



## Interacción de objetos. La estructura de control iterativa.

*Esta unidad de trabajo nos enseñará cómo construir pequeñas aplicaciones formadas por un conjunto de objetos de diferentes clases que cooperan e interactúan entre ellos. Los objetos se envían mensajes entre ellos para conseguir un objetivo común.*

*Aprenderemos además una nueva estructura de control de la programación, la iteración. Con esta estructura podremos diseñar métodos que realicen tareas repetitivas.*

*Por último, veremos qué es lo que debe incluir una aplicación Java para que pueda ejecutarse fuera del entorno BlueJ, tal como lo hacen las aplicaciones profesionales y empezaremos a utilizar algunas clases de la librería Java para dotar de mayor funcionalidad a nuestros programas.*

### 4.1.- Abstracción y modularización

Cuando un problema es simple, por ej, el proyecto MaquinaExpendedora del tema anterior, es posible encontrar una solución utilizando una única clase (la clase MaquinaExpendedora en ese caso). Sin embargo, a medida que crece la complejidad crece la dificultad para mantener la pista de todos los detalles del problema y dar una solución.

Para manejar la complejidad de un problema hay que utilizar:

- a) la **abstracción** - la abstracción es la habilidad para ignorar los detalles del problema centrando la atención en un nivel más alto del mismo.
- b) la **modularización** – se trata de dividir un problema en subproblemas, éstos a su vez en problemas más pequeños y dar una solución a cada uno de ellos individualmente. Es la técnica del “*divide y vencerás*”. Cada una de estas partes se construye y experimenta separadamente e interactúa con las demás a través de mecanismos bien definidos. Además cada uno de los subproblemas puede ser resuelto por personas diferentes.

En una compañía de coches, para diseñar un coche los ingenieros utilizan también la abstracción y la modularización. Ellos imaginan el coche como un conjunto de componentes. Cada ingeniero se centra en cómo construir uno de esos componentes (motor, ruedas, asiento, ...). Cuando un componente se construye se utiliza la abstracción, cada componente se considera una pieza más a ser utilizada para construir otros componentes más complejos.

Los mismos principios de abstracción y modularización se utilizan en el desarrollo del software. A la hora de diseñar un programa se trata de identificar subproblemas que se puedan programar como entidades separadas.

En la POO estos subcomponentes son objetos. Si intentásemos construir un coche a través de software, en lugar de implementar el coche como un único objeto, lo diseñaríamos como un conjunto de objetos separados, un objeto para el motor, otros para las ruedas, otros para los asientos, y después construiríamos un objeto coche a partir de estos objetos, ensamblándolos.

### 4.2.- Las clases como tipos

Para explicar los conceptos que surjan utilizaremos el proyecto Reloj Digital que implementa un visor (un *display*) para un reloj digital de estilo europeo: un reloj que va desde las 00:00 hasta las 23:59 horas (11:59 p.m.).

11:03

Si utilizamos la abstracción podemos considerar el display del reloj como dos paneles, uno para representar la hora (de 00 a 23) y otro para representar los minutos (de 00 a 59). Ambos tienen las mismas características, visualizan un valor que va de 0 hasta un límite determinado y se incrementan de uno en uno. Cuando alcanzan el límite se ponen otra vez a 0.

El código completo del proyecto es el siguiente:

```
/**
 * La clase VisorReloj implementa un panel de un reloj digital para un reloj
 * estilo 24 horas europeo. El reloj muestra horas y minutos. El rango del reloj es
 * 00:00 (medianoche) a 23:59 (un minuto antes de medianoche)
 *
 * El reloj recibe "ticks" (vía el método emitirTic()) cada minuto y reacciona
 * incrementando el visor. Esto se hace de la manera habitual: la hora se incrementa
 * cuando los minutos alcanzan de nuevo el valor cero.
 */
public class VisorReloj
{
    private VisorNumero horas;
    private VisorNumero minutos;
    private String visorString;    // simula el visor actual

    /**
     * Constructor de objetos VisorReloj. Este constructor crea un nuevo
     * reloj puesto en hora a las 00:00
     */
    public VisorReloj()
    {
        horas = new VisorNumero(24);
        minutos = new VisorNumero(60);
        actualizarReloj();
    }

    /**
     * Constructor de objetos VisorReloj. Este constructor crea un nuevo
     * reloj puesto en hora a partir de los parámetros pasados
     * al constructor
     */
    public VisorReloj(int hora, int minuto)
    {
        horas = new VisorNumero(24);
        minutos = new VisorNumero(60);
        ponerEnHora(hora, minuto);
    }

    /**
     * Este método debería llamarse una vez cada minuto -
     * permite que el reloj avance un minuto
     */
    public void emitirTic()
    {
        minutos.incrementar();
        if (minutos.getValor() == 0)
        {
            // se da la vuelta
            horas.incrementar();
        }
        actualizarReloj();
    }
}
```

```

/**
 * Pone la hora en el visor a un determinado valor de hora y minuto
 */
public void ponerEnHora(int hora, int minuto)
{
    horas.setValor(hora);
    minutos.setValor(minuto);
    actualizarReloj();
}

/**
 * Devuelve la hora actual en el formato HH:MM.
 */
public String getHora()
{
    return visorString;
}

/**
 * Actualiza la cadena interna que representa el visor
 */
private void actualizarReloj()
{
    visorString = horas.getValorVisor() + ":" +
        minutos.getValorVisor();
}
}

/**
 * La clase VisorNumero representa un visor - panel - digital que permite almacenar
 * valores entre 0 y un límite. El límite puede ser especificado cuando se crea
 * el visor. El rango de valores va de 0 a límite -1. Si se usa, por ejemplo, para
 * los segundos un reloj digital, el límite sería 60, resultando en este caso que
 * el visor mostraría valores entre 0 y 59. Al incrementarse, el visor da la vuelta
 * automáticamente y se pone a 0 al alcanzar el límite.
 */
public class VisorNumero
{
    private int limite;
    private int valor;

    /**
     * Constructor para objetos de la clase VisorNumero
     */
    public VisorNumero(int limiteMaximo)
    {
        limite = limiteMaximo;
        valor = 0;
    }

    /**
     * Devuelve el valor actual
     */
    public int getValor()
    {
        return valor;
    }
}

```

```

/**
 * Devuelve el valor del visor como una cadena de dos dígitos
 * Si el valor es menor que 10 se le añade a la izquierda un 0
 */
public String getValorVisor()
{
    if (valor < 10)
    {
        return "0" + valor;
    }
    else
    {
        return "" + valor ;
    }
}

/**
 * Pone el valor del visor al nuevo valor especificado. Si el
 * nuevo valor es menor que 0 o supera el límite no hace nada
 */
public void setValor(int nuevoValor)
{
    if ((nuevoValor >= 0) && (nuevoValor < limite))
    {
        valor = nuevoValor;
    }
}

/**
 * Incrementa el valor del display en 1, dando la vuelta a cero
 * si se alcanza el límite
 */
public void incrementar()
{
    valor = (valor + 1) % limite;
}
}

```

Nuestro reloj puede representarse a través de la clase VisorReloj:

```

public class VisorReloj
{
    private VisorNumero horas;
    private VisorNumero minutos;
    .....
}

```

donde VisorNumero es una clase tal que:

```

public class VisorNumero
{
    private int limite; // cuando se alcanza este límite el valor se pone a 0
    private int valor; // valor actual
    .....
}

```

En la sentencia, *private VisorNumero horas;* vemos que el atributo *horas* tiene como tipo la clase *VisorNumero*. Una clase puede ser utilizada para definir el tipo de una variable. Cuando ocurre esto la variable puede almacenar objetos (*referencias a objetos*) de esa clase.

### 4.3.- Diagramas de clases y diagramas de objetos

La estructura de la clase VisorReloj que utiliza otra clase, la clase VisorNumero, puede ser descrita utilizando un **diagrama de clases**. Este diagrama muestra las clases de la aplicación y las relaciones entre ellas. Muestra una *vista estática* del programa. Es lo que nos interesa cuando escribimos el código.

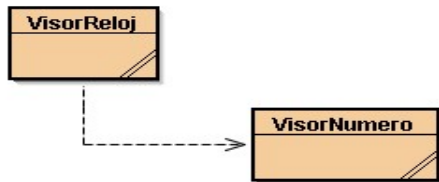


Diagrama de clases – La clase VisorReloj utiliza (depende) de la clase VisorNumero



Relación de **composición** en UML (un objeto VisorReloj se compone de 2 objetos VisorNumero)

El **diagrama de objetos** muestra los objetos y sus relaciones en un determinado momento de la ejecución del programa. Proporciona información de los objetos en tiempo de ejecución. Presenta una *vista dinámica* del programa.

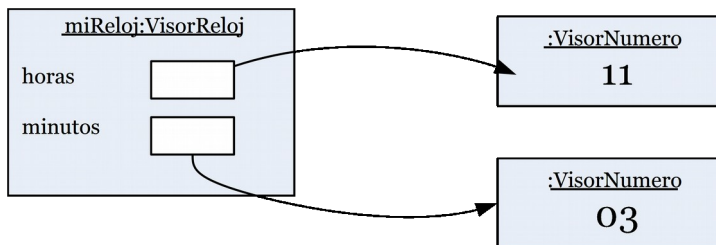


Diagrama de objetos – en nuestro ejemplo, al iniciar el programa se creará un objeto de la clase VisorReloj y este objeto (su constructor) creará en tiempo de ejecución dos objetos de la clase VisorNumero (horas y minutos)

BlueJ visualiza únicamente la vista estática, el diagrama de clases, además es un diagrama muy simplificado (no se refleja toda la variedad de relaciones entre clases que puede haber y que se pueden representar a través de UML).

**Ejer.4.1.** Dadas las siguientes clases:

```
public class Cuenta
{
    private String identificador;
    private Persona titular;
    private double balance;
    .....
}
```

```
public class Persona
{
    private String nombre;
    .....
}
```

- Dibuja el diagrama de clases
- Dibuja el diagrama de objetos en tiempo de ejecución para la cuenta *unaCuenta* con identificador = “14345BA” cuyo propietario es “Ana López” y el balance actual es 1800€.

**Ejer.4.2.** Haz lo mismo con las siguientes clases:

```
public class Circulo
{
    private Punto centro;
    private double radio;
    .....
}
```

```
public class Punto
{
    private int x;
    private int y;
    .....
}
```

- Dibuja el diagrama de clases
- Dibuja el diagrama de objetos en tiempo de ejecución para el círculo *miCirculo* de radio 5.4 y centro las coordenadas (13, 19) .

## 4.4.- Tipos primitivos y tipos objeto (tipos referencia)

Ya sabemos que las clases pueden ser usadas como tipos y, por tanto, podemos definir una variable de una determinada clase. A estos tipos se les denomina **tipos objeto** (o **tipos referencia**).

```
private VisorNumero horas;   private Bola unaBola;
private Persona titular;     private Profesor tutor;
```

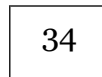
Java posee:

- tipos primitivos* – byte, short, int, float, double, long, char, boolean. Son los tipos predefinidos por el lenguaje.
- tipos objeto o tipos referencia* – aquellos definidos por las clases. Algunas clases son clases predefinidas de Java, como la clase String, otras las define el usuario.

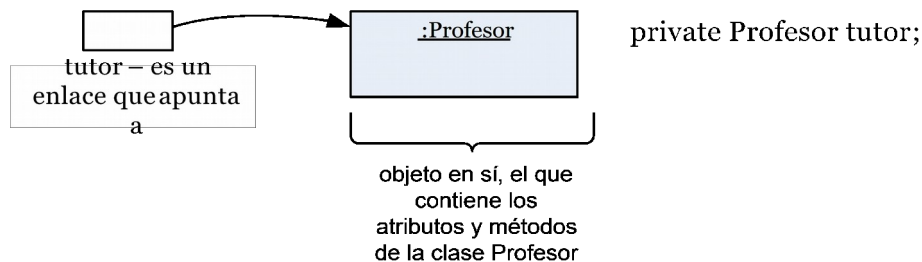
Ambos tipos pueden utilizarse para indicar el tipo de una variable pero hay diferencias entre ellos:

- las variables de un tipo primitivo almacenan directamente el valor

```
int numero = 34;
```



- las variables de un tipo objeto (aquellas cuyo tipo es una clase) almacenan una *referencia* al objeto, no el objeto en sí.



### Ejer.4.3.

- Abre el proyecto Reloj Digital
- Crea un objeto VisorReloj y prueba su comportamiento
- Analiza el código de la clase VisorNumero. ¿Cuál es la función del argumento en su constructor?
- ¿Qué hace el método incrementar()? Analiza la sentencia  $valor = (valor + 1) \% limite;$  Expresa esta misma sentencia con un *if*
- Analiza el código del método public String getValorVisor(). ¿Es un accesor o un mutador?
- ¿Qué ocurriría si el código de ese método fuese el siguiente? Haz los cambios y pruébalo. Luego restablece el método a su código inicial.

```
if (valor < 10) {
    return "0" + valor;
}
else {
    return valor ;
}
```

- Testea la clase `VisorNumero` creando varios objetos y llamando a sus métodos

## 4.5.- Creación de nuevos objetos

Centrémonos en la clase `VisorReloj`, en concreto en el primero de sus constructores:

```
public class VisorReloj
{
    private VisorNumero horas;
    private VisorNumero minutos;
    private String visorString;    // simula el visor actual

    /**
     * Constructor de objetos VisorReloj. Este constructor crea un nuevo
     * reloj puesto en hora a las 00:00
     */
    public VisorReloj()
    {
        horas = new VisorNumero(24);
        minutos = new VisorNumero(60);
        actualizarReloj();
    }
    .....
}
```

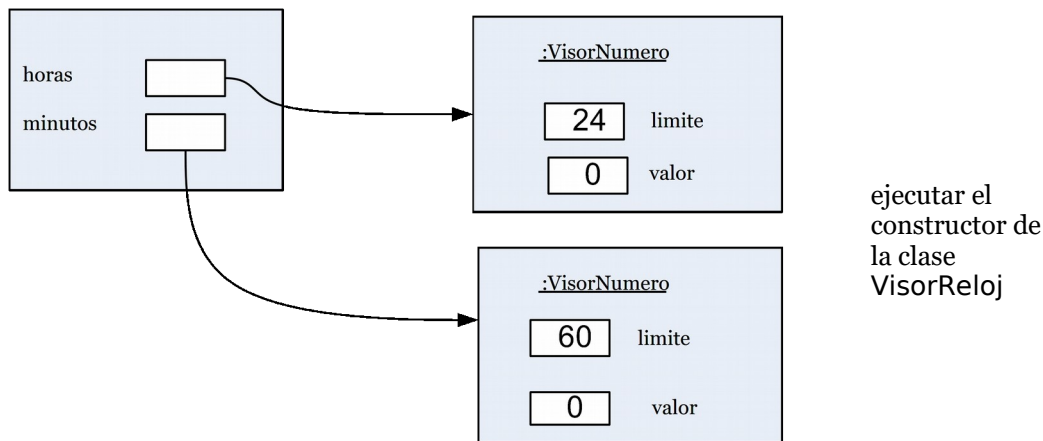
La clase define dos atributos, *horas* y *minutos*, ambos tienen como tipo una clase, la clase `VisorNumero`. Estos atributos almacenarán una referencia a un objeto de esa clase.

Cuando un atributo (en general, una variable) de un tipo objeto se declara, el objeto todavía no existe. Por analogía, cuando una variable de tipo *int* se declara la variable no tiene ningún valor todavía. Hay que crear los nuevos objetos con el **operador new**. En nuestro ejemplo, cuando desde BlueJ se crea un objeto de la clase `VisorReloj` se ejecutará el constructor de esa clase. Dentro del constructor se crean los objetos *horas* y *minutos* de la clase `VisorNumero`.

```
horas = new VisorNumero(24);
minutos = new VisorNumero(60);
```

El operador **new** realiza dos cosas:

- crea un nuevo objeto de la clase especificada detrás de *new* y asigna una referencia a ese objeto a la variable especificada a la izquierda de la asignación
- ejecuta el constructor de esa clase (el código incluido en el constructor)



Ya que el constructor de la clase `VisorNumero` tiene un argumento,

`public VisorNumero(int limiteMaximo),` ← parámetro formal

al crear un objeto de esta clase habrá que pasar un parámetro al constructor,

`horas = new VisorNumero(24);` ← parámetro actual

**Ejer.4.4.** Sea la siguiente declaración:

```
public class Tren
{
    private int velocidadMaxima;
    private String nombre;

    /**
     * Constructor de objetos Tren.
     */
    public Tren(int v, String n)
    {
        velocidadMaxima = v;
        nombre = n;
    }
    .....
}
```

- ¿Qué hacemos cuando escribimos, `Tren unTren;`?
- ¿Es correcto, `unTren = new Tren()` ?
- Crea correctamente un objeto `Tren` de nombre “ave” y velocidad 250.
- Dibuja el diagrama de objetos para el apartado anterior

**Ejer.4.5.** Escribe la signatura de un constructor que se ajuste a la siguiente creación de un objeto:

`miEditor = new Editor("readme.txt", -1);`

## 4.6.-Múltiples constructores. Constructores sobrecargados

En Java es posible definir varios constructores en la misma clase. Todos tendrán el mismo nombre, el nombre de la clase. Ninguno devolverá un valor de retorno (ni siquiera `void`).

Lo que distinguirá a unos de otros será el nº y/o tipo de sus argumentos (no puede haber más de un constructor con igual nº de parámetros y mismo tipo).



```

public VisorReloj()
public VisorReloj(int hora, int minuto)

```

La razón para incluir varios constructores en una misma clase es proporcionar alternativas a la hora de construir e inicializar los objetos.

El hecho de definir con el mismo nombre varios constructores se denomina **sobrecarga** y de los constructores se dice que están *sobrecargados*. La sobrecarga se puede aplicar no solo a los constructores sino a cualquier método. Los argumentos que se pasan en la llamada determinarán qué constructor o método se ejecutará.

**Ejer.4.6.** Analiza los dos constructores de la clase VisorReloj examinando similitudes y diferencias. ¿Por qué no se llama a actualizarReloj() en el segundo constructor?

**Ejer.4.7.** Dada la siguiente clase:

```

public class Pasajero
{
    private String nombre;
    private int peso;
    private double altura;

    public Pasajero( String n)
    {
        nombre = n;
        peso = 78;
        altura = 1.80;
    }

    public Pasajero( String n, int p)
    {
        nombre = n;
        peso = p;
        altura = 1.80;
    }

    public Pasajero( String n, double a)
    {
        nombre = n;
        peso = 73;
        altura = a;
    }

    public Pasajero( String n, int p, double a)
    {
        nombre = n;
        peso = p;
        altura = a;
    }
}

```

- Indica qué sentencias son correctas:

```

Pasajero p = new Pasajero("pepe");
Pasajero p = new Pasajero("pepe", 1.83);
Pasajero p = new Pasajero("pepe", 65.8, 1.83);
Pasajero p = new Pasajero("pepe", 69, 1.83);

```

**Ejer.4.8.** Siguiendo con la definición anterior,

- ¿qué ocurre si hacemos?

```

Pasajero pasa1;
Pasajero pasa2;
pasa1 = new Pasajero("Jose Luis");
pasa1 = pasa2;

```

- Dibuja el diagrama de objetos.
- Declara y crea en la misma sentencia un pasajero *pasa3* de nombre “Alejandro” y altura 1.75. ¿Qué peso tendrá este pasajero?

## 4.7.-Llamadas a métodos

### 4.7.1.-Llamadas internas a métodos

Los métodos pueden llamar a otros métodos de la misma clase como parte de su implementación. Es lo que se denomina **llamada interna** a un método (desde dentro de la clase). Nuestra clase *VisorReloj* define un método, `private void actualizarReloj()`, al cual se le invoca desde dentro del constructor de la clase:

```
/**
 * Constructor de objetos VisorReloj. Este constructor crea un nuevo
 * reloj puesto en hora a las 00:00
 */
public VisorReloj()
{
    horas = new VisorNumero(24);
    minutos = new VisorNumero(60);
    actualizarReloj();
}
```

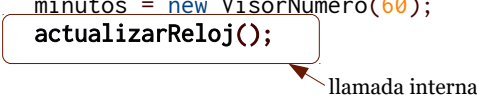


Diagram illustrating an internal call: A box around the `actualizarReloj();` line in the constructor is connected by an arrow to the text "llamada interna".

Para efectuar una llamada interna a un método de la misma clase (*enviarse un mensaje a sí mismo el objeto*) se especifica:

*nombre\_método(lista de parámetros);*

Cuando se invoca a un método interno se localiza en la clase un método con el mismo nombre y parámetros (nº y tipo) especificados en la llamada y se ejecuta el método. Cuando acaba la ejecución se retorna al punto donde se produjo la llamada y continúa la ejecución en la siguiente sentencia después de la llamada.

Otro ejemplo:

```
/**
 * Constructor de objetos VisorReloj. Este constructor crea un nuevo
 * reloj puesto en hora a partir de los parámetros pasados
 * al constructor
 */
public VisorReloj(int hora, int minuto)
{
    horas = new VisorNumero(24);
    minutos = new VisorNumero(60);
    ponerEnHora(hora, minuto);
}
```

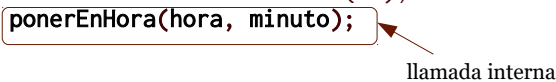


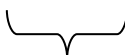
Diagram illustrating an internal call: A box around the `ponerEnHora(hora, minuto);` line in the constructor is connected by an arrow to the text "llamada interna".

### 4.7.2.-Llamadas externas a métodos

Los métodos pueden llamar a métodos de otros objetos (pueden enviar mensajes a otros objetos). *Enviar un mensaje* a un objeto *obj* significa querer ejecutar un método definido dentro de la clase a la que pertenece *obj*. Es lo que se denomina efectuar una **llamada externa** a un método.

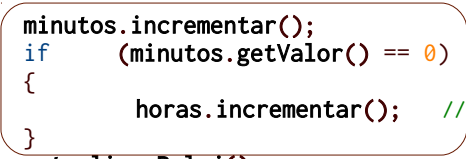
La sintaxis a utilizar para llamar a un método externo es la siguiente:

***objeto.nombre\_método(lista de parámetros);***

  
referencia al objeto

Otro ejemplo:

```
/**
 * Este método debería llamarse una vez cada minuto -
 * permite que el reloj avance un minuto
 */
public void emitirTic()
{
    minutos.incrementar();
    if (minutos.getValor() == 0)
    {
        horas.incrementar(); // se da la vuelta
    }
    actualizarReloj();
}
```

  
llamadas externas

## 4.8.- Visibilidad *public* / *private*

Cuando se define un método en una clase se especifica su **visibilidad** con las palabras reservadas *public/private*:

- a) **public** – significa que el método es público, por lo tanto, es visible desde fuera de la clase. Esto quiere decir que puede ser invocado desde otros objetos

`minutos.incrementar();`

- b) **private** – si un método se define con esta visibilidad significa que sólo podrá ser llamado desde dentro de la propia clase (sólo llamadas internas)

`actualizarReloj();`

La visibilidad se aplica también a los atributos. En los ejemplos utilizados hasta ahora todos los atributos definidos en las clases son *private*. Únicamente son accesibles desde dentro de la propia clase. Esto debe ser siempre así ya que garantiza la encapsulación.

**Ejer.4.9.** Dada la siguiente definición de atributo en una determinada clase:

*private Impresora impre;*

La clase *Impresora* tiene definidos dos métodos con las siguientes signatures:

```
public void imprimir(String nombreFichero, boolean dobleCara)
public int getEstado()
```

Supongamos que en algún punto se ha construido ya un objeto de la clase *Impresora* (*impre = new Impresora();*).

Realiza dos posibles llamadas a cada uno de los métodos anteriores.

**Ejer.4.10.** Dada la siguiente definición:

```
public class Rueda
{
    private double presion;

    public Rueda(double p)
    {
        presion = p;
    }

    public void inflar()
    {
        presion += 0.5;
    }

    public void desinflar()
    {
        presion -= 0.5;
    }

    public boolean estaDesinflada ()
    {
        return (presion == 0.0);
    }
}
```

Define una clase *Bicicleta* tal que:

- tenga 2 atributos, *delantera* y *trasera* que representan las 2 ruedas de una bici
- el constructor cree las ruedas con una presión inicial de 0.0
- incluya un método público *verificar()* sin parámetros y que no devuelve nada. Este método verifica cada una de las ruedas de la bici. Si están desinfladas las infla quedando las ruedas con una presión de 1.5.

## 4.9.- La palabra clave *this*. La palabra clave *null*

### 4.9.1.-Uso de *this*

Imaginemos la siguiente situación:

```
public class Rueda
{
    private double presion;

    public Rueda(double presion)
    {
        presion = presion;
    }

    .....
}
```

En principio Java no genera ningún error ante la siguiente situación. Estamos ante lo que se llama *sobrecarga de nombres*. Hay un atributo en la clase, *presion*, con el mismo nombre que el parámetro local en el constructor. Esto no genera errores, sin embargo, la asignación en el constructor no es correcta porque lo que queremos es dejar en el atributo *presion* el valor del parámetro *presion* que se pasa al constructor. Pero lo que ocurre es que se asigna al argumento local su propio valor. La forma de solucionar esto es haciendo uso de ***this*** para referirnos al objeto actual (el *objeto receptor* del método) .

```

public class Rueda
{
    private double presion;

    public Rueda( double presion)
    {
        this.presion = presion;
    }
    .....
}

```

asignar al atributo *presion* del objeto actual el valor del parámetro formal *presion*

**Ejer.4.11.** Dada la siguiente definición:

```

public class Artículo
{
    private String nombre;
    private double precio;

}

```

Completa la clase:

- escribe el constructor parametrizado que inicialice los dos atributos
- escribe los accesores para el nombre y el precio
- escribe el método, `public boolean masBaratoQue(Artículo a)`, que devuelve *true* si el precio del objeto actual es menor que el del objeto que se pasa como parámetro (no utilices *if*)

Otro uso de la palabra *this* es para pasar el objeto actual como parámetro a otro método.

**Ej.** `public class Coche`

```

{
    .....
    public void aparcar(Parking p)
    {
        p.aparcarCoche(this); // paso el objeto actual como parámetro
    }
    .....
}

public class Parking
{
    private Coche c;
    .....
    public void aparcarCoche(Coche c)
    {
        this.c = c;
    }
    .....
}

```

## 4.9.2.- La palabra clave *null*

Cuando se declara una variable de tipo objeto la variable no apunta a nada, a ningún objeto, por defecto, tiene como valor inicial ***null***. Si definimos, `private Artículo unArtículo;`, hasta que no se invoque al constructor (o se asigne con el valor de otro objeto *Artículo*) la variable *unArtículo* contiene *null*.

Cuando una variable objeto no apunta a nada no es posible llamar a ningún método de ese objeto ya que se generaría un error (una excepción `NullPointerException`). Para evitar esto se puede verificar si la variable es *null* o no.

Java tiene un mecanismo, el ***Garbage Collector***, para destruir aquellos objetos que no son referenciados por ninguna variable.

**Ej.**

```
public boolean masBaratoQue(Articulo a)
{
    if (a == null) { // este test permite verificar si la variable apunta o no a un objeto
        return false; // mejor lanzar una excepción
    }
    return (a.getPrecio() > this.precio);
}
```

**Ejer.4.12.** Siguiendo con la clase `Articulo`:

- añade un mutador para el precio
- representa gráficamente, con un diagrama de objetos, las siguientes situaciones,

```
Articulo uno = new Articulo("Tornillo", 2.67);
Articulo otro;
uno.setPrecio(3.4);
```

- ¿Es posible esta sentencia: `uno.setNombre("Tornillo - B2");` ?
- `otro = uno;`  
`uno.setPrecio(3.7);`  
¿Es posible esta sentencia: `otro.setPrecio(13.28);` ?  
`uno = null;`  
`otro = null;`

**Ejer.4.13.** Repasa el código completo del proyecto Reloj Digital. Vamos a modificarlo para que en lugar de ser un reloj de 24 horas sea un reloj de 12 horas y muestre la hora de la forma 4:23 a.m. o 4:23 p.m. Las 00:30 se mostrarán como 12:30. Los minutos se mostrarán de 0 a 59 pero las horas son ahora de 1 a 12. Modificaremos únicamente el método `actualizarReloj()`. El resto quedará igual.

## 4.10.- Usando Java fuera de BlueJ

Hasta ahora hemos utilizado BlueJ para desarrollar y ejecutar aplicaciones Java. BlueJ nos proporciona herramientas para realizar fácilmente las tareas de desarrollo y ejecución: editamos, compilamos, creamos objetos dentro de BlueJ, testeamos individualmente cada uno de sus métodos, podemos ejecutar la herramienta *Debug* para efectuar una ejecución paso a paso.

Lo habitual, sin embargo, será que tengamos que desarrollar y ejecutar nuestras aplicaciones fuera de este entorno.

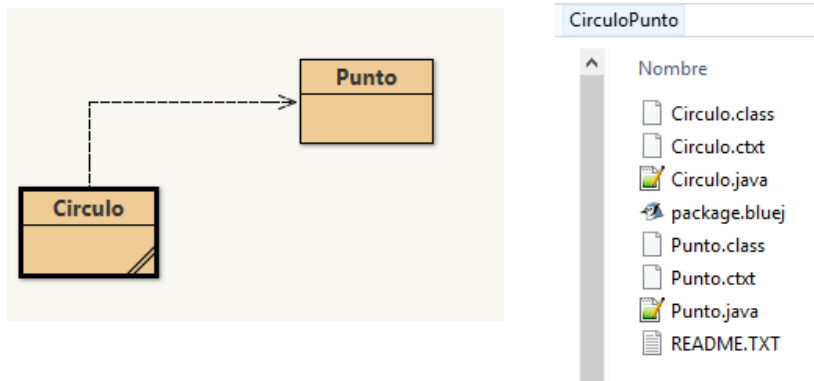
Veamos en primer lugar la estructura del directorio que crea BlueJ para un proyecto.

### 4.10.1.- Estructura de un proyecto BlueJ

Un proyecto BlueJ se almacena en un directorio del disco, por ej, `C:\.....\Ejercicios Java\Circulo y Punto`.

Un paquete BlueJ se almacena en varios ficheros. Algunos contienen código fuente, otros código compilado y otros información adicional. BlueJ utiliza el formato standard Java para algunos ficheros y añade otros ficheros con información adicional extra.

La estructura de ficheros que genera BlueJ para un proyecto es la siguiente:



<b>package.bluej</b>	Hay uno por paquete. Guarda detalles del proyecto.
<b>*.java</b>	Un fichero fuente java por cada clase del proyecto
<b>*.class</b>	Fichero de código standard java compilado. Hay uno por clase
<b>*.ctxt</b>	Fichero BlueJ con información adicional. Hay uno por clase

Recordemos que los ficheros *\*.java* son los ficheros fuente. Contienen el código fuente que escribe el programador. Hay uno por clase. Los ficheros *\*.class* contienen el código compilado, el que más tarde interpretará la máquina virtual. Hay uno por clase y los genera el compilador Java a partir del código fuente.

#### 4.10.2.- Desarrollando fuera de BlueJ

La edición y compilación del proyecto la hemos hecho desde BlueJ. Pero es posible efectuar estos procesos fuera de este entorno:

- la edición se puede hacer con cualquier editor (tipo *NotePad* o *WordPad*). Una vez editado el fichero se guarda con extensión *.java*.
- un fichero fuente *.java* puede compilarse desde la línea de comandos con el compilador java incluido en el JDK. El nombre del compilador es **javac** (hay que asegurarse de que la ruta al compilador está establecida en el PATH).

...>*javac Circulo.java* - este comando compila la clase *Circulo* y cualquier otra que dependa de ella. Crea un fichero *Circulo.class* con el código que ejecutará la JVM (máquina virtual)

- para ejecutar se llama al comando **java**

...>*java Circulo* - no es necesario indicar la extensión. Java inicia la máquina virtual, la clase se carga en memoria y comienza su ejecución. Si se necesitan otras clases se cargan también

### 4.10.3.- Ejecutando sin BlueJ

Cuando se llama a ejecución, `...>java Circulo` se genera un error ,

```
"Exception in thread main"  
java.lang.No Such Method Error: main
```

El problema es: ¿cómo sabe el sistema operativo qué método ejecutar de los que incluye la clase Circulo? ¿Dónde iniciar la ejecución?

El sistema Java siempre inicia la ejecución de una aplicación buscando un método llamado **main()**. Este método debe existir y debe tener la siguiente signatura:

```
public static void main(String[] args)
```

El método *main()*:

- debe existir en alguna clase del proyecto
  - ha de ser público (para que pueda ser invocado desde fuera de la clase)
  - debe ser *static* (método de clase – quiere decir que no se necesitan instancias de la clase que contiene a *main()* para invocarlo)
  - debe tener un array de parámetros *String* (esto permitirá pasar argumentos a la aplicación en el momento de su inicio)
  - sólo se puede invocar a *main()* para ejecutar una aplicación
- **¿Dónde incluir el main()? -** Normalmente este método incluye sentencias que inician la aplicación (crear objetos, llamar a algún método que inicie la aplicación, ....). Puede incluir cualquier sentencia.

De momento para cada proyecto incluiremos una clase adicional, en nuestro ejemplo, la clase *TestCirculo* (o *PruebaCirculo*) en la que incluiremos el método *main()*. Dentro de este método incluiremos las sentencias que nos permiten probar (testear) la aplicación. Desde BlueJ llamaremos al método *main()* sin crear ningún objeto de la clase *TestCirculo*.

```
public class TestCirculo  
{  
    public static void main(String[] args)  
    {  
        Punto unPunto = new Punto(3,6);  
        Circulo miCirculo = new Circulo(unPunto, 6);  
        double x = unPunto.getX();  
        double y = unPunto.getY();  
        double area = miCirculo.calcularArea();  
        System.out.println("x=" + x + " y= " + y +  
            "\nArea =" + area + "\n");  
    }  
}
```

Más adelante cuando hagamos aplicaciones completas incluiremos el *main()* en una única clase y dentro de él incluiremos las sentencias necesarias para iniciar la aplicación.



## 4.11.- Añadiendo comportamiento iterativo: la estructura de control iterativa

Los métodos representan el comportamiento que tienen los objetos de una clase. Un método consta de una serie de sentencias que se ejecutan en orden secuencial, unas detrás de otras.

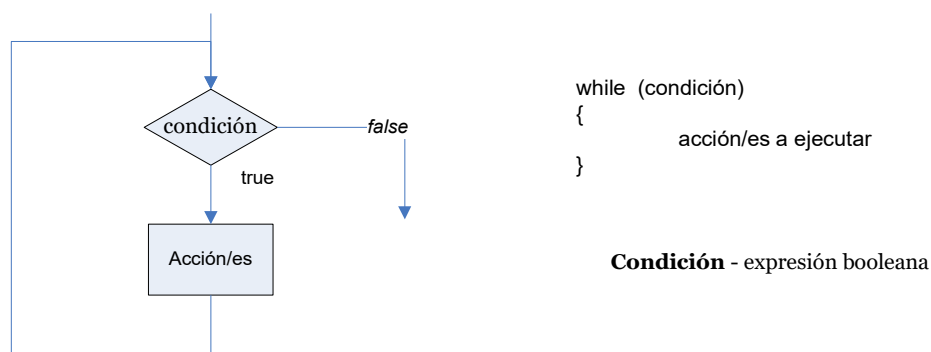
Hasta ahora las sentencias que hemos utilizado han sido:

- sentencias para escribir valores en la *Terminal de texto* (*System.out.println()*)
- sentencia de asignación ( *variable = valor* )
- llamadas a métodos (*minutos.incrementar()*)
- la sentencia *if* (o *switch*) para ejecutar unas u otras instrucciones dependiendo del valor de una condición

Muchas veces en programación es necesario repetir una o varias instrucciones un n° determinado o indeterminado de veces. Los lenguajes incluyen estructuras de control para expresar las iteraciones o bucles.

Java proporciona tres instrucciones de control repetitivas: *while*, *for*, *do ... while*.

### 4.11.1.- La sentencia *while*



- la condición (expresión booleana) se evalúa. Si es cierta se ejecutan las instrucciones incluidas en el cuerpo del *while* y se vuelve a evaluar la condición. Este proceso se repite hasta que la condición es falsa. En este caso el bucle termina.
- la condición se evalúa siempre en primer lugar. Esto quiere decir que el bucle puede que no se ejecute nunca si la condición se evalúa a *false* la primera vez.
- el conjunto de acciones (sentencias) se repiten un n° determinado o indeterminado de veces.
- siempre tiene que haber alguna sentencia dentro del cuerpo del *while* que modifique la condición para que el bucle pueda terminar ( y no se generen bucles infinitos).

**Ej.**

Analizaremos el código mostrado en los siguientes ejemplos.

Realizaremos en cada ejemplo la **traza del algoritmo**.

Hacer la **traza** de un algoritmo es seguir la ejecución de ese algoritmo paso a paso para valores concretos de las variables que intervienen en él.

Una traza permita comprobar si un algoritmo es incorrecto (no si es correcto, ya que para ello habría que hacer la traza con multitud de valores) .

```
int veces = 1;
while (veces <= 10)
{
    System.out.println("Saludo " + veces);
    veces = veces + 1;
}
```

```
int tiradas = 1;
while (tiradas <= 10)
{
    moneda.tirar();
    tiradas++;
}

int caras = 0; //es una variable contador
int cruz = 0; //es una variable contador
int tiradas = 1; //es una variable contador
while (tiradas <= 30)
{
    moneda.tirar();
    if (moneda.esCara()) {
        caras++;
    }
    else {
        cruz++;
    }
    tiradas++;
}
```

```
boolean salioCara = false; // es una variable que actúa como un switch o conmutador
int tiradas = 1; //es una variable contador
while (tiradas <= 30 && ! salioCara)
{
    moneda.tirar();
    if (moneda.esCara()) {
        salioCara = true;
    }
    else {
        tiradas++;
    }
}
System.out.println("Salió cara en la tirada " + tiradas);
```

```
int suma = 0; //es una variable acumulador
int producto = 1; //es una variable acumulador
int contador = 20; //es una variable contador
while (contador >= 1)
{
    suma = suma + contador;
    producto = producto * contador;
    contador = contador - 1;
}
```

Un **contador** es una variable cuyo valor se incrementa en una cantidad fija, positiva o negativa. Se utiliza en los siguientes casos:

- a) Para contabilizar el número de veces que es necesario realizar una acción (variable de control de un bucle). **Ej.** tiradas++; contador--;

- b) Para contar un suceso particular (asociado o no a un bucle). **Ej.** `caras++;` `cruz++;`

Un **acumulador** es una variable cuyo valor se incrementa sucesivas veces en cantidades variables. Se utiliza:

- a) Para obtener un total acumulado de un conjunto de cantidades siendo preciso, en este caso inicializar el acumulador a 0. **Ej.** `suma = suma + contador;`
- b) Para obtener un total como producto de distintas cantidades, inicializándose entonces a 1. **Ej.** `producto = producto * contador;`

Un **switch** o conmutador es una variable que toma dos valores exclusivos, 0 / 1, true / false, -1 / 1, y que se utiliza para:

- a) hacer que se ejecuten alternativamente dos grupos de sentencias dentro de un bucle
- b) recordar la ocurrencia o no de un determinado suceso o para salir de un bucle

**Ej.**

```
/* Suma de pares entre 1 y 100
y producto de impares entre 1 y 100 */
int sumaPar = 0; // es un acumulador
int productoImpar = 1; //es un acumulador
boolean tocaImpar = true; // es un switch
int numero = 1; //es un contador
while (numero <= 100)
{
    if (tocaImpar)
    {
        productoImpar *= numero;
        tocaImpar = false;
    }
    else
    {
        sumaPar += numero;
        tocaImpar = true;
    }
    numero++;
}
```

**Ejer.4.14.** Codifica en Java los siguientes métodos. Realiza también la traza.

- a) `public void escribirNumero(int desde, int hasta)` – escribe los números que están comprendidos entre el parámetro *desde* y el parámetro *hasta*. Si *desde* = 10 y *hasta* =30, escribe 10, 11, 12, .....,30
- b) `public void mostrarPares()` - Visualiza en la ventana de terminal los números pares entre 2 y 50.
- c) `public int generarAleatorios()` - genera y escribe  $n^{\text{os}}$  aleatorios entre 1 y 100 parando el proceso cuando se generan 30 o bien cuando salga un 99. El método cuenta también las veces que sale el 12 y devuelve este valor.

Para generar un  $n^{\text{o}}$  aleatorio entre 1 y 100: **`numero = (int) (Math * random() * 100) + 1;`**

- d) `public double sumarSerie(int n)` - Calcula y devuelve la suma de la serie  

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n}$$
- e) `public double sumarSerie(int n)` - Calcula y devuelve la suma de la serie  

$$\frac{1}{1} - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \dots \pm \frac{1}{n}$$
- f) `public int sumarDivisores(int numero)` - Calcula y devuelve la suma de los divisores de `numero`
- g) `public int sumarDigitos(int numero)` - Calcula y devuelve la suma de los dígitos de `numero`
- h) `public boolean esPrimo(int numero)` - devuelve *true* si *numero* es primo, *false* en otro caso, Un *nº* es primo si únicamente tiene como divisores él mismo y la unidad

**Ejer.4.15.** (Para trabajar con *bucles anidados*) Escribe los siguientes métodos:

- a) `public void escribirFigura(int n)` - escribe la siguiente figura, si *n* = 6 (por ej.)

```

1 1 1 1 1 1
2 2 2 2 2 2
3 3 3 3 3 3
4 4 4 4 4 4
5 5 5 5 5 5
6 6 6 6 6 6

```

- b) `public void escribirFigura(int n)` - escribe la siguiente figura, si *n* = 6 (por ej.)

```

1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
6 6 6 6 6 6

```

- c) `public void escribirFigura(int n)` - escribe la siguiente figura, si *n* = 6 (por ej.)

```

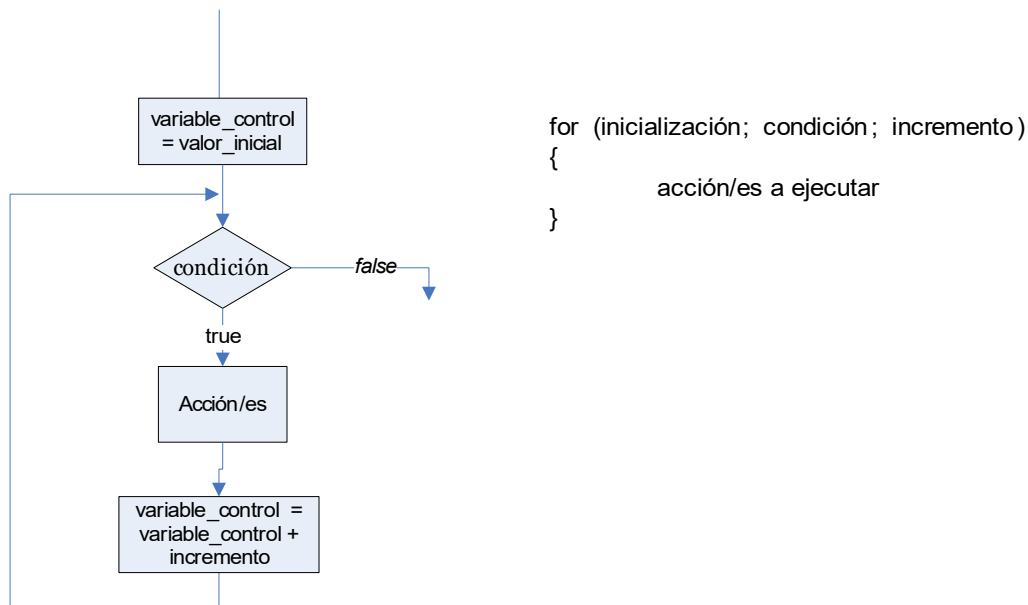
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4 5 6

```

- d) `public void escribirTablasMultiplicar(int numero)` - escribe la tabla de multiplicar de 1 1, del 2, del 3, .... hasta la tabla de *numero*
- e) `public void calcularSumatorios(int cuantos, int limite)` - genera *cuantos* *nºs* aleatorios entre 1 y *limite* y por cada *nº* generado calcula su sumatorio

### 4.11.2.- La sentencia *for*

Esta sentencia permite ejecutar un n<sup>o</sup> determinado de veces una o varias sentencias.



- *inicialización* – establece la situación inicial para empezar a iterar. Generalmente consiste en asignar un valor inicial a la variable de control del bucle (la que lleva la cuenta del n<sup>o</sup> de veces que se han de repetir las sentencias)
- *condición* – si se evalúa a *true* se ejecuta el bucle, si es *false* termina
- *incremento (o decremento)* – incrementa (decrementa) la variable de control para continuar la iteración

**Ej.**

```
for (int veces = 1; veces <= 10; veces++)
{
    System.out.println("Saludo " + veces);
}
```

```
for (int tiradas = 1; tiradas <= 30; tiradas++)
{
    moneda.tirar();
}
```

```
int ncaras = 0;
int ncruz = 0;
for (int tiradas = 1; tiradas <= 30; tiradas++)
{
    moneda.tirar();
    if (moneda.esCara()) {
        ncaras++;
    }
    else {
        ncruz++;
    }
}
```

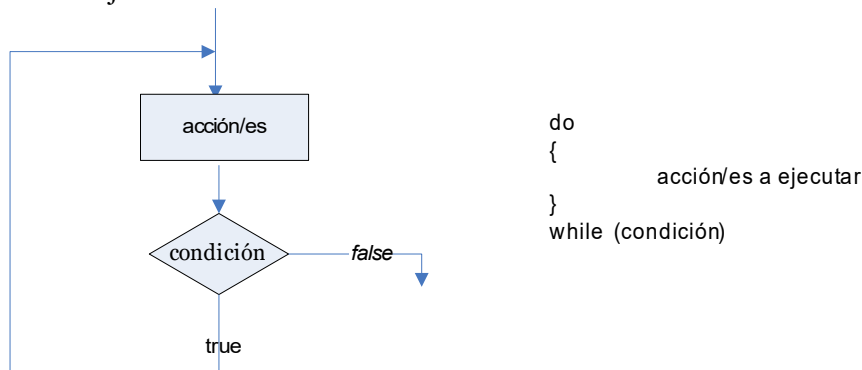
```
}  
}
```

**Ejer.4.16.** Escribe los siguientes métodos utilizando para las repeticiones la sentencia *for*:

- a) `public void contarParesImpares()` - genera 20 n<sup>os</sup> aleatorios comprendidos entre 1 y 50 y cuenta los pares e impares generados escribiendo al final esta cuenta
- b) `public int maximo(int cuantos)` - calcula y devuelve el máximo de una secuencia de valores (tantos como indique *cuantos*) aleatorios comprendidos entre 1 y 100
- c) `public void escribirEstadisticas()` - genera TOTAL notas aleatorias (comprendidas entre 1 y 10) y escribe la siguiente estadística: media de las notas, nota máxima, nota mínima, cuántas veces ha aparecido la nota máxima y cuántas la nota mínima. TOTAL es una constante de valor 30.

### 4.11.3.- La sentencia *do .. while*

El bucle *do .. while* es una variación del bucle *while* con la diferencia de que las sentencias del cuerpo del bucle se ejecutan al menos una vez.



**Ej.**

```
int veces = 1;  
do  
{  
    System.out.println("Saludo " + veces);  
    veces++;  
}  
while (veces <= 10);
```

## 4.12.- La clase *Random*. La clase *Scanner*

Aunque en temas posteriores veremos en profundidad la API (librerías) de Java, vamos a introducir ahora las clases *Random* y *Scanner*.

### 4.12.1.- La clase *Random*

Esta clase es una de las contenidas en la librería de clases de Java. Permite generar n<sup>os</sup> aleatorios. Para ello:

- a) hay que crear una instancia de la clase *Random*

- b) hay que llamar a un método de esa instancia para obtener el n° aleatorio, por ej, **nextInt()**
- c) para poder utilizar la clase hay que importarla incluyendo la sentencia **import java.util.Random** al comienzo de la declaración de la clase que utilizará a Random.

Ej.

```
.....
Random generador = new Random();
int valor = generador.nextInt(10) + 1;
```

genera un n° aleatorio entre 0 y 9

**Ejer.4.17.** Completa la siguiente clase que modela un dado de 6 caras:

```
import java.util.Random;
public class Dado
{
    private int cara;
    private Random generador;

    public Dado( )
    {
        generador = new Random();
        cara = 1;
    }

    public int getCara( )
    {
        return cara;
    }

    public void tirarDado( )
    {
        // a completar
    }
}
```

### 4.12.2.- La clase *Scanner*

La clase *Scanner* es una clase de la librería Java incorporada a partir de la versión 1.5. Permite leer valores desde el dispositivo de entrada standard, el teclado. Para ello:

- a) se crea una instancia de la clase *Scanner*

```
Scanner teclado = new Scanner(System.in);
```

*System.in* se refiere a la entrada standard (*System.out* es la salida standard)

- b) se llama a un método de la clase para leer un entero, o un real, o un String, ...

- **nextInt()** – lee un entero del teclado
- **nextDouble()** – lee un real
- **next()** – lee un string
- **nextBoolean()** – lee un valor booleano
- **nextLine()** – lee un string completo hasta el fin de línea incluyendo los separadores (espacios, tabuladores, ....). (Lee los *tokens* que quedan hasta el fin de línea).

**Ej.** El siguiente ejemplo muestra una clase prueba que testea una clase `Persona`. En esta clase se utiliza la clase `Scanner` para leer del teclado los valores que se pasan como parámetros actuales a los métodos de la clase `Persona` (lo que en BlueJ hacemos desde el propio entorno):

```
import java.util.Scanner;
public class TestPersona
{
    public static void main(String[] args)
    {
        Persona p = new Persona();
        Scanner teclado = new Scanner(System.in);
        String nombre;
        System.out.print(" Nombre de la persona =");
        nombre = teclado.nextLine();
        p.setNombre(nombre);
        System.out.print(" Edad de la persona =");
        p.setEdad(teclado.nextInt());
        p.printDatosPersona();
    }
}
```

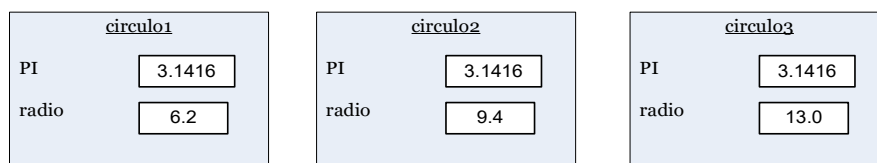
## 4.13.- Miembros de clase (miembros *static*)

Los miembros de las clases definidas hasta el momento han sido *miembros de instancia*, los atributos (variables de instancias), las constantes y los métodos (métodos de instancia), miembros que se asocian a los objetos individuales, a las instancias de la clase. Cada objeto tiene sus propios valores para sus atributos (constantes y variables) y los métodos se invocan sobre los objetos.

**Ej.** `public class Circulo`

```
{
    private final double PI = 3.1416;
    private double radio;
    .....

    public double getPerimetro()
    {
        return 2 * PI * radio;
    }
    .....
}
```



Las clases pueden contener también *miembros de clase*, tanto constantes, como variables y métodos de clase. Se denominan miembros **static** y pertenecen a la clase, no a las instancias (los objetos).

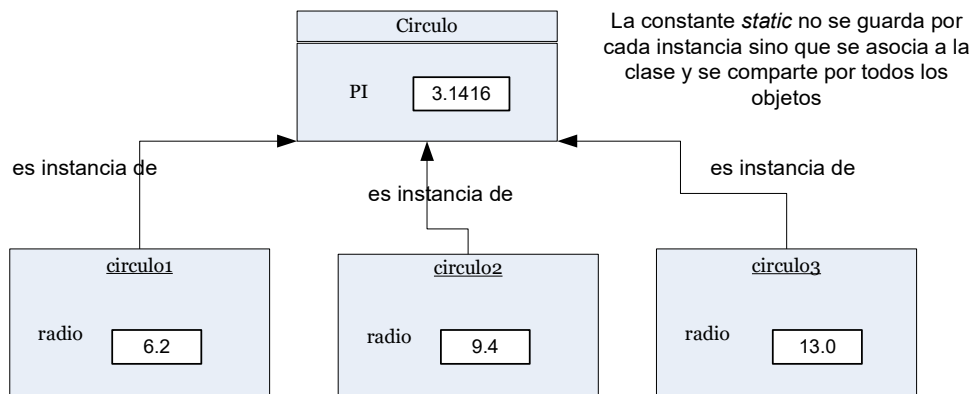
### 4.13.1.- Constantes de clase (constantes *static*)

Se declaran como las constantes habituales pero anteponiendo la palabra reservada *static*.

```
private static final double PI = 3.1416; // constante de clase con un valor inicial no modificable
```



Las constantes *static* no pertenecen a los objetos sino a la clase.



```
public class Circulo
{
    private static final double PI = 3.1416;
    private double radio;
    .....

    public double getPerimetro()
    {
        return 2 * PI * radio;
    }
    .....
}
```

Es habitual el uso de constantes *static* y además es muy común definir las con visibilidad *public* para que sean accesibles por todas las clases (nosotros, salvo excepciones las definiremos *private*). Un ejemplo de este tipo de constantes lo encontramos en el propio lenguaje Java y la clase *Math*:

```
public class Math
{
    .....
    public static final double PI = 3.1415192....;
    public static final double E = 2.718281;
    .....
}
```

Para utilizar la constante de la clase *Math*: `perimetro = 2 * Math.PI * radio;`, se antepone el nombre de la clase (no hay que crear instancias de la clase *Math*).

#### 4.13.2.- Variable de clase (variables *static*)

Una clase puede declarar *variables de clase* o variables *static*, es decir, variables que pertenecen a la clase y no a las instancias.

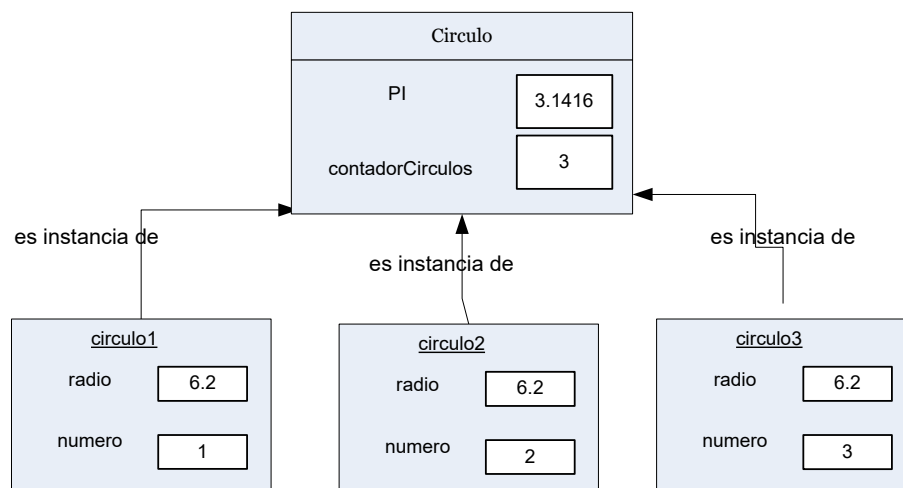
Para declarar una variable de clase se especifica *static* después de la visibilidad y antes del tipo de la variable.

**Ej.** `private static int contador;`

Imaginemos que en la clase `Circulo` anterior queremos incluir un nuevo atributo entero *numero* de forma que cada vez que creamos un nuevo objeto `Circulo` este atributo guarde el nº de círculo creado (1 para el 1º, 2 para el 2º, 3 para el 3º, ...). ¿Cómo conseguimos esto? Se necesita un contador que no esté asociado a ninguna instancia (si lo estuviera el constructor asignaría siempre el mismo valor inicial) y que se incremente cada vez que se crea un nuevo objeto `Circulo`. La solución es utilizar una variable de clase y asignar el valor de esta variable al atributo *numero* dentro del constructor.

```
public class Circulo
{
    private static final double PI = 3.1416;
    private static int contadorCirculos = 0;
    private double radio;
    private int numero;
    .....

    public Circulo()
    {
        contadorCirculos++;
        numero = contadorCirculos;
    }
    .....
}
```



Cada vez que se llama al constructor de la clase `Circulo` la variable *static* `contadorCirculos` se incrementa en 1 y su nuevo valor se asigna al atributo *numero*.

Las variables de clase mantienen el estado de la clase compartido por las instancias.

Java asigna un valor por defecto (igual que con las variables de instancia) a este tipo de variables : 0 para las de tipo `int`, 0.0 para `double`, *null* para las de tipo `String`, .... Las variables de clase se inicializan por 1ª vez cuando se carga la clase.

Dentro de una clase se definen antes que las variables de instancia y después de las constantes.

Los métodos de instancia pueden acceder a las variables *static* de la misma forma que lo hacen con las variables de instancia.

No utilizaremos este tipo de variables habitualmente.

**Ejer.4.18.** Crea un proyecto que incluya una clase `Empleado` con la siguiente estructura:

```
public class Empleado
{
    private int numeroEmpleado;
    .....

    public Empleado()
    {
        // a completar
    }
    .....
}
```

- Completa la definición de la clase y el constructor de forma que el atributo *numeroEmpleado* guarde un valor consecutivo 1, 2, 3, ... por cada nuevo empleado creado.
- Prueba la clase desde BlueJ creando varios empleados e inspeccionando su estado. Observa como BlueJ muestra también los miembros *static* de una clase.

### 4.13.3.- Métodos de clase (métodos *static*)

Los métodos también pueden definirse como *static* en una clase. Como el resto de miembros de clase, los métodos *static* se asocian a la clase y no a las instancias por lo que para ser invocados no se necesita crear ningún objeto (instancia) de la clase.

Un método se declara de clase anteponiendo la palabra *static* después de su visibilidad.

**Ej.**

```
public static int máximo(int numero1, int numero2)
public static double factorial(double numero)
```

La clase `Math` incorpora varios métodos *static*. Esta clase funciona como una librería de utilidades donde los métodos *static* devuelven un valor de retorno a partir de los parámetros que se les pasa.

```
public class Math
{
    .....
    public static double sqrt(double n)
    public static double random()
}
```

Los métodos de clase no pueden acceder a las variables de instancia sólo a las variables de clase. Estos métodos no operan sobre objetos, se invocan con el nombre de la clase. Tampoco pueden llamar a los métodos de instancia (al revés sí, los métodos de instancia pueden llamar a los métodos de clase). `double resultado = Math.sqrt(89);`

```

public class Libreria
{
    .....
    public static int    maximo(int numero1, int numero 2)
    {
        .....
    }
    public static double    factorial(double numero)
    {
        .....
    }
}

int valorMaximo = Libreria.maximo(7,89);
double facto = Librería.factorial(6);

```

Es habitual cuando se define una variable de clase definir también un método de clase para obtener su valor: `public static int getContadorCircuitos()`

Desde la versión Java 1.5 es posible importar variables y métodos *static* sin necesidad de especificar al utilizarlos la clase a la que pertenecen:

```

import static java.lang.Math.abs; ó    import static java.lang.Math.*;
import static java.lang.Math.max;
int nuevoX = abs(destino.getX() - x);

```

No utilizaremos métodos *static* (salvo que se pida explícitamente).

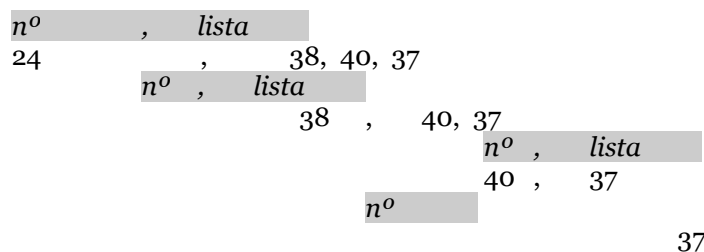
## 4.14.- Recursión. Métodos recursivos

La **recursión** es una técnica utilizada en programación que proporciona una solución elegante y sencilla a cierta clase de problemas.

Una definición es *recursiva* si el concepto o palabra que se define aparece en la propia definición. Por ejemplo, imaginemos la lista de n<sup>os</sup>: 24, 38, 40, 37 podríamos definir la lista así,

- una lista es un n<sup>o</sup> o
- una lista es un n<sup>o</sup> y una , junto con una lista

Aquí el concepto de lista aparece en la propia definición.



Todas las definiciones recursivas tienen una parte que no lo es (sino la recursión sería infinita), esta parte no recursiva es lo que se denomina *caso base* o *caso trivial*.

Aplicado a la programación el uso de la recursión es idóneo en problemas cuyos cálculos o estructuras de datos pueden definirse en términos recursivos y es una *alternativa a la iteración*. Cualquier problema que se resuelva de forma recursiva puede resolverse también sin recursión, con una solución iterativa, sólo que muchas veces esta última es más compleja.

Detrás de la recursión se esconde el principio del “*divide y vencerás*”, el problema inicial se plantea en términos de problemas más pequeños pero de las mismas características, similar al original.

Veamos un ejemplos de método recursivo teniendo en cuenta que el método manejará únicamente tipos primitivos (por sencillez, no aplicaremos la recursión a los objetos).

Un método es recursivo si puede “llamarse” (invocarse) a sí mismo.

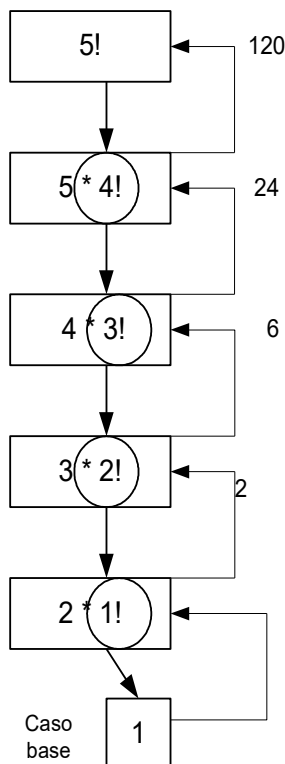
Planteemos el cálculo del factorial de un  $n^o$  en términos recursivos.

$$\begin{aligned} 5! &= 5 * 4 * 3 * 2 * 1 \\ 4! &= 4 * 3 * 2 * 1 \\ 3! &= 3 * 2 * 1 \\ 2! &= 2 * 1 \\ 1! &= 1 \\ 0! &= 0 \end{aligned}$$

$$\begin{aligned} \text{El factorial de } 5! &= 5 * 4! = \\ &= 5 * 4 * 3! = \\ &= 5 * 4 * 3 * 2! = \\ &= 5 * 4 * 3 * 2 * 1! = \dots \end{aligned}$$

En general, el problema  $n!$  expresado recursivamente queda:

$$\begin{array}{ll} 1 & \text{si } n = 1 \text{ o } n = 0 \text{ es el } \mathbf{\text{caso base o caso trivial}}, \text{ el que se resuelve sin} \\ & \text{recursión} \\ n * (n - 1)! & \text{si } n > 1 \quad \mathbf{\text{caso general}} \end{array}$$



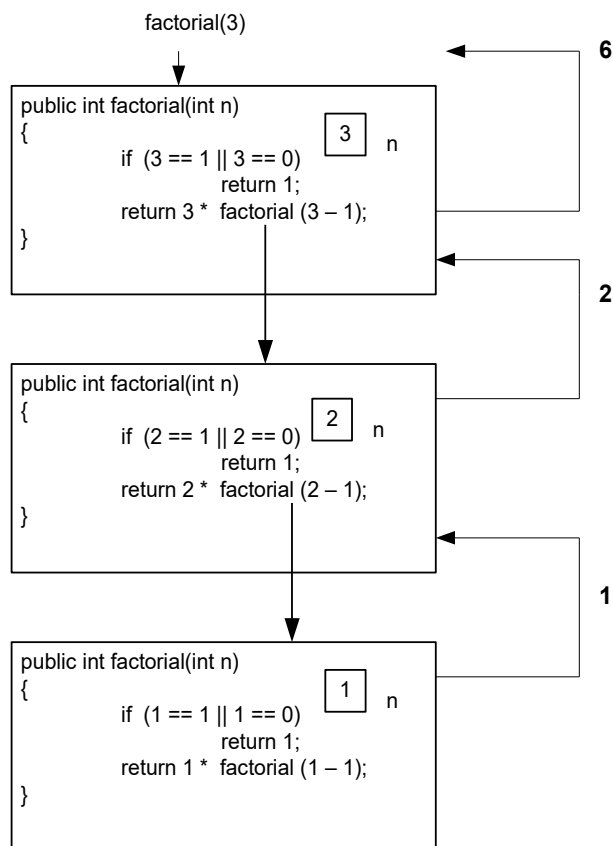
En todo problema recursivo hay que encontrar primero el caso base y luego el caso general en el que se incluirá la llamada al método recursivo.

En Java quedaría:

```
public int factorial(int n)
{
    if (n == 1 || n == 0) {
        return 1;
    }
    return n * factorial (n - 1);
}
```

```
public int factorial(int n)
{
    int resul;
    if (n == 1 || n == 0) {
        resul = 1;
    }
    else {
        resul = n * factorial(n - 1);
    }
    return resul;
}
```

Cada nueva llamada al método supone un nuevo entorno de ejecución para él con nuevos parámetros y variables locales que no son visibles de una llamada a otra.



**Ejer.4.19.**

- Crea un proyecto Librería recursiva e incluye una clase Librería. Esta clase únicamente va a contener métodos *static* y no tendrá atributos (será uno de los pocos ejemplos en los que trabajaremos con métodos de clase). Los métodos van a ser utilidades y serán métodos recursivos. Todos los métodos son *public*.
- Incluye el método `int factorial(int num)` y Pruébalo.
- Añade un método `int sumatorio (int num)` que devuelve el sumatorio del parámetro proporcionado al método. Si `num = 5` devolverá `1 + 2 + 3 + 4 + 5 = 15`
- Define el *método* `int maximoComunDivisor(int a, int b)` que calcula y devuelve el máximo común divisor de `a` y `b` utilizando el *algoritmo de Euclides*
- Construye el método `int contarDigitos(int num)` que devuelve el `nº` de cifras de `num`
- Método `int potencia(int base, int expon)` que calcula  $base^{expon}$
- Método `int fibonnaci(int n)` que calcula y devuelve el `n-simo` término de la serie de Fibonacci. La serie de Fibonacci es: 0 1 1 2 3 5 8 13 ..... Cada término es la suma de los dos anteriores. Si el `nº` de término a calcular es el 7º el método devolverá 8.