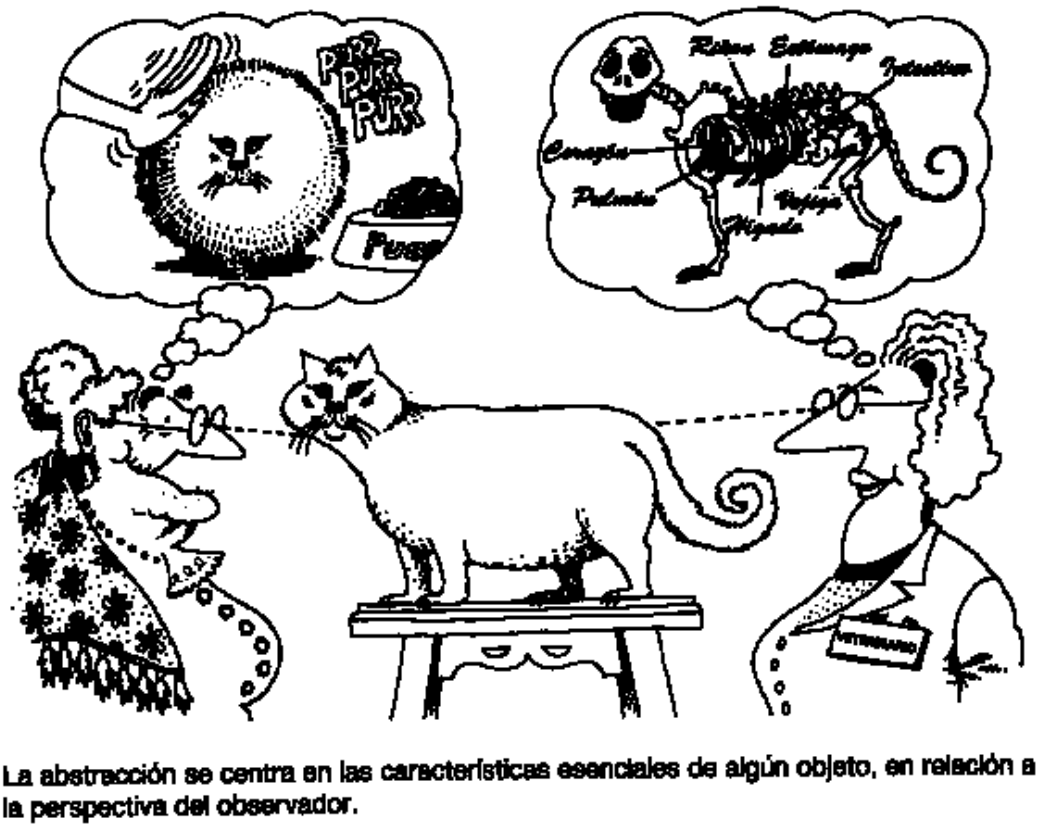


Figura 2.1: Abstracción

Clases y objetos

Unos de los mecanismos básicos de la POO son los objetos y las clases.

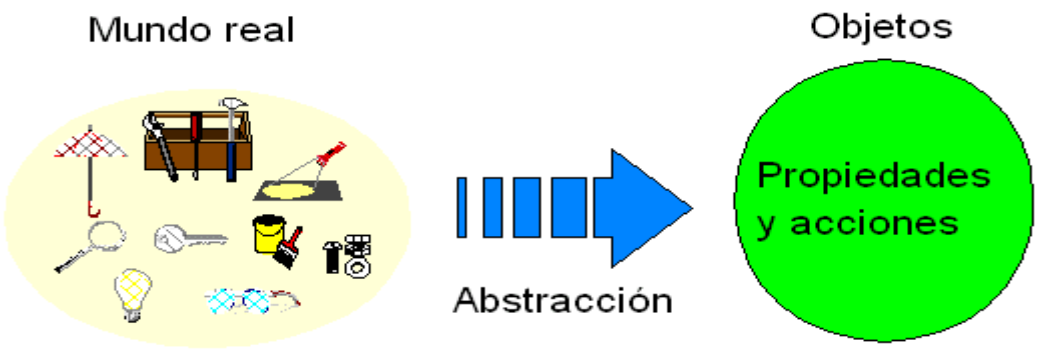


Figura 2.2: Abstracción de entidades del mundo real en objetos

Un programa orientado a objetos se compone sólo de **objetos**. Un objeto es una encapsulación genérica de datos y de los procedimientos para manipularlos. Un objeto tiene unos datos (*atributos*), que definen su estado, y una forma de operar con ellos (*métodos*), que definen su comportamiento.

Una **clase** se puede considerar como una plantilla para crear objetos de esa clase. Una clase describe los métodos y atributos que definen las características comunes a todos los objetos de esa clase.

Una clase es un tipo de objeto definido por el usuario. Un objeto es la concreción de una clase (instancia).

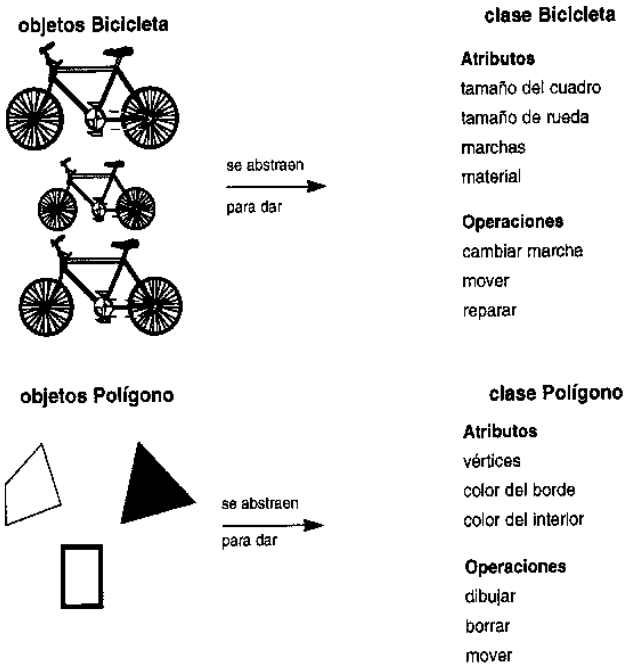


Figura 2.3: Abstracción de objetos en clases

Encapsulación u ocultamiento de información

"incluir dentro de un objeto todo lo que necesita, de tal forma que ningún otro objeto necesite conocer nunca su estructura interna."

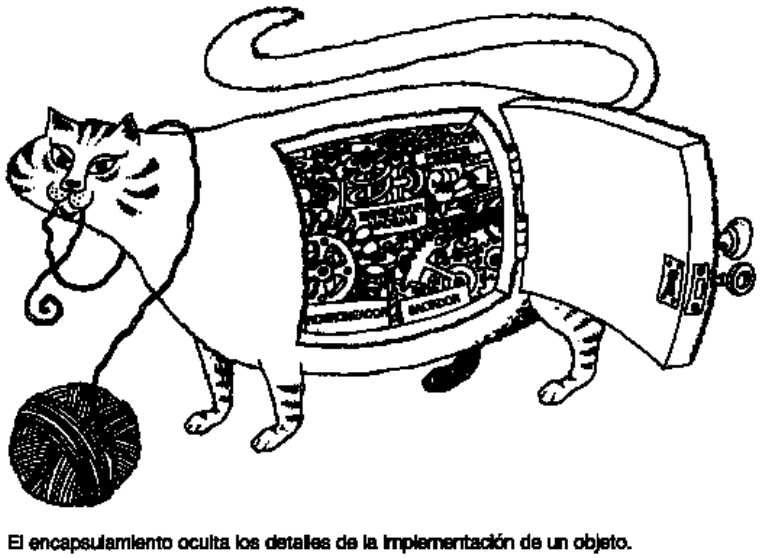


Figura 2.4: Encapsulamiento

Permite ver el objeto como una caja negra y manipularlo como una unidad básica, permaneciendo oculta su estructura interna.

En la encapsulación los datos o propiedades son privados y se accede a ellos a través de los métodos.

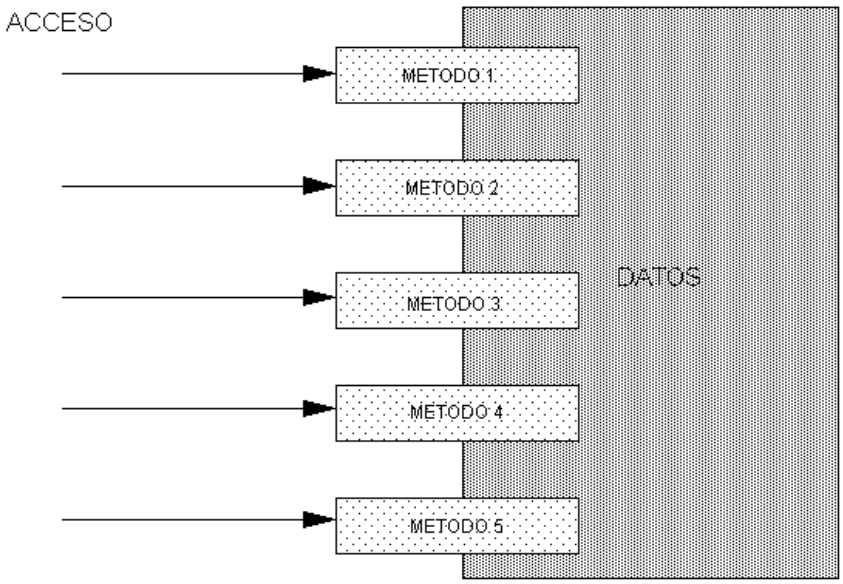


Figura 2.5: Acceso a los datos mediante métodos

2.3.- ESTRUCTURA GENERAL DE UN PROGRAMA EN JAVA

Un programa se compone de una o más clases. Uno de los métodos de la clase principal debe ser obligatoriamente `main( )`.

Un programa Java puede incluir:

- Declaraciones para importar clases de los paquetes.
- Declaraciones de clases.
- El método `main( )`.
- Métodos definidos por el usuario dentro de las clases.
- Comentarios del programa.

El método main() tiene la siguiente estructura:

```
public static void main(String [] args)
{
     bloque de sentencias….
}
```

Argumentos de la línea de comandos

El argumento args es un array de cadenas que permite introducir datos en forma de cadenas de caracteres mediante la línea de comandos. Puede que no haya argumentos en la línea de comandos; en cualquier caso es obligatorio especificar `String[ ]`.

Por ejemplo, si ejecutamos la siguiente línea:

```
c:\>java Nombres Luis Gerardo Fernando
```

El array contendrá la siguiente información:

```
args[0] → Luis
args[1] → Gerardo
args[2] → Fernando
```

2.4.- ATRIBUTOS Y MÉTODOS

Los **atributos** son las propiedades de un objeto, es decir, sus datos. Al definir un objeto, se indica el nombre y tipo de todos sus atributos.

Por ejemplo, para la clase Coche:

- velocidadMaxima de tipo entero
- consumo de tipo real
- matricula de tipo texto

En ejecución, en cada instancia de la clase se pueden usar los atributos de instancia (atributos).

Todas las instancias de una misma clase tendrán los mismos atributos (nombre y tipo), pero su espacio y valores serán independientes.

Hay otro tipo de atributo, los atributos de clase, que son los atributos que sólo existen una vez para cada clase (p. ej. numDeCoches).

Atributos de instancia: cada instancia creada tiene sus propios valores y almacenamiento en memoria.

Atributos de clase:

- Se crea un atributo para toda la clase
- Todos los objetos acceden a ese atributo, se comparte.
- Lleva el modificador *static*

El comportamiento viene determinado por funciones o **método**s, que indican qué puede hacer una clase, cómo se cambia de estado, cómo se relaciona con otras.

Por ejemplo, para la clase Coche:

- Arrancar
- ParararMotor
- Acelerar
- Decelerar
- Girar
- Frenar
- CambiarMarcha

Los métodos son funciones definidas en la clase, que operan sobre instancias de esa clase.

Además, también hay métodos de instancia y métodos de clase.

Métodos de instancia:

- Método asociado a una instancia en particular
- Cuando se llama a un método de instancia, éste tiene acceso a los datos contenidos en la instancia a la que está asociada
- <objeto>.<nombre_metodo>(arg1, arg2,...,argN)

Métodos de clase:

- Se asocian a toda la clase, no a una instancia particular
- Tienen un ámbito más amplio que los métodos de instancia, manejan información significativa para toda la clase
- Se declaran como *static*
- Los métodos de clase no pueden sobreescribirse
- <nombre_clase>.<nombre_metodo>(arg1, arg2,...,argN)

En P.O.O. se suele hablar de mensaje. Cuando alguien llama al método X de un objeto, se dice que le está pasando el mensaje X. No sólo el programador llama a los métodos, los

métodos pueden enviarse mensajes entre sí y también el sistema puede ejecutar algunos métodos, por ejemplo, la gestión de ratón o teclado.

En la ejecución de un programa orientado a objetos se realizarán fundamentalmente tres cosas:

1. Crear los objetos necesarios
2. Los mensajes enviados a unos y otros objetos darán lugar a que se procese internamente la información
3. Cuando los objetos no son necesarios, serán borrados, liberándose la memoria ocupada por los mismos

2.5.- SINTAXIS BÁSICA DE JAVA

Comentarios

En Java, al igual que en C, hay dos tipos de comentarios básicos: los comentarios de línea y los comentarios de varias líneas. Los primeros comienzan con la doble barra "// " y sólo abarcan una línea. Los de varias líneas comienzan con los caracteres "/* " y terminan con "*/ " (no se pueden anidar).

Javadoc

Existe otro tercer tipo de comentario denominado **comentario de documentación**, que pueden ser extraídos a archivos HTML utilizando *javadoc*. Es necesario introducirlos entre los símbolos /**...*/.

Hay varias marcas especiales que se pueden incluir en los comentarios javadoc. Algunas de ellas son `@author, @param, @returny @exception`.

A continuación se muestra una clase comentada para generar documentación utilizando `javadoc`:

```
/**
* Clase de prueba para ver el funcionamiento de javadoc
*/
public class A
{
     private int x;
     /**
     * Constructor con un argumento que inicializa el valor de la
               propiedad.
     * @param valor con el que inicializar la propiedad x.
     */
     public A (int a){
          x = a;
     }
     /**
     * @param no tiene parámetros
     * @return devuelve el valor de la propiedad
     */
     public int getX(){
```

```
        return x;
    }
    public static void main(String[] args)
    { }
}
```

Para obtener la documentación hay que utilizar el programa javadoc, seguido del fichero cuya documentación queremos obtener, en el ejemplo anterior sería:

```
c:\>javadoc A.java
```

Y el resultado sería un conjunto de páginas HTML con la siguiente información:

Class A

```
java.lang.Object
  |
  +--A
```

---

public class **A**

extends java.lang.Object

Clase de prueba para ver el funcionamiento de javadoc

A

`public A(int a)`

Constructor con un argumento que inicializa el valor de la propiedad.

Parameters:

`valor` - con el que inicializar la propiedad x.

getX

`public int getX()`

Parameters:

no - tiene parámetros

Returns:

devuelve el valor de la propiedad

---

main

`public static void main(java.lang.String[] args)`

Tipos

Java es un lenguaje con **control estricto de tipos**, lo cual significa que cada variable debe declararse con un tipo y antes de ser utilizada.

Los tipos principales son los **primitivos** y las **clases**. Entre los primeros tenemos los tipos char, boolean, enteros (byte, short, int, long) y reales (float, double). Se pueden hacer conversiones entre tipos enteros, reales y char. Las conversiones pueden ser **implícitas**:

```
5 + 2.3         [5 → 5.0]
double x = 5 / 2.0   [5/2.0 → 5.0/2.0]
real = entero   [real → entero]
```

O **explícitas** (casting):

```
(int) 5.4
double x = 5 / (double) 2
```

Todo lo demás en Java son clases, bien definidas por el usuario o bien definidas en el API de Java. Las clases se gestionan mediante referencias, que apuntan a la instancia de una clase o a null.

```
ClaseX referencia = InstanciaClaseX;
ClaseX referencia = null;
ClaseY referencia; /* No se ha creado la instancia de la clase, es sólo una
                referencia a null*/
```

Operadores

Los operadores en Java se clasifican en:

- Aritméticos (enteros y reales)

  ```
  + - * /
  ```

- Aritméticos (enteros)

  ```
  % (módulo o resto)
  ```

- Relacionales (char, enteros y reales → boolean)

  ```
  <, <=, >, >=, ==, !=
  ```

- Booleanos

  ```
  And (&&) Or (||) Not(!)
  ```

- Manipulación de bit (enteros)

  ```
  And (&), Or (|), Xor (^), Not(~), Desplazamiento (<<, >>, >>>)
  ```

- Incremento y decremento (forma prefija y postfija)

  ```
  ++, --
  ```

- Asignación

=

2.6.- SINTAXIS PARA LA CREACIÓN DE CLASES

Declaración de clase:

[Modificadores de Clase] **class** *Identificador* [Super] [Interfaces] CuerpoClase

[Modificadores de Clase]: **public** / **abstract** / **final**

[Super]: **extends** TipoClase

[Interfaces]: **implements** ListaTipoInterface

```
public class Circulo
{
     public double x, y; // The coordinates of the center
     public double r; // The radius
     // Methods that return the circumference and area of the circle
     public double circunferencia()
     {
          return 2 * 3.14159 * r;
     }
     public double area()
     {
          return 3.14159 * r*r;
     }
}
```

2.7.- MODIFICADORES DE MÉTODOS Y DE ATRIBUTOS

Modificadores de métodos y su ámbito

Los modificadores de métodos pueden dividirse en dos grupos: basado en si afectan o no el ámbito de los métodos. El ámbito en lenguajes de programación se refiere a la región del programa en el que un elemento puede ser accedido: clase, método o variable.

Modificadores que no afectan al ámbito

Modificador de método	**Efecto**	**Utilizado para...**
Final	El método no puede ser sobreescrito en una subclase	Métodos que no se quiere que cambien
Static	Es un método de clase	Métodos que no se basan en datos internos, particulares de una instancia
Native	El cuerpo del método se escribirá en C y se enlazará en el intérprete	Métodos específicos para una plataforma o para utilizar código existente
Abstract	El método no está definido en la clase. Debe definirse en una subclase	Métodos de propósito general que no tienen un comportamiento por defecto significativo. Se definen completamente en subclases
synchronized	El método bloqueará la instancia (o la clase, si es un método de clase)	Métodos que interfieren en la ejecución de threads

Modificadores que afectan al ámbito

Modificador de método	**Efecto**	**Usado para definir…**
Public	El método puede ser accedido desde cualquier clase	La interfaz externa de la clase
Private	El método sólo puede ser accedido por métodos de la misma clase	Métodos internos y sólo relevantes para una clase particular (e irrelevante para usuarios de la clase)
friendly (*no se especifica explícitamente)*	El método puede ser accedido por métodos de la clase o métodos de otras clases en el mismo paquete que la clase	Métodos que serán accedidos por otras clases relacionadas
protected	El método puede ser accedido por métodos en subclases de la clase	Métodos que serán accesibles desde las subclases

Ambito de un método *public*

Se le puede llamar desde cualquier parte del programa. Se utiliza los métodos que definen el interfaz externo de las clases.

Ambito de un método *private*

Sólo puede ser llamado por métodos de la misma clase. Los usuarios de la clase no necesitan conocer la existencia de estos métodos.

Ambito de un método *friendly*

Por defecto, los métodos de java son friendly. Se puede llamar desde otros métodos en la clase o por métodos de otras clases en el mismo paquete.

Ambito de un método *protected*

Es como un método *friendly* y además puede ser accedido por las subclases de la clase en las que el método está definido.

Modificadores de atributos

Modificador de atributo	Efecto
Public	El atributo puede ser accedida desde cualquier clase
Private	El atributo sólo puede ser accedida por métodos de la misma clase
protected	El atributo puede ser accedida por subclases de la clase donde está definida y por clases del mismo paquete
Static	Es un atributo de clase
Final	El valor del atributo no puede ser modificado
transient	El atributo no es parte del estado persistente del objeto
volatile	El atributo puede ser modificada asíncronamente. El compilador la mantendrá en memoria

Ambito de las variables y su duración en java

Tipo de variable	Ambito	Duración
De instancia	private- sólo por métodos de la clase public- cualquier clase friendly- esta clase o cualquiera en el paquete protected- cualquier subclase de la actual, o cualquier claes dentro del mismo paquete	Desde que se crea la instancia y hasta que no hay más referencias a esa instancia
De clase	Sujetos a los mismos criterios que las variables de instancia	Desde que la clase es cargada y hasta que no haya más referencias a la clase
Locales	En el trozo de código dondes están definidas	El tiempo que el trozo de código está activo

Valores de inicializacón de los tipos primitivos

Si la variable es de tipo...	El compilador de Java lo inicializa a...
Float	0.0f
Double	0.0d
int	0
Byte	0
Short	0
Char	'\u000'
Long	0L
boolean	false
todos los demás	null

Constantes

Java no tiene constantes como en otros lenguajes. Para ofrecer una capacidad similar se incluye el modificador de variables **final**. Cuando se aplica a una variable indica que su valor no puede variar.

2.8.- CONSTRUCTORES

Un tipo especial de método es el llamado **constructor**. Este es el método al que llama new cuando se crea una instancia nueva.

Aunque no lo definamos, siempre hay un constructor por defecto, sin parámetros.

Se puede definir el constructor como un método más de la clase, con el mismo nombre que la clase y sin que se indique ningún tipo de retorno (ni siquiera void). Por ejemplo:

```
class CocheElectrico extends Coche
{
     int autonomia; /* Constructor que asigna a la autonomía el valor
                       100 */
     CocheElectrico()
     {
          autonomia = 100; /* Supongamos que por omisión, la autonomía es
               de 100 km. */
     }
     /* Constructor que recibe un parámetro que indica el valor al que
               asignar la autonomía. */
     CocheElectrico( int valorAutonomia )
     {
          autonomia=valorAutonomia;
     }
}
```

También se pueden definir varios constructores para la misma clase, siempre y cuando tengan diferente número y/o tipo de parámetros. Depende de la llamada al constructor que se haga, se ejecutará uno u otro constructor. En el ejemplo anterior uno de los constructores no tiene ningún parámetro, y se le llamaría: *new CocheElectrico( );* El otro constructor recibe un parámetro de tipo entero y se le llamaría: *new CocheElectrico(auton);* (siempre y cuando auton fuese de tipo int o compatible)

```
class CocheElectrico extends Coche
{
     int autonomia; /* Constructor que asigna a la autonomía el valor 100
               */
     CocheElectrico()
     {
          autonomia = 100; /* Supongamos que por omisión, la autonomía es
               de 100 km */
     }
     /* Constructor que recibe un parámetro que indica el valor al que
               asignar la autonomía */
     CocheElectrico( int valorAutonomia )
     {
          autonomia=valorAutonomia;
     }
}
```

Otro ejemplo de constructores:

```
public class Circulo
{
      double x, y;
      double radio;
      public Circulo()
      {
            x = 0;
            y = 0;
            radio = 0;
}
public Circulo (double cX, double cY, double r)
{
      x = cX;
      y = cY;
      radio = r;
}
}
```

2.9.- SINTAXIS PARA INSTANCIAS. LLAMADAS Y CREACIÓN.

Ahora que ya se ha creado la clase Circulo, si queremos hacer algo con ella debemos crear un objeto concreto, una instancia de la clase Circulo. Al definir la clase Circulo se ha creado un nuevo tipo de dato. Ahora se puden crear variables de ese tipo:

```
Circulo tuCirculo;
```

Pero tuCirculo no es más un nombre que *hace referencia* a un objeto Circulo, no es el objeto en sí. En Java todos los objetos deben ser creados dinámicamente. Esto casi siempre se hace con la palabra clave **new**:

```
Circulo tuCirculo;
tuCirculo = new Circulo();
```

o también se puede hacer directamente:

```
Circulo tuCirculo = new Circulo();
```

Ahora hemos creado una instancia de la clase Circulo-un objeto circulo-y se lo hemos asignado a una variable tuCirculo, que es de tipo Circulo. Para acceder a sus atributos:

```
tuCoche.x = 20;
tuCoche.y = 30;
tuCoche.r = 10;
```

Con la misma sintaxis que se accede a los atributos se accede a los métodos:

```
tuCirculo.area( );
tuCirculo.circunferencia( );
```

Otro ejemplo:

```
Circulo c = new Circulo();
double a;
c.r = 2.5;
a = c.area();
```

Observar la siguiente línea. No se llama a los métodos así:

```
a = area(c);  //Incorrecto
```

Sino:

```
a = c.area(); //Correcto
```

Esto es por lo que se llama programación "orientada a objetos": el centro es el objeto, no la llamada a la función. Esta es probablemente la característica más importante del paradigma de orientación a objetos. Se invocan los métodos del objeto o bien se accede a sus propiedades para lectura/escritura.

Resumiendo podríamos decir que el acceso a los atributos y métodos de un objeto se hacen según el siguiente esquema:

```
<referencia>.<nombre_metodo>(<argumentos>)
<referencia>.<nombre_atributo>
```

Siempre y cuando los modificadores de esos métodos permitan el acceso a ellos.

2.10.-MÉTODOS SET/GET

Ya hemos visto cómo se puede acceder a los atributos para modificar su valor. Sin embargo, existe otra forma de acceder a ellos, que es la que se suele emplear para así mantener el concepto de **encapsulación**: los métodos get y set.

El método getXXX se utiliza para devolver el valor del atributo XXX. Por otra parte, el método setXXX se utiliza para asignar un valor al atributo XXX. Es decir, los métodos get y set son métodos de acceso a los atributos de un objeto. En la medida de lo posible hay que evitar el acceso directo a los atributos de un objeto, el programador debe proporcionar los métodos necesarios para acceder de una forma controlada a los mismos.

Siguiendo el ejemplo de la clase Circulo, podríamos obtener los siguientes métodos:

Ejemplo de clase Circulo sin set/get

```
public class Circulo
{
     //ATRIBUTOS DE INSTANCIA
     public double x, y; // Coordenadas del centro
     public double r; // The radius
     //ATRIBUTOS DE CLASE
     static int cont=0;
     // METODOS DE INSTANCIA
     Circulo (double x1, double y1, double r1)
     {
          cont++;
          x=x1;
          y=y1;
          r=r1;
     }
     public double circunferencia()
     {
          return 2 * 3.14159 * r;
     }
     public double area()
     {
          return 3.14159 * r*r;
     }
     //METODO DE CLASE
     public static void NumCirculos() {
          System.out.println("El número total de círculos es: " + cont);
     }
     //FUNCION PRINCIPAL
     public static void main(String args[])
     {
          Circulo c1=new Circulo(1,1,1);
          System.out.println("La circunfernecia es: " +
                    c1.circunferencia());
          System.out.println("El área es: " + c1.area());
          Circulo.NumCirculos();
          System.out.println("===============");

          Circulo c2=new Circulo(2,2,2);
          System.out.println("La circunfernecia es: " +
               c2.circunferencia());
          System.out.println("El área es: " + c2.area());
          Circulo.NumCirculos();
          System.out.println("===============");

          Circulo c3=new Circulo(3,3,3);
          System.out.println("La circunfernecia es: " +
               c3.circunferencia());
          System.out.println("El área es: " + c3.area());
          c3.NumCirculos();
          System.out.println("===============");
     }
}
```

Ejemplo de clase Circulo con set/get

```
public class Circulo2
{
     //ATRIBUTOS DE INSTANCIA
     private double x, y; // Coordenadas del centro
     private double r; // The radius

     public void setR(double r1)
     {
          r=r1;
     }
     public double getR()
     {
          return r;
     }
     public void setCentro(double x, double y)
     {
          this.x=x;
          this.y=y;
     }
     public double getX()
     {
          return x;
     }
     public double getY()
     {
          return y;
     }
     public static void main(String args[])
     {
          Circulo2 c1 = new Circulo2();
          c1.setCentro (1,1);
          System.out.println("El centro es: " + c1.getX() + ", " +
               c1.getY());
          c1.setCentro (2,2);
          c1.setX(4);
          System.out.println("El centro es: " + c1.getX() + ", " +
               c1.getY());
     }
}
```

Propuesta de ejercicio, clase Autobús

Crear una clase en la que se realicen las siguientes operaciones:

1. Se crearán tres autobuses
2. Cuando se cree cada uno de ellos se nos indicará por pantalla si su nivel de gasolina y aceite son adecuados (el intervalo lo elige el alumno). Para ello habrá sido necesario inicializar estos datos (como cada uno crea conveniente)
3. Se harán varias reservas y anulaciones de plazas para cada autobús. Si no hay plazas suficientes se avisará al usuario de ello.
4. También se deberá consultar el número de plazas disponibles para cada autobús.

2.11.-HERENCIA

Construcción del lenguaje que permite relacionar unas clases con otras, de forma que las subclases pueden heredar las características de las superclases

La herencia diferencia TADs de POO. Por ejemplo, consideremos las clases Coche y Ornitorrinco. Son muy diferentes, no tienen nada en común. Cada clase se definiría por su cuenta. Pero, por otro lado, pensemos en las clases Coche y Camión. Estas clases tienen muchas cosas en común. Muchos atributos serán parecidos o iguales, al igual que ocurre con los métodos.

Con la programación convencional tendríamos que reescribir todo de nuevo en Camión y en Coche. Sin embargo, la herencia nos permite reutilizar código indicando cómo una clase se diferencia de otra definida previamente, tomando parte de sus características (o todas) y cambiando o ampliando el resto.

La herencia funciona de un modo **estrictamente jerárquico**. Cada clase tiene **una** superclase, o clase padre y varias subclases o clases hijas. Los atributos y métodos se heredan **hacia abajo** en toda la jerarquía.

En Java **sólo hay herencia simple**, es decir sólo una clase es padre. En otros lenguajes, como C++, existe la herencia múltiple.

Toda clase (excepto la especial Object) es subclase de otra. **Object** es la cima de la jerarquía de clases en Java. Las clases superiores describen un comportamiento muy **general** (abstracto); las inferiores son más **específicas** (concretas).

La ventaja fundamental de la herencia es la **reutilización de código**. Para ello, el diseño de la jerarquía debe ser preciso.

Por ejemplo, Coche y Camión:

- ¿Cuál es más general?

- Estas clases no se incluyen mutuamente, pero sí podemos definir una clase más general que recoja el comportamiento común de coches y camiones. Es decir, definiríamos una clase que fuera padre de ambas.
- ¿Cómo la llamamos? Vehículo es un buen nombre.

Esta jerarquía podría ser mucho más compleja, ya que se podría añadir otra clasificación de tipos de vehículos, por ejemplo, considerando el tipo de tracción del vehículo (aéreos, terrestres, marinos,...). Estas jerarquías se llaman también taxonomías.

Sintaxis de Java para la herencia

```
public class Coche extends Vehiculo
{
     …
}
```

La clase de la que hereda Coche (Vehiculo) debe ser accesible:

- Estar en el mismo paquete que la clase Coche
- Estar declarada en otro paquete como public
- No estar declarada como final

La claúsula **extends** indica la superclase directa de la clase que se está declarando. La implementación de la clase que se está declarando deriva directamente de la implementación de su superclase directa. Si no se indica ninguna claúsula extends, la superclase directa de la clase que se está declarando es la clase java.lang.Object.

La clase que se indica en la claúsula extends debe ser accesible, si no se produciría un error de compilación:

- Todas las clases del mismo paquete son accesibles
- Las clases de otros paquetes son accesibles si el sistema permite el acceso al paquete y la clase está declarada como **public**
- Si la clase de la claúsula extends es **final** se produce un error de compilación, las clases finales no pueden tener subclases

La relación de subclase es transitiva. Una clase A es subclase de C si se cumple alguna de las siguientes condiciones:

- A es subclase directa de C
- Existe alguna clase B, de tal forma que A es subclase de B y B es subclase de C

```
class Punto
{
     int x, y;
}
class PuntoColoreado extends Punto
{
     int color;
```

```
}
final class PuntoColoreado3D extends PuntoColoreado
{
     int z;
}
```

- Punto es subclase de java.lang.Object
- java.lang.Object es superclase de la clase Punto
- PuntoColoreado es subclase de Punto
- La clase Punto es superclase de la clase PuntoColoreado
- La clase Punto es superclase de la clase PuntoColoreado3D
- La clase PuntoColoreado es subclase de la clase Punto
- La clase PuntoColoreado es superclase de la clase PuntoColoreado3D
- La clase PuntoColoreado3D es subclase de la clase PuntoColoreado
- La clase PuntoColoreado3D es subclase de la clase Punto

Modificadores de clase

Pueden ser **public**, **abstract** o **final.**

No pueden aparecer más de una vez en una declaración de clase. Si aparecen varios pueden ir en cualquier orden.

abstract

Una clase abstracta es una clase incompleta. Sólo las clases abstractas pueden tener métodos abstractos, métodos declarados pero no implementados. Si una clase no abstracta contiene un método abstracto se produce un error de compilación. Una clase es abstracta si cumple alguno de los siguientes requisitos:

- Explícitamente contiene la declaración de un método abstracto
- Hereda un método abstracto de su superclase directa

Una clase abstracta NO PUEDE SER INSTANCIADA, se produciría un error de compilación.

```
abstract class Punto
/* Debe declararse abstract porque tiene la declaración de un método
abstracto */
{
     int x=1, y=1;
     void mover(int dx, int dy)
     {
          x += dx;
          y += dy;
          avisar();
     }
     abstract void Dibujar();
}
abstract class PuntoColoreable extends Punto
/* Como hereda un método abstracto y no lo implementa debe ser declarado
```

```
abstract */
{
     int color;
}
class PuntoSimple extends Punto
//Al implementar el método abstracto la clase no es abstracta
{
     void dibujar()
     {
          System.out.println("(" + x + ", " + y + ")");
     }
}
```

final

Una clase se puede declarar como final si su definición está completa y no se necesitan o no se quieren crear subclases. Se produce un error de compilación si el nombre de una clase final aparece en la claúsula extends de alguna clase. Esto implica que las clases final no pueden tener subclases. Se produce un error de compilación si una clase es declarada *abstract* y *final,* porque la implementación de esa clase nunca podría estar completa. **NO SE PUEDE HEREDAR DE UNA CLASE FINAL.**

Modificador *final* aplicado a...

Clases	no puede tener subclases
métodos	no puede ser sobreescrito
atributos	su valor no puede ser modificado, se convierte en un atributo constant

Preguntas de repaso sobre modificadores

1.- ¿Qué es una clase abstracta?

2.- ¿Cuándo se dice que una clase es abstracta?

3.- ¿Cuál es la utilidad de una clase abstracta?

4.- ¿Se puede crear una subclase de una clase abstracta?

5.- ¿Se puede crear una instancia de una clase abstracta?

6.- ¿Qué quiere decir que una clase/método/atributo es final?

7.- Si D es subclase directa de B, y B y C son subclase directas de A. ¿Cuáles son subclases de A?

8.- Indicar cuáles de los siguientes modificadores de método influyen en el ámbito del mismo: protected, final, public, private, static, abstract, friendly

9.- ¿Qué significa que un método es *public*?

10.- ¿Qué significa que un método es *protected*?

2.12.-REFERENCIAS THIS Y SUPER

this

- Es la referencia al propio objeto sobre el que se está trabajando
- No hace falta usarlo para acceder a los atributos y métodos propios
- A veces es necesario cuando se quiere pasar la referencia al propio objeto com parámetro de otro método

super

- Permite referenciar a la clase padre
- Se emplea cuando se ha redefinido un método y se quiere acceder al método del padre (si no se usara esta palabra reservada se llamaría al método de la propia clase y no habría forma de invocar el del padre)

```
class Rectangulo
{
      void saludar()
      {
            System.out.println(“Hola, soy Rectanbulo, y saludo”);
      }
}
class Cuadrado extends Rectangulo
{
      void saludar() //Método redefinido
      {
            System.out.println(“Hola, soy Cuadrado, y saludo”);
      }
      void saludos()
      {
            saludar(); //Hola, soy Cuadrado, y saludo
            this.saludar(); //Hola, soy Cuadrado, y saludo
            super.saludar();//Hola, soy Rectangulo, y saludo
      }
      public static void main(String args[])
      {
            Cuadrado c1 = new Cuadrado();
            c1.saludos();
      }
}
```

2.13.- LA CLASE STRING

- El String (en Java) no es un tipo primitivo, es una clase. Sin embargo, Java le ofrece un tratamiento especial
- Los literales de la cadena van entre comillas dobles

```
String saludo = "Hola"; /* Se está generando automáticamente un objeto de
                la clase String, es lo mismo que hacer:*/
String saludo = new String("Hola");
```

- Java permite la concatenación de cadenas mediante el +

```
System.out.println("Hola" + saludo);
```

- Si se concatena una cadena con otro tipo, este último se convierte automáticamente a cadena

```
int valor = 5;
System.out.println("Valor = " + valor);
```

2.14.- ARRAYS

Los arrays son **objetos** en Java, por lo tanto necesitan una referencia para ser manipulados. Esto llevará consigo importantes mejoras con respecto a los arrays convencionales.

Declaración y Creación

Si el tipo base de un array es T, el tipo del array se escribe T[ ]. Aunque, similarmente a C, se permite que los corchetes se indiquen también detrás del nombre del atributo.

```
int[] v;
String[] vecString;
```

Cuando se declara un atributo de tipo array no se crea realmente el array ni se asigna espacio para sus componentes. Simplemente se crea la referencia.

Si no se inicializa el array originalmente, la variable contendrá null. Por ejemplo las siguientes declaraciones no crean arrays:

```
int[] vi; // vector de enteros
Object[] vo; // vector de objetos
float r[][]; // array bidimensional (vector de vectores) de float
float[][] s; // lo mismo
```

Para crear los arrays debe llamarse a **new** igual que en las instancias, pasando como tipo, ahora sí, el tamaño específico que queremos crear en cada uno de ellos.

También podemos inicializar en la misma creación un array con una lista de valores entre llaves separados por comas y entonces no hace falta el new (en este caso, los valores indicados marcan la longitud):

```
String[] lisStrings = new String[5];
String[] lisString2 = { "vector", "de", "strings" };
double[] vecReales = { 4.5, 3, 18.3, 27, 0.15 };
Coche[] vecCoches = new Coche[5][3];
Coche[] vecCoche2 = { new Coche(1), null, null, new Coche(17) };
```

Estas declaraciones sí que inicializan el valor de los arrays (no así cada componente, salvo en el caso de las listas de valores entre llaves).

Cuando estamos declarando un array de un tipo primitivo (int, float, etc.), es suficiente con la sentencia:

```
int[] v = new int[10];
```

Ya hemos creado un array de 10 enteros. No pasa lo mismo cuando creamos un array de una clase, con la siguiente sentencia sólo creamos una estructura, que más adelante contendrá objetos de la clase especificada, pero por el momento su contenido es *null:*

```
String[] v=new String[10];
```

Para crear arrays de una clase hay que llegar hasta el **constructor** de la misma:

```
for (int i = 0; i<10; i++)
     v[i] = new String( );
```

Ahora sí tenemos un array de 10 objetos de tipo String.

Las características generales son las siguientes:

- Java controla absolutamente el acceso a las posiciones que existan (nunca se podrá acceder "fuera" del array).
- Los arrays tienen longitud variable. Esto es, el tipo del array no viene determinado por el número de elementos, simplemente por su número de índices. Cuando se crea el array con *new* es cuando se determina el número de elementos del array.
- Pueden tener el número de índices que se desee.
- Pueden contener cualquier elemento, de tipo primitivo u objetos de una clase.
- Cada array de Java se considera descendiente de Object.
- Para acceder a los elementos de un array se usa la notación clásica de arrays

  ```
  v[0]  = 5;
  int a  = v[2];
  ```

- El rango de un array está entre 0 y N-1 (siendo N el tamaño del array)

Un atributo de la clase que se necesita a menudo es **length**:

- length indica siempre el tamaño del array. Puede ser cero (en cuyo caso se dice que el array está vacío).

- Podremos acceder a cualquier posición entre 0 y (length - 1).

Además todo array permite también el uso del método público clone() que duplica el array. Hablaremos en futuros capítulos de este método clone y de su semántica, bastante importante en Java.

Ejemplo de creación de arrays de String

```
class PruebaArrays
{

     public static void main(String[] args)
     {
          int[] p = new int[10];
          String[] pp = new String[10];

          System.out.println("Longitud: " + p.length);
          for (int i=0; i < p.length; i++)
               System.out.println(" " + p[i]);
          System.out.println("Longitud: " + pp.length);
          for (int i=0; i<pp.length;i++)
               System.out.println("  " + pp[i]);
          for (int i=0; i<pp.length; i++)
               pp[i]=new String(""+ i);
          for (int i=0; i<pp.length; i++)
               System.out.println(" " + pp[i]);
     }
}
```

Una vez que un array ha sido creado, su longitud ya no cambia. Para cambiar la longitud del array, debe crearse otro diferente y asignarlo.

Acceso a arrays

Los arrays pueden ser accedidos por valores enteros: char, byte, short o int son tipos válidos ya que se convierten automáticamente a enteros int. Los long usados como índices producen un error de compilación. Todo índice se controla en ejecución, si se accede por debajo de cero o por encima de la longitud del array se genera una excepción IndexOutOfBoundException.

Arrays multidimensionales

Los arrays multidimensionales no son más que arrays (vectores) donde cada elemento es a su vez otro vector, y así sucesivamente hasta que la última dimensión finalmente representa valores.

- El número de dimensiones lo indica el número de corchetes

```
int [][][]v;
float [][]x;
```

- A la hora de crear el array se indicará el tamaño de cada dimensión

```
V1 = new int[10][2];
V2 = new float[6][4][2];
```

- Para acceder a un elemento del array multidimensional se deben especificar todos sus índices

```
V[1][4][0] = 4;
V1[5][2] = 3+2;
```

- Los arrays multidimensionales se pueden inicializar en el momento de su declaración, tener especial cuidado con la inicialización de arrays de objetos

```
static int[][] a = { {1,2,3}, null, {5,6,7,8,9,} };
Coche[][] b = { null, { new Coche(), new Coche() }, null };
```

Ejemplo de Arrays

¿Qué hace el siguiente código?¿qué se visualiza por pantalla?

```
class QueHago
{
     public static void main(String[] args)
     {
          String [][] arrayOfStringArrays = new String[5][];
          for (int i=0; i < arrayOfStringArrays.length; i++)
          {
               String[] nuevoArray = new String[10];
               ArrayOfStringArrays[i] = nuevoArray;
               for (int j = 0; j<arrayOfStringArrays.length; j++)
                    nuevoArray[j] = new String (“(“ + i + “, “ + j + “)”);
          }
          for (int i=0;i < arrayOfStringArrays.length; i++)
               for (int j = 0; j< 10; j++)
                    System.out.println(arrayOfStringArrays[i][j]);
     }
}
```

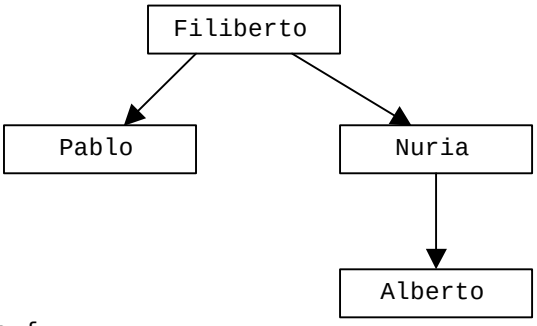
Ejemplo de utilización de arrays

```
class HolaConTiempo {
     static int contInstancias = 0;
     static HolaConTiempo[] instancia = new HolaConTiempo[10];
     int num;
     long tiempoCreacion;
     HolaConTiempo()
     {
          num = ++contInstancias;
          tiempoCreacion = System.currentTimeMillis();
     }
     void saludar()
     {
          System.out.println( "¡Hola! Soy la instancia " + num + " y he
               nacido en el milisegundo " + tiempoCreacion );
     }
     public static void main (String args[])
     {
          // Creamos las 10 instancias
```

```
            for (int i = 0; i < 10; i++)
                instancia[i] = new HolaConTiempo();
            // Que saluden las 10 después
            for (int i = 0; i < 10; i++)
                instancia[i].saludar();
        }
}
```

2.15.-ENCADENAMIENTO DE OBJETOS

Las clases pueden incluir, y a menudo lo hacen, atributos que a su vez tienen tipo clase, o métodos que devuelven objetos. Los objetos se pueden encadenar para hacer llamadas, aunque al principio resulte extraño leerlo. Por ejemplo:




```
class Persona {
      Persona padre; //Atributo de tipo clase
      String nombre;
      Persona( Persona p, String s )
      {
            padre = p;
            nombre = s;
      }
      Persona getPadre()
      {
            return padre;
      }
      String saluda()
      {
            return nombre;
      }
      public static void main (String args[])
      {
            Persona filiberto = new Persona( null, "Filiberto");
            Persona pablo = new Persona( filiberto, "Pablo" );
            Persona nuria = new Persona( filiberto, "Nuria" );
            Persona alberto = new Persona( nuria, "Alberto" );
            //Hemos creado el siguiente árbol genealógico
            System.out.println( "El nieto es: " + alberto.saluda() );
            //Alberto
            System.out.println( "La madre es: " + alberto.getPadre().saluda()
                  ); //Nuria
            System.out.println( "El abuelo es: " +
                  alberto.getPadre().getPadre().saluda() ); //Filiberto
```

```
        }
    }
```

- **alberto:** es un objeto de tipo Persona por lo que se puede acceder a sus atributos (padre y nombre) y a sus métodos (getPadre y saluda).
- **alberto.padre:** a su vez también es un objeto de tipo Persona, por lo que se aplica lo dicho en el caso anterior
- **alberto.getPadre:** devuelve un objeto de tipo Persona (ídem).

Ejemplo de encadenamiento de objetos

```
class Punto
{
     private int x, y;
     public Punto(int x, int y)
     {
          this.x = x;
          this.y = y;
     }
     public int getX()
     {
          return x;
     }
     public int getY()
     {
          return y;
     }
}
public class Cuadrado
{
     Punto[] vertices;

     Cuadrado (Punto p1, Punto p2, Punto p3, Punto p4)
     {
          vertices = new Punto[4];
          vercices[0] = p1;
          vercices[1] = p2;
          vercices[2] = p3;
          vercices[3] = p4;
     }
     Punto getOrigen()
     {
          return vertices[0];
     }
     public static void main(String[] args)
     {
          Punto p1 = new Punto(0,0);
          Punto p2 = new Punto(2,0);
          Punto p3 = new Punto(0,2);
          Punto p4 = new Punto(2,2);
          Cuadrado c = new Cuadrado (p1, p2, p3, p4);
          System.out.println(c.getOrigen().getX());
```

```
            System.out.println(c.getOrigen().getY());
        }
    }
```

2.16.-EJERCICIOS PROPUESTOS

Ejercicio 1:

Escribir un programa que coja todos los parámetros de la línea de comandos, si los hay, y los muestre por pantalla, uno en cada línea, indicando con un solo asterisco después de cada no si está repetido (es decir, si algún otro parámetro previo es un string igual)

Ejercicio 2:

¿Por qué el siguiente trozo de código produce un error de compilación? ¿Cuál es la diferencia entre *a* y *c*?

```
class Inicializacion
{
    static int a;
    public static void main(String[] args)
    {
        int c;
        int b=17;
        if ( b!= 0)
            System.out.println(b);
        else
        {
            System.out.println(a);
            System.out.println(c);
        }
    }
}
```

Ejercicio 3:

Solucionar los problemas de la siguiente clase, reescribiéndola para que se compile correctamente.

```
class HolaMundo2
{
int cont;

public static void main( String args[] )
{
    cont = 0;
    HolaMundo2 a = new HolaMundo2();
    HolaMundo2 b = new HolaMundo2();
    HolaMundo2 c = new HolaMundo2();
    a.saludar();
    b.saludar();
    c.saludar();
}
```

```
HolaMundo2()
{
     cont++;
}
void saludar()
{
     System.out.println("¡Hola, mundo! Soy la instancia " +
cont );
}
}
```

Modificarla para que cada instancia indique su número secuencial. No se puede cambiar el método main (excepto la primera sentencia, aunque no es necesario).

Ejercicio 4:

Crear una clase Vaca que tenga:

- **Atributos:** color de pelo (String), edad (entero), nombre (String)
- Métodos:
  - debes elegir tres constructores diferentes
  - **Muu:** método que hace que la vaca muja y diga su nombre y color de pelo *"Muuu...mi nombre es XXX y mi color de pelo es YYY"*
  - **CompararEdad:** dadas las edades de dos vacas (la que envía el mensaje y otra como parámetro) saca el nombre de la vaca más antigua *"La vaca XXX es más vieja que la vaca YYY"*
- Programa principal:
  - Crear tres instancias de la clase Vaca, llamadas miVaca1, miVaca2 y miVaca3. Para cada una de ellas utilizar un constructor diferente.
  - Hacer que las tres mujan.
  - Comparar la edad de miVaca1 y miVaca2.
- **Modificar el programa** para que cada vez que se cree una instancia de la clase vaca salga un mensaje indicando el número de vaca creada: *"Se ha creado la vaca 1"*, etc.

Ejercicio 5:

Modificar la clase HolaMundo, llamándola HolaMundoAMedias: (Hacerlo utilizando los métodos más apropiados de String)

- para que se parta de una variable String que contenga " ¡Esta es la cadena Hola Mundo... a medias! ".
- posteriormente debe eliminar los espacios del inicio y del final.
- finalmente, quitar el substring que sobra para dejar simplemente "¡Hola Mundo... a medias!", que debe ser el String visualizado.
- Ayudas para la realización del ejercicio:

- El método de instancia trim() de la clase String elimina espacios en blanco de los extremos de un String.
- El método de instancia subString (inicio, fin) de la clase String devuelve los caracteres de inicio a fin del String.
- El método de instancia indexOf(subString) de la clase String indica en qué posición se encuentra la primera ocurrencia del subString en el String que llama al método.

Ejercicio 6:

Tras la realización de los ejercicios y prácticas, habrás podido ver qué pasos son necesarios para la edición y ejecución de un programa Java.

- Describe con tus palabras todos los pasos desde la edición hasta la ejecución, los problemas que pueden surgir y cómo solucionarlos.
- ¿Qué ficheros y programas son necesarios en cada etapa?
- ¿Cuál es el conjunto mínimo de ficheros y programas necesarios para la ejecución final de un programa Java?