

## UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEMENTOS DEL LENGUAJE

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## Indice

1. Cuarta vuelta: otros elementos del lenguaje
2. Quinta vuelta: estructuras de control condicionales
3. Sexta vuelta: estructuras de control repetitivas
4. Séptima vuelta: arrays

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 1 - Cuarta vuelta: otros elementos del lenguaje

### ► 1 – Cuarta vuelta: otros elementos del lenguaje

#### ► 1.1 – Comentarios y documentación

Hay un tipo de comentarios que permite generar una **documentación HTML** a partir de las clases que hemos creado.

Esta documentación tiene el mismo formato que la documentación oficial de Java. Ver ejemplo: <https://docs.oracle.com/javase/8/docs/api/>

Este tipo de comentarios añadidos al código empiezan por `/**` y terminan con `*/`

Además, pueden llevar etiquetas especiales que empiezan por `@` y que permiten aportar más información a nuestro código.

**Javadoc** es el nombre de la utilidad que procesa estos comentarios para generar la documentación. Es una utilidad que normalmente integran los IDEs.

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 1 - Cuarta vuelta: otros elementos

► Veamos un ejemplo:

```
package com.kike.u2.p1.javadoc;
```

```
/**
```

```
 * Esta clase tiene un propósito académico y permite presentar el uso de las  
 * propiedades y métodos de una clase.
```

```
 *
```

```
 * @author Kike
```

```
 * @version 1.0
```

```
 */
```

```
public class Bombilla {
```

```
    /**
```

```
     * Representa la marca de la bombilla.
```

```
    */
```

```
    public String marca;
```

```
    /**
```

```
     * Representa la potencia en Watios de la bombilla.
```

```
    */
```

```
    public int potencia;
```

```
    /**
```

```
     * Expresa mediante un valor lógico si está encendida o no.
```

```
    */
```

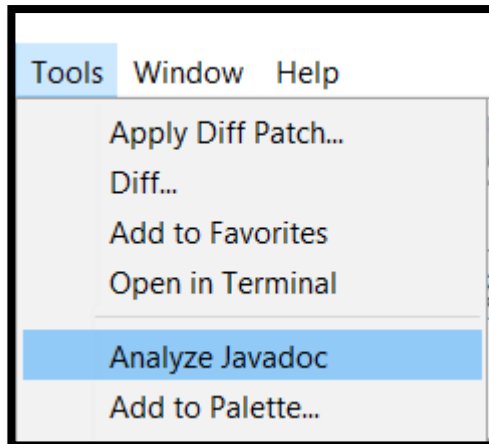
```
    public boolean encendida;
```

► ...continúa

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 1 - Cuarta vuelta: otros elementos

- ▶ La sintaxis es muy sencilla.
- ▶ NetBeans nos ayuda a escribirla. Si ponemos `/**` y pulsamos **INTRO**, ya nos crea la plantilla.
- ▶ Además, con la siguiente opción, nos avisa de lo que nos falta y nos crea las plantillas automáticamente.



```
/**
 * Este método permite cambiar la propiedad marca de la Bombilla
 *
 * @param nuevaMarca cadena con el nuevo valor de la marca
 */
public void cambiaMarca(String nuevaMarca) {
    marca = nuevaMarca;
}

/**
 * Este método imprime la marca de la bombilla en pantalla.
 */
public void imprimeMarca() {
    System.out.println("La marca de la bombilla es " + marca);
}

/**
 * Este método permite cambiar la propiedad potencia de la Bombilla
 *
 * @param nuevaPotencia entero con el nuevo valor de la potencia
 */
public void cambiaPotencia(int nuevaPotencia) {
    potencia = nuevaPotencia;
}
...
```

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 1 - Cuarta vuelta: otros elementos

- Para generar la documentación HTML, que es completamente **navegable**, seguimos el menú: **Run -> Generate Javadoc**

```
public class Bombilla  
extends Object
```

Esta clase tiene un propósito académico y permite presentar el uso de las propiedades y métodos de una clase.

**Version:**

1.0

**Author:**

Kike

### Field Summary

#### Fields

Modifier and Type	Field	Description
boolean	<b>encendida</b>	Expresa mediante un valor lógico si está encendida o no.
<b>String</b>	<b>marca</b>	Representa la marca de la bombilla.
int	<b>potencia</b>	Representa la potencia en Watios de la bombilla.

Realiza el  
ejercicio 1 del  
boletín de  
problemas

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 1 - Cuarta vuelta: otros elementos

### ► 1.2 – Impresión con formato

Vamos a ver otras formas de imprimir en la pantalla del ordenador:

```
System.out.println("Escribe este texto y pasa a la siguiente línea");
```

```
System.out.print("Escribe este texto pero no pasa a la siguiente línea");
```

Ahora vamos a usar otro método nuevo que permite añadir unos elementos nuevos, llamados **especificadores de formato**, que indican **cómo queremos que se imprima la información**. Su sintaxis es:

```
System.out.printf("Texto con especificadores de formato", param1, param2, ...);
```

► Veámoslo con ejemplos:

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 1 - Cuarta vuelta: otros elementos

### ► *Especificadores de formato con cadenas de texto:*

```
System.out.printf("Esto imprime un %s", "TEXTO");
```



```
System.out.printf("Esto imprime un %s y luego %s", "TEXTO", "OTRO");
```




Estas dos líneas imprimirían en la pantalla:

Esto imprime un TEXTO  
Esto imprime un TEXTO y luego OTRO

Para imprimir un el carácter de fin de línea se usa el especificador %n.

```
System.out.printf("Esto %nse imprime%n así");
```



Esto  
se imprime  
así



# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 1 - Cuarta vuelta: otros elementos

System.out.**printf**("'%s'", "TEXTO"); ➡ 'TEXTO'

Forzamos a que el texto ocupe 15 caracteres y rellenamos con espacios por la izquierda, si es necesario:

System.out.**printf**("'%15s'", "TEXTO"); ➡ ' TEXTO'

Forzamos a que el texto ocupe 15 caracteres y rellenamos con espacios por la derecha, si es necesario:

System.out.**printf**("'%-15s'", "TEXTO"); ➡ 'TEXTO '

Esto permite imprimir información formateada... Ejemplo:

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 1 - Cuarta vuelta: otros elementos

```
Persona p = new Persona();  
p.setNombre("Pedro");  
p.setApellidos("López Haro");  
p.setDni("98712345K");  
System.out.printf("%-15s%15s%n", "Nombre:", p.getNombre());  
System.out.printf("%-15s%15s%n", "Apellidos:", p.getApellidos());  
System.out.printf("%-15s%15s%n", "DNI:", p.getDni());  
Esto imprimiría en pantalla del siguiente modo:
```

Nombre:	Pedro
Apellidos:	López Haro
DNI:	98712345K

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 1 - Cuarta vuelta: otros elementos

### ► *Especificadores de formato con números:*

System.out.printf("%d", 1473); ➡ 1473 // Números enteros

System.out.printf("%8d", 1473); ➡ ' 1473' // Se rellena con espacios por  
//la izquierda hasta 8 caracteres

System.out.printf("%f", 5.1473); ➡ 5.147300 // Con 6 decimales

System.out.printf("%.4f", 5.1473); ➡ 5.1473 // Con 4 decimales

System.out.printf("%10.4f", 5.1473); ➡ ' 5.1473' // Con 4 decimales  
// Se rellena con espacios por  
//la izquierda hasta 10 caracteres

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 1 - Cuarta vuelta: otros elementos

- Hay otros especificadores de formato para fechas, booleanos, caracteres... para forzar a imprimir todo en mayúsculas...
- Para saber más: <https://www.baeldung.com/java-printstream-printf>

Realiza el ejercicio 2 del  
boletín de problemas

PRIMER PLATO	SEGUNDO PLATO
Ensalada de la casa	Solomillo al whisky
-----	
CALORÍAS	PRECIO
798	11,95€

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 1 - Cuarta vuelta: otros elementos

### ► 1.3 – Literales de los tipos primitivos que no usamos normalmente

Cuando Java encuentra un número entero como el 23454, lo asocia al tipo **int**

Si se encuentra un número real como el 3242.89, lo asocia al tipo **double**

En el caso de que queramos utilizar enteros grandes (tipo **long**) o números reales de precisión simple (tipo **float**), tendremos que añadir un sufijo al valor para aclarárselo a Java y que no lo entienda como uno de los casos anteriores.

Veamos los ejemplos:

```
int entero1 = 1233;
```

```
long entero2 = 1233L;
```

```
double real1 = 2343.34;
```

```
float real2 = 2343.34F;
```

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 1 - Cuarta vuelta: otros elementos

### ► 1.4 – Tipo de una expresión. Conversiones de tipo.

Cuando el ordenador ejecuta una expresión obtiene un resultado de un tipo de dato concreto (int, double, String...)

**precio** = 5 + 6 + 45/5; // 5 + 6 + 9 = 20 (el resultado es un número entero)

Si la expresión devuelve como resultado un entero (int) se dice que “la expresión es de tipo int o entero”.

Si tuviéramos algo como **precio** = 12.99 + 12.99\*0.21 // 12.99 + 2.7279 = 15.7179 (double) entonces diríamos que “la expresión es de tipo double o real”

Si tuviéramos algo como:

**nombreCompleto** = “Arturo” + “+” “López Haro” // “Arturo “+”López Haro” = “Arturo López Haro”

entonces diríamos que “la expresión es de tipo String o cadena de texto”

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 1 - Cuarta vuelta: otros elementos

Con esto podemos decir que el “**tipo de una expresión**” es el tipo del dato resultante cuando la expresión es ejecutada por nuestro programa.

Ahora bien ¿qué ocurre cuando se mezclan distintos tipos de datos en una misma expresión?

```
public class PruebaTipoExpresion {  
    public static void main(String[] args) {  
        double suma;  
  
        suma = 5 + 6 + 9.5; // int + int + double  
  
        System.out.println("suma = "+suma);  
    }  
}
```

Se imprime:



suma = 20.5

Parece lógico que el tipo de la expresión sea “double” ya que los números reales pueden contener a los números enteros y así se ofrece el resultado exacto.

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 1 - Cuarta vuelta: otros elementos

ANTES de evaluar una expresión, Java busca el tipo de dato con más capacidad y **CONVIERTE AUTOMÁTICAMENTE** los tipos de datos de menor capacidad al tipo de mayor capacidad.

Entonces nuestra expresión **suma = 5 + 6 + 9.5;**

Java la convierte a esta otra: **suma = 5.0 + 6.0 + 9.5;**

Esto ocurre porque Java tiene un operador de suma para números enteros: **int + int = int**

Y también tiene un operador de suma para números reales: **double + double = double**

Pero **NO EXISTE** un operador de suma que mezcle ambos tipos: **int + double = NO EXISTE**

Por esta razón, Java **BUSCA** el tipo de “mayor capacidad” y convierte todos los elementos de la expresión a este tipo, pudiendo así aplicar un operador (+, -, \*, /...) en el que no se mezclen distintos tipos de datos.

A esto se le llama CONVERSIÓN AUTOMÁTICA DE TIPOS o CASTING IMPLÍCITO



# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 1 - Cuarta vuelta: otros elementos

Pongamos otro caso, este un poco más extraño:

```
public class PruebaTipoExpresion {  
    public static void main(String[] args) {  
        double notaMedia1, notaMedia2;  
  
        notaMedia1 = (5 + 6) / 2;  
        notaMedia2 = (5 + 6) / 2.0;  
  
        System.out.println("notaMedia1 = "+notaMedia1);  
        System.out.println("notaMedia2 = "+notaMedia2);  
    }  
}
```




Se imprime:

```
notaMedia1 = 5  
notaMedia2 = 5.5
```

### EXPLICACIÓN:

- Java tiene dos operadores de división, uno para enteros:  $\text{int} / \text{int} = \text{int}$
- y otro para números reales:  $\text{double} / \text{double} = \text{double}$



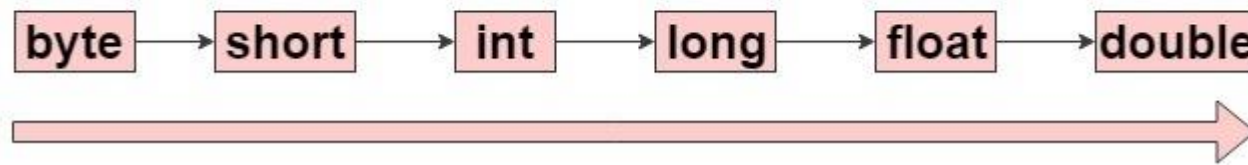
11	2
1	5

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 1 - Cuarta vuelta: otros elementos

Dado que estas conversiones se realizan **SIN PÉRDIDA DE INFORMACIÓN** se realizan de forma **automática** por la máquina virtual java.

### Automatic Type Conversion (Widening - implicit)

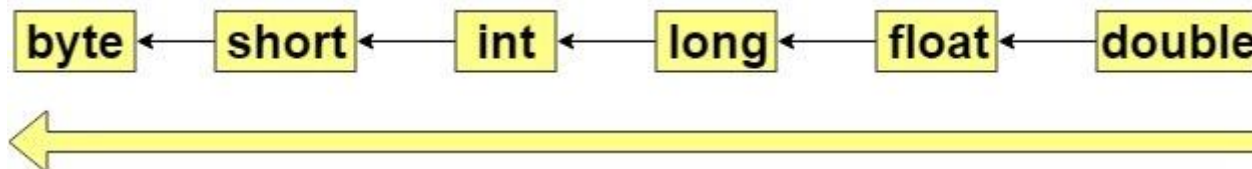


Un byte “cabe” en un short, y este en un int...

Widening es “ensanchando”

Sin embargo, en ocasiones nos puede hacer falta una conversión en “sentido contrario”, es decir, una conversión **CON PÉRDIDA DE INFORMACIÓN**. En este caso, el programador tiene que indicar explícitamente en el código que no se trata de un error y que es eso lo que desea hacer realmente.

### Narrowing (explicit)



Un double “no cabe” en un float, y este tampoco cabe en un long...

Narrowing es “estrechando”

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 1 - Cuarta vuelta: otros elementos

Veamos un ejemplo:

```
int notaTrim1 = 5;
```

```
int notaTrim2 = 5;
```

```
int notaTrim3 = 6;
```

```
double mediaReal = (notaTrim1 + notaTrim2 + notaTrim3) / 3.0;
```

```
System.out.println("La nota media con decimales es: "+ mediaReal);
```

 ➡ Se imprime 5,33

```
System.out.println("La nota media entera es: "+ (int) mediaReal);
```

 ➡ Se imprime 5

Estamos convirtiendo un **double** a un **int**, así que estamos **perdiendo información**. La parte decimal se elimina, no se redondea. Para que Java nos permita hacer este tenemos que escribirlo explícitamente con la notación:

**(tipoMenor)** expresiónConUnTipoMayor

*A esto se le llama CONVERSIÓN EXPLÍCITA o CASTING EXPLÍCITO*

Realiza el ejercicio 3 del boletín de problemas

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 1 - Cuarta vuelta: otros elementos

### ► 1.5 – Declaración abreviada de variables con 'var'

Cuando declaramos e inicializamos una referencia a un objeto tenemos que “**repetirnos**”. Observa:

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

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

```
    Persona p = new Persona();
```

```
    ...
```

```
}
```

Tenemos la posibilidad de evitar la repetición utilizando la declaración abreviada de variables con '**var**'. Es muy sencillo:



# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 1 - Cuarta vuelta: otros elementos

Usando 'var' el código anterior quedaría así:

```
public static void main (String[] args) {  
    var sc = new Scanner(System.in);  
    var p = new Persona();  
    ...  
}
```

Es como decirle a NetBeans: “Te voy a declarar una **variable** y el tipo del dato **lo debes deducir** de la expresión que está a la derecha de la asignación”.

Eso nos obliga a asignarle un valor siempre a nuestra variable. La siguiente instrucción daría un error de compilación:

```
var p;
```



# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 1 - Cuarta vuelta: otros elementos

Aunque el uso de '**var**' con variables de tipo referencia a objetos es el más habitual, también se puede usar para:

- **Tipos de datos primitivos:**

```
var edad = 32;
```

```
var estaLleno = false;
```

```
var precio = 19.99;
```

- **Declarar propiedades de una clase:**

```
public class Persona {  
    public var nombre = "Anónimo";  
    public var edad = 0;  
    //...  
}
```

OJO: var está disponible desde Java 10. Las versiones anteriores no lo soportan

V  
I  
V  
A

VAR



# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 1 - Cuarta vuelta: otros elementos

### ► 1.6 – Ámbito de una variable

Hasta ahora para declarar una variable lo hacíamos, más o menos, en la parte del código que nos hacía falta o en la zona en la que se nos venía a la cabeza. Sin embargo, la **zona del código en la que se declara la variable es importante**. Vamos a ver en qué sentido.

Cada vez que abrimos y cerramos unas llaves **{ }** estamos definiendo una ZONA DE CÓDIGO o ÁMBITO DE CÓDIGO del siguiente modo:

```
public void setEdad (int nuevaEdad)
```

```
{ // Inicio de ámbito
```

```
    Instrucciones dentro del ámbito
```

```
} // Fin de ámbito
```

Cada vez que creamos una clase o un método estamos definiendo ámbitos de código.

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 1 - Cuarta vuelta: otros elementos

Además, podríamos definir un ámbito dentro de otro:

```
{ // Inicio de ámbito externo  
    Instrucciones del ámbito externo  
    { // Inicio de ámbito interno  
        Instrucciones del ámbito interno  
    } // Fin de ámbito interno  
    Más instrucciones del ámbito externo  
} // Fin de ámbito externo
```

Las nuevas instrucciones que vamos a estudiar en este tema definen sus propios ámbitos de código, por eso es importante entender cómo funciona la relación entre variables y ámbitos.



# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 1 - Cuarta vuelta: otros elementos

Las variables y los ámbitos se relacionan del siguiente modo:

- Una variable **SOLO EXISTE** dentro del ámbito en el que se declara.  
Ejemplo:

```
{ // Inicio de ámbito 1
  int numAmbito1;
  { // Inicio de ámbito 2
    int numAmbito2;
    System.out.println("Suma = "+(numAmbito1+numAmbito2));
  } // Fin de ámbito 2

  System.out.println("numAmbito2 aquí ya no existe. Está fuera de su ámbito de declaración");
  System.out.println("numAmbito1 sí existe aquí. Sigue dentro de su ámbito de declaración");
} // Fin de ámbito 1
```

- Si una variable se declara en un ámbito "externo" que contiene otros ámbitos "internos", **esta variable también existe en los ámbitos internos.**

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

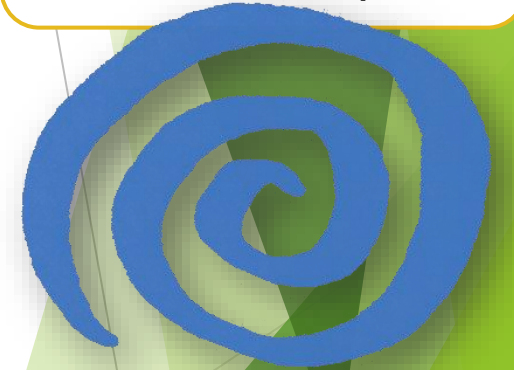
## 1 - Cuarta vuelta: otros elementos

Si intentamos utilizar una variable fuera del ámbito en el que fue declarada, el compilador nos dará un error sintáctico

```
3 public class Circunferencia {
4     public double radio;
5
6     public void imprimePerimetro() {
7         double perimetro = 2*Math.PI*radio;
8         System.out.println("Perimetro = "+perimetro);
9     }
10
11     public double getPerimetro() {
12         return perimetro;
13     }
14 }
15
```

cannot find symbol  
symbol: variable perimetro  
location: class Circunferencia  
----  
(Alt-Enter shows hints)

No olvides estudiar la chuleta de la 4ª vuelta a la espiral



Se suele decir también que una variable ES VISIBLE en su ámbito y NO VISIBLE fuera de su ámbito.

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

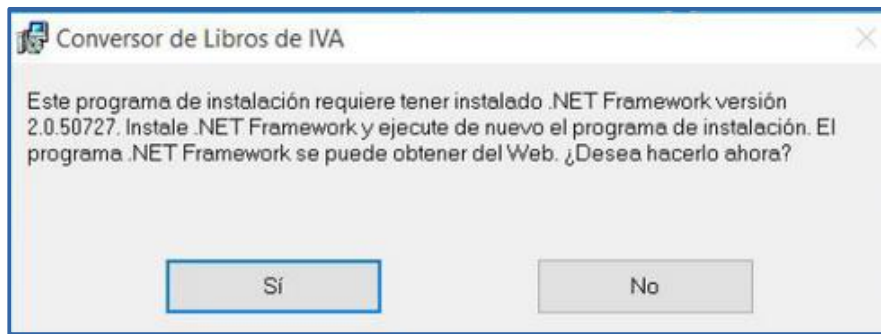
## 2 - Quinta vuelta de espiral: condicionales

### ► 2 - Quinta vuelta - Estructuras de control condicionales

Hasta ahora nuestros bloques de instrucciones realizaban una **serie de pasos de forma secuencial o lineal**.

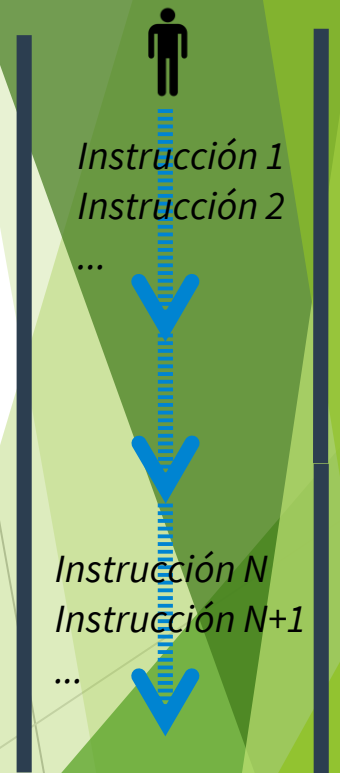
Las estructuras de control son instrucciones que permiten romper esa secuencialidad/linealidad.

Queremos que nuestros programas puedan hacer una cosa u otra dependiendo de ciertas condiciones. Ejemplo:



¿Se comportará igual el programa si el usuario pincha en 'Sí' que si pincha en 'No'?

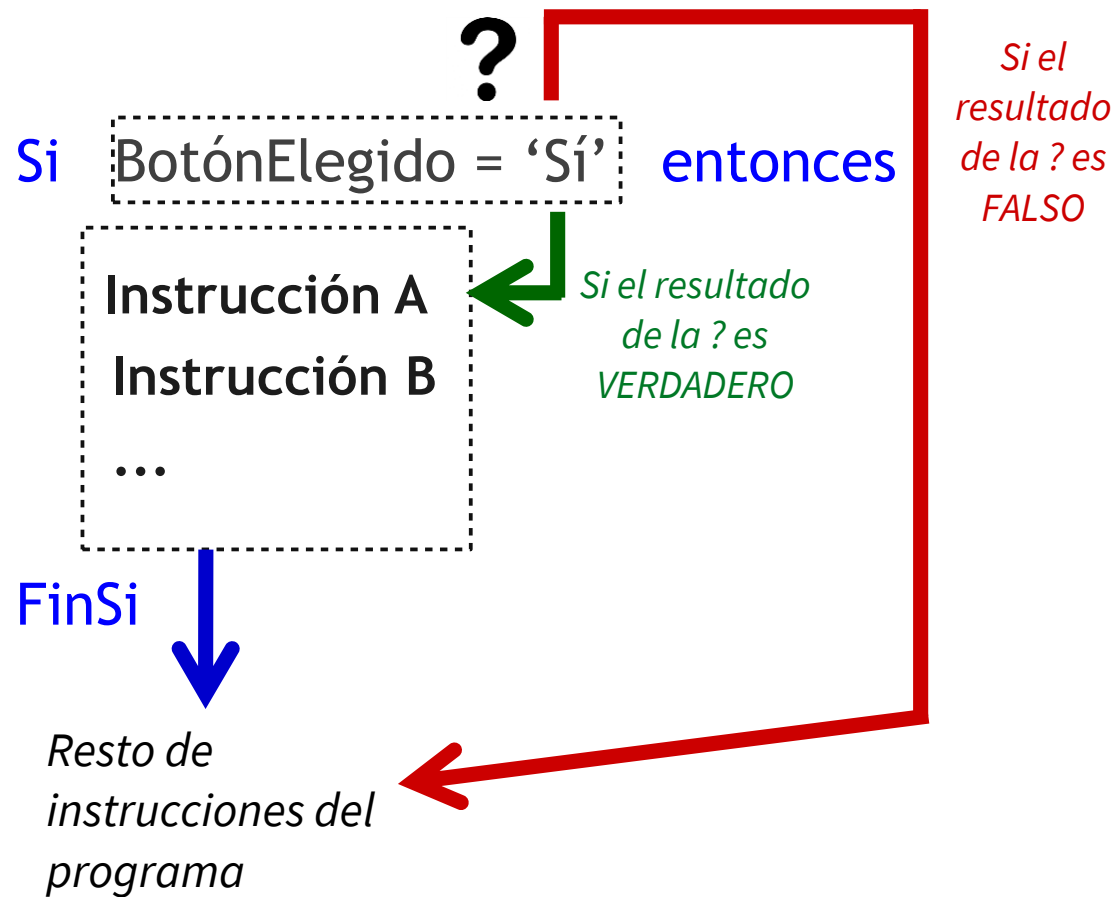
¿Debería el programa tener un conjunto de instrucciones para cada una de las posibles respuestas del usuario?



# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 2 - Quinta vuelta de espiral: condicionales

Lo que buscamos ahora es una instrucción como esta:



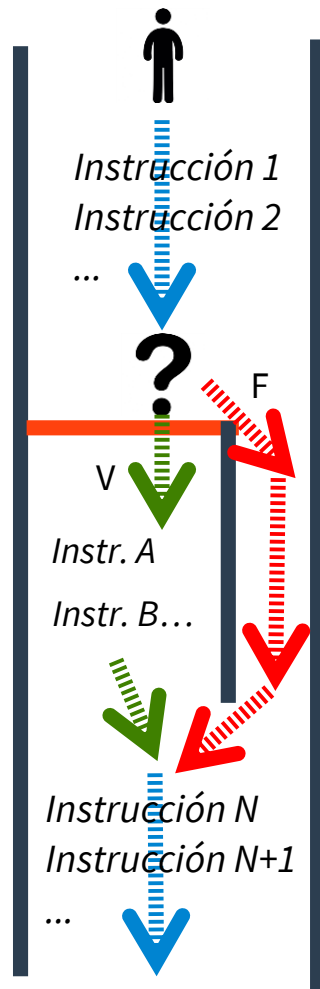
Dependiendo del resultado de la pregunta el programa escoge un camino u otro:

- Si BotónElegido = 'Sí' es VERDADERO entonces se ejecutará el bloque de instrucciones interno y después se continuará con las instrucciones del resto del programa.
- Si BotónElegido = 'Sí' es FALSO entonces NO se ejecuta el bloque interno y se pasa directamente a las instrucciones del resto del programa

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 2 - Quinta vuelta de espiral: condicionales

El programa se comporta como si fuéramos andando por “*pasillo de instrucciones*” y de pronto nos encontramos con una puerta cerrada:



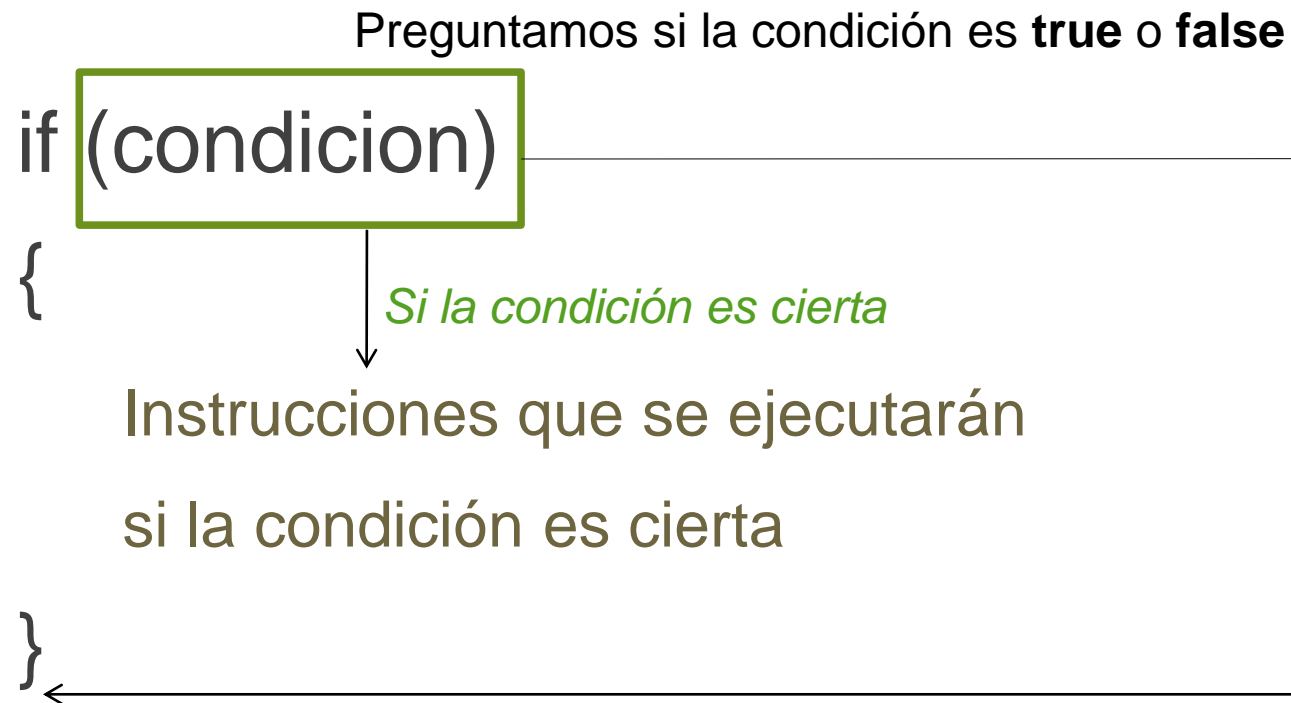
Nuestro programa va por “el pasillo de instrucciones” ejecutando las que se encuentra. De pronto llega a una puerta cerrada que le hace una pregunta, pudiendo pasar dos cosas:

- Si el resultado de la pregunta es VERDADERO entonces pasará a la habitación y continuará ejecutando las instrucciones que allí se encuentre. Después saldrá de la habitación y continuará su camino por el pasillo.
- Si el resultado de la pregunta es FALSO entonces no entrará en la habitación, cogerá por el pasillito de al lado y continuará su camino por el pasillo de instrucciones.

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

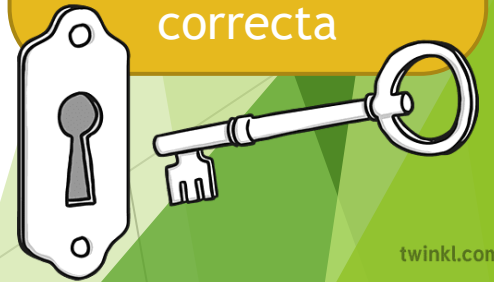
## 2 - Quinta vuelta de espiral: condicionales

A esta estructura se le llama Condicional Simple y su sintaxis en Java es la siguiente:



*Si la condición es  
falsa salimos de la  
estructura  
condicional simple*

La condición funciona como una cerradura que solo se abrirá si la llave es la correcta



# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 2 - Quinta vuelta de espiral: condicionales

### Ejemplo:

```
System.out.print("Estimado ");  
if (edad >= 18)  
{  
    System.out.print("Don ");  
}  
System.out.println("Alberto");
```

Si edad  $\geq 18$  es verdadero

Estimado Don Alberto

Si edad  $\geq 18$  es falso

Estimado Alberto



# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 2 - Quinta vuelta de espiral: condicionales

- Vamos a probarlo. Teclea las siguientes clases.
- Usa **Alt + Insert** para generar los getters y setters de la clase Persona

*Guárdalo todo, lo usaremos más adelante*

```
public class Persona {  
  
    public int edad;  
    public String nombre;  
  
    public int getEdad() {  
        return edad;  
    }  
  
    public String getNombre() {  
        return nombre;  
    }  
  
    public void setEdad(int edad) {  
        this.edad = edad;  
    }  
  
    public void setNombre(String nombre) {  
        this.nombre = nombre;  
    }  
}
```

```
import java.util.Scanner;  
public class PruebaPersona {  
  
    public static void main(String[] args) {  
        Persona p = new Persona();  
        Scanner sc = new Scanner(System.in);  
  
        System.out.println("Introduce el nombre: ");  
        p.setNombre(sc.nextLine());  
  
        System.out.println("Introduce la edad: ");  
        p.setEdad(sc.nextInt());  
  
        System.out.print("Estimado ");  
        if(p.getEdad() >= 18)  
        {  
            System.out.print("Don ");  
        }  
        System.out.println(p.getNombre());  
    }  
}
```



# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

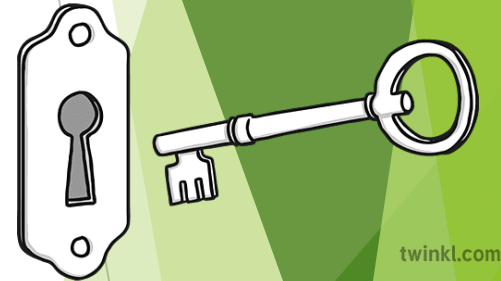
## 2 - Quinta vuelta de espiral: condicionales

Miremos con más atención a la **condición**. Normalmente es una pregunta que relaciona dos valores y devuelven un booleano.

Ejemplos:

- if (importe >= 100) ...
- if (porcentaje <= 0.15) ...
- if (p.getEdad() >= 18) ...
- if (resultado != 0) ...
- if (vivo == true) ...
- if (encendida == false) ...
- if (fundida !=true) ...

Normalmente son del tipo:  
Expresión1 operador Expresión2 -> boolean



# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 2 - Quinta vuelta de espiral: condicionales

Los **operadores relaciones** son aquellos que nos permiten hacer una pregunta cuya respuesta es un valor cierto o falso.

Los operadores relaciones son:

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

menor, mayor, menor o igual, mayor o igual, igual a, distinto a

**OJO:** no confundir el operador de asignación = con el de comparación de igualdad ==.

Asignación =	Comparación igualdad ==
Edad = 18;	If (edad == 18)
<i>Asigna o copia el valor 18 sobre la propiedad edad</i>	<i>Pregunta si el valor de la propiedad edad es igual a 18</i>



**ERROR MUY FRECUENTE**

Realiza el ejercicio 4 del boletín de problemas

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 2 - Quinta vuelta de espiral: condicionales

### Sentencias condicionales y ámbitos de variables

Cada vez que hacemos un **if** abrimos un nuevo **ámbito de código** delimitado con las { }

Un **error muy frecuente** consiste en declarar una variable en este ámbito y querer utilizarla fuera de dicho ámbito. Ejemplo:

```
if (importe > 100) {  
    double descuento = importe * 10/100;  
}  
System.out.println("El descuento es de "+descuento);
```



La variable **descuento** ha sido declarada dentro del ámbito del if y solo tiene validez y existencia entre las { } que delimitan dicho ámbito. Fuera del ámbito NO EXISTE y el compilador nos lo marca como un error.


## UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

### 2 - Quinta vuelta de espiral: condicionales

Para arreglarlo solo tenemos que sacar la declaración de **descuento** fuera del ámbito del **if**.

```
23      double descuento;  
24      if (importe > 100) {  
25          importe * 10/100;  
26          System.out.println("El descuento es de "+descuento);  
28
```

variable descuento might not have been initialized  
----  
(Alt-Enter shows hints)



Sin embargo, el IDE nos dice ahora que **descuento** podría no haber sido inicializada con un valor si la condición del **if** fuera falsa, así que no sabría qué valor imprimir en pantalla. La solución pasa por declarar **descuento** dándole un “valor por defecto” que tenga cierta coherencia.

```
double descuento = 0.0;  
if (importe > 100) {  
    descuento = importe * 10/100;  
}  
System.out.println("El descuento es de "+descuento);
```



# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 2 - Quinta vuelta de espiral: condicionales

### ► Depuración del código

Dado que nuestro código ahora empezará a seguir **distintos caminos**, es conveniente conocer la utilidad de **DEPURACIÓN DE CÓDIGO** que ofrece NetBeans.

La utilidad de depurar (debug) nos permite ejecutar nuestro código **instrucción a instrucción** pudiendo observar qué camino toma la ejecución. Además, nos permite **observar (watch) la evolución** de los objetos y variables en el tiempo.

Es una herramienta muy potente para encontrar errores y problemas en nuestro código.

Veámoslo con un ejemplo.



Bug = bicho = error

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 2 - Quinta vuelta de espiral: condicionales

- Rescatamos el código que nos permitió introducir las condicionales simples:

```
package com.kike.ud2.condicionalsimple;

public class Persona {

    public int edad;
    public String nombre;

    public int getEdad() {
        return edad;
    }

    public String getNombre() {
        return nombre;
    }

    public void setEdad(int edad) {
        this.edad = edad;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

}
```

```
import java.util.Scanner;
public class PruebaPersona {

    public static void main(String[] args) {
        Persona p = new Persona();
        Scanner sc = new Scanner(System.in);

        System.out.println("Introduce el nombre: ");
        p.setNombre(sc.nextLine());

        System.out.println("Introduce la edad: ");
        p.setEdad(sc.nextInt());

        System.out.print("Estimado ");
        if(p.getEdad() >= 18)
        {
            System.out.print("Don ");
        }
        System.out.println(p.getNombre());
    }

}
```

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 2 - Quinta vuelta de espiral: condicionales

- A continuación vamos a **DEPURAR** (Debug) el código. Para ello debemos marcar una línea que contenga una instrucción ejecutable del código para que el depurador se detenga sobre ella, a esto se le llama **insertar un breakpoint** (punto de ruptura)


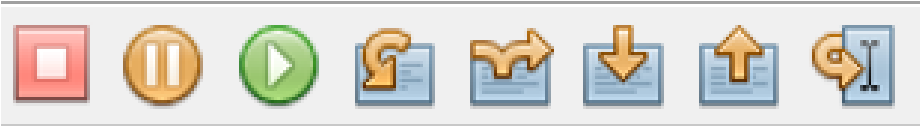
*Hacemos  
doble clic  
sobre el  
número de la  
línea en la  
que deseamos  
poner el  
punto de  
ruptura.  
Se pueden  
poner más de  
uno*

```
8      public class PruebaPersona {
9
10     public static void main(String[] args) {
11         Persona p = new Persona();
12         p.setNombre("Pedro");
13         p.setEdad(13);
14
15         System.out.print("Estimado ");
16         if(p.getEdad() >= 18)
```



# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 2 - Quinta vuelta de espiral: condicionales

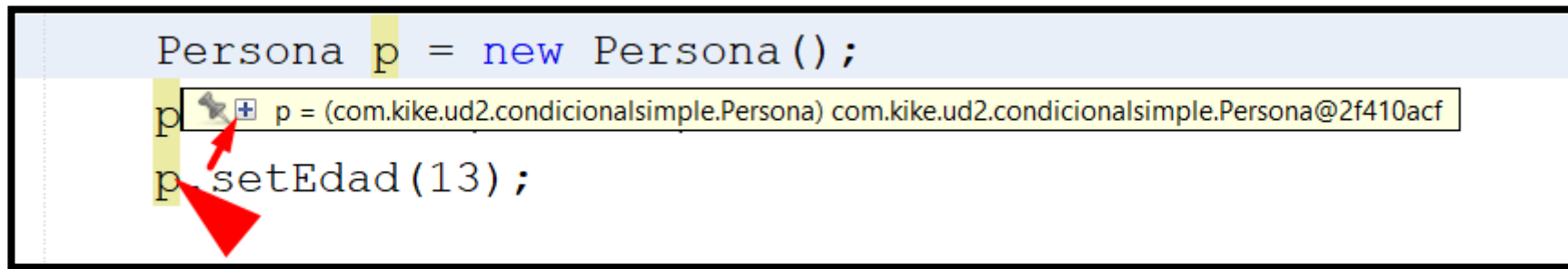
- Presionamos el botón  para iniciar la depuración del código y a continuación, usamos las siguientes opciones para avanzar en la ejecución del código (de izda. a derecha):
- 
- Finaliza (finish) la depuración
  - Detiene momentáneamente (pause) la depuración
  - Continúa (play) con la depuración
  - Avanza a la siguiente línea (step over) – F8
  - Avanza a la siguiente línea (step over expression) pero observando lo que ocurre en la expresión de una instrucción (si la hay). *Todavía no nos hace falta.*
  - Avanza en la depuración pero entrando en los métodos (step into) que se encuentren en el código – F7
  - Avanza hasta salir del método (step out) en el que se entró con la opción anterior
  - Avanza hasta donde esté posicionado el cursor (run to cursor) – F4



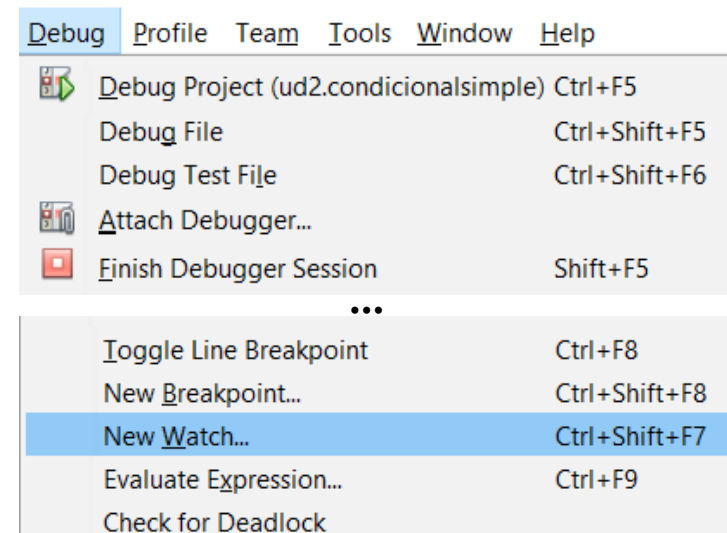
# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 2 - Quinta vuelta de espiral: condicionales

- ▶ Además, durante la depuración podemos observar el valor de cualquier elemento del código de dos formas:
- Poniendo el ratón sobre el elemento y desplegando con el +



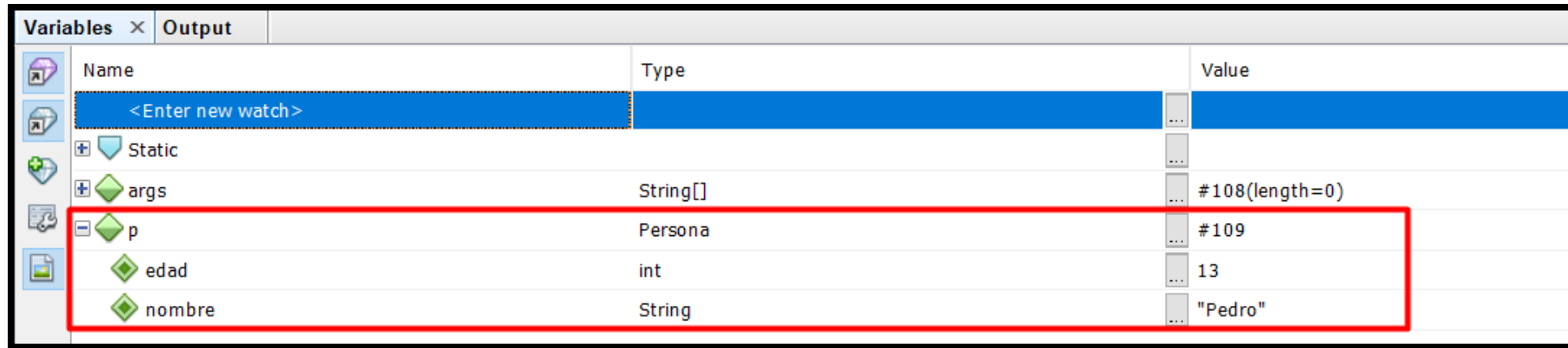
- Añadiendo un “Watch” (elemento a observar). Nos preguntará el nombre del elemento y a continuación nos creará una ventana en la zona de salida (output) de nuestro programa en la que podremos observar un elemento elegido



# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 2 - Quinta vuelta de espiral: condicionales

Elementos observados durante la depuración:



Name	Type	Value
<Enter new watch>		
Static		
args	String[]	#108(length=0)
p	Persona	#109
edad	int	13
nombre	String	"Pedro"

*La utilidad de **DEPURACIÓN** nos ayuda a comprender mejor cómo fluye la ejecución de nuestro código y será de gran ayuda cuando tengamos que arreglar errores “inexplicables” o “expedientes X”*

Realizar el ejercicio 5 del boletín

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 2 - Quinta vuelta de espiral: condicionales

### ► Condiciones complejas

Si queremos comprobar más de una condición dentro de un **if** debemos usar los **operadores lógicos** para relacionarlas entre sí.

Los operadores lógicos son: && (Y), || (O), ! (NO)

Ejemplos:

```
if (edad >= 18 && estaCasado == true) ...
```

```
if (tieneTrabajo == true || estaJubilado == false) ...
```

```
if ( ! edad >= 18) ...
```

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 2 - Quinta vuelta de espiral: condicionales

- Las tablas que resumen cómo funcionan estos operadores son las siguientes:

Y

A	B	A && B
False	False	False
True	False	False
False	True	False
True	True	True

O

A	B	A    B
False	False	False
True	False	True
False	True	True
True	True	True

NO

A	!A
False	True
True	False

Pero ¿qué pasa si tenemos algo como esto?

`a && b || c && !d`

¿Qué operadores se evalúan primero?

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 2 - Quinta vuelta de espiral: condicionales

- Actualizamos nuestra tabla de la prioridad de los operadores añadiendo todos los nuevos:

	Descripción	Operadores
+ PRIORITARIO	paréntesis	()
	operadores incremento/decremento	op++ op--
	operadores unarios	+op -op !
	multiplicación y división	* /
	suma y resta	+ -
	operadores relacionales	< > <= >= == !=
	AND booleano	&&
	OR booleano	
- PRIORITARIO	operador de asignación	=

Realizar los ejercicios 6 al 10 del boletín

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 2 - Quinta vuelta de espiral: condicionales

### ► Tips and Tricks (consejos y trucos) cuando trabajamos con condiciones:

- Cuando se encadenan varios `&&` seguidos, basta que 1 operando sea falso para poder afirmar que toda la expresión será falsa. Ejemplo:
  - `if (a < 3 && casado == true && 1 < 0) → false`
- Cuando se encadenan varios `||` seguidos, basta que 1 operando sea cierto para poder afirmar que toda la expresión será cierta. Ejemplo:
  - Ej: `if (a < 3 || casado == true || 1 > 0) → true`
- IMPORTANTE, el operador `&&` (AND) es más prioritario que `||` (OR). Observa las siguientes frases:
  - *“Te daré la paga si recoges la cocina y ordenas tu cuarto o sacas la basura o paseas al perro.”*
    - `if (cocinaRecogida == true && cuartoOK == true || basuraFuera == true || perroPaseo == true)`
    - La condición completa sería cierta tan solo con que saque la basura o bien pasee al perro.
  - *“Te daré la paga si recoges la cocina y ordenas tu cuarto y, o bien sacas la basura, o bien paseas al perro.”*
    - `if (cocinaRecogida == true && cuartoOK == true && (basuraFuera == true || perroPaseo == true))`
    - En este caso debería recoger la cocina Y ordenar el cuarto Y, además, hacer al menos una de las dos tareas restantes (o sacar basura o pasear al perro)



# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 2 - Quinta vuelta de espiral: condicionales

- ▶ Cuando una variable o propiedad es de tipo **boolean** y se quiere preguntar en la condición si es igual a true o false se usa la siguiente notación abreviada:
  - ▶ `if ( cocinaRecogida )` es equivalente a `if(cocinaRecogida == true)`
  - ▶ `if ( !cocinaRecogida )` es equivalente a `if(cocinaRecogida == false)`
- ▶ La doble negación se convierte en afirmación.
  - ▶ Ejemplos: `!!true` → `true` o bien `!!false` → `false`
- ▶ Cuando tenemos una condición como `if ( ! ( z > 3 || b == 9 ) )` podemos buscar una expresión equivalente aplicando la “**regla de los operadores complementarios**”. Veamos un ejemplo:
  - ▶ `!(z > 3 || b==9)` es equivalente a `!(z > 3) && !(b == 9)` que equivale a su vez a `z <= 3 && b != 9`
    - ▶ Por ejemplo, con `z == 1` y `b == 1` el resultado de ambas expresiones sería `true`.
  - ▶ La regla dice que el operador `!` pasa a aplicarse a los elementos dentro de los paréntesis alterando cada operador por su “complementario”, es decir:
    - ▶ `&&` pasa a `||` y viceversa
    - ▶ `==` pasa a `!=` y viceversa
    - ▶ `<` pasa a `>=` y viceversa
    - ▶ `>` pasa a `<=` y viceversa

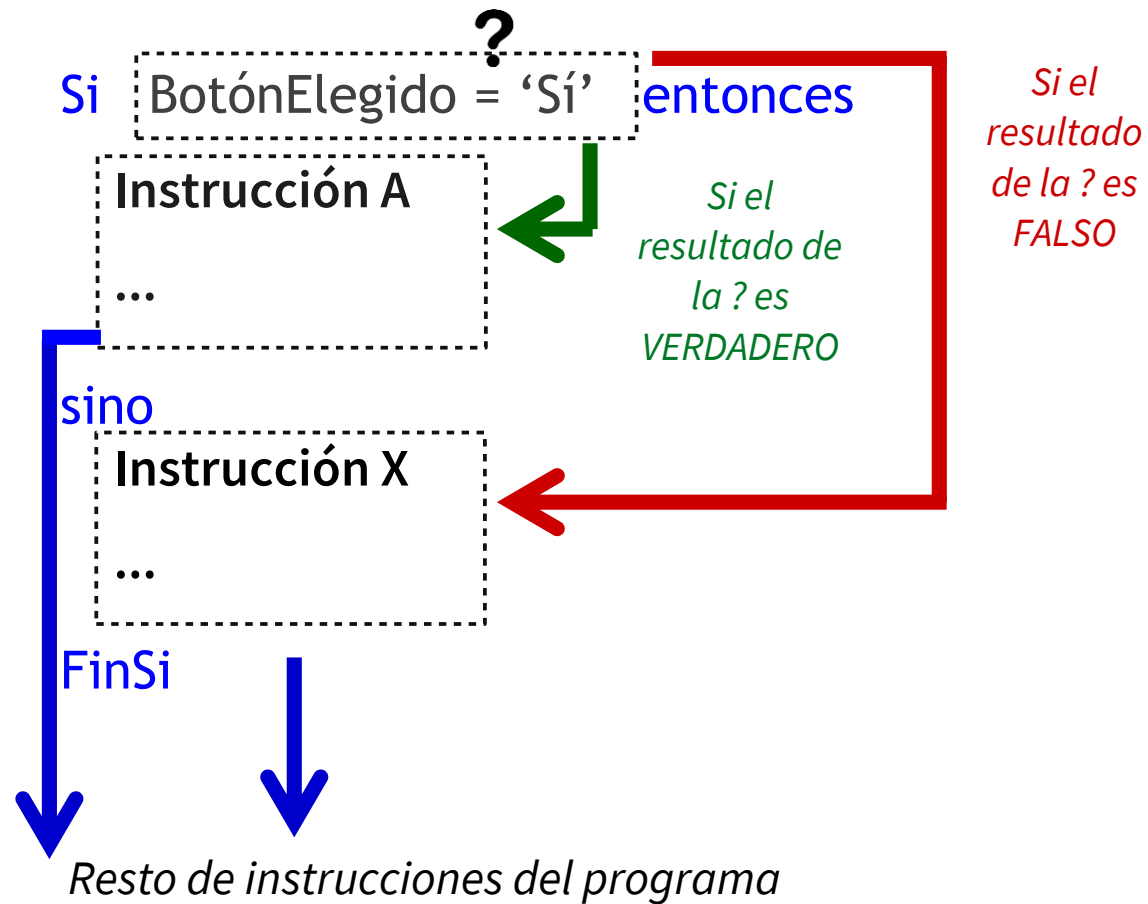




# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 2 - Quinta vuelta de espiral: condicionales

**Condicional doble:** es una variante de la condicional simple. La idea es la siguiente:



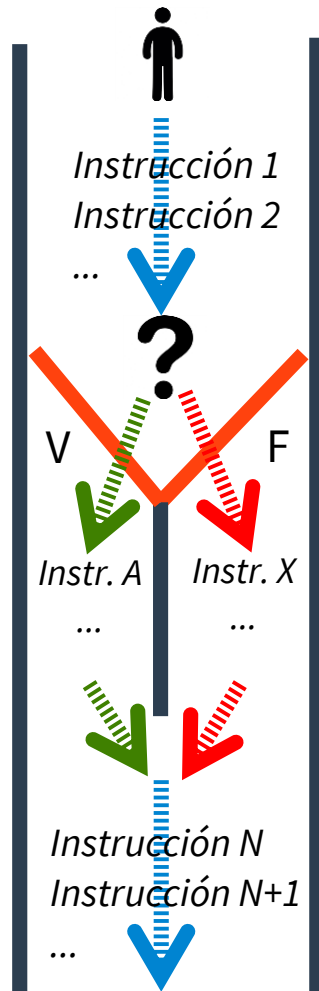
Dependiendo del resultado de la pregunta el programa escoge un camino u otro:

- Si BotónElegido = 'Sí' es VERDADERO entonces se ejecutará el PRIMER bloque de instrucciones interno y después se continuará con las instrucciones del resto del programa.
- Si BotónElegido = 'Sí' es FALSO entonces se ejecutará el SEGUNDO bloque de instrucciones interno y después se continuará con las instrucciones del resto del programa.

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 2 - Quinta vuelta de espiral: condicionales

- Si volvemos al símil del “*pasillo de instrucciones*” la nueva instrucción sería así:



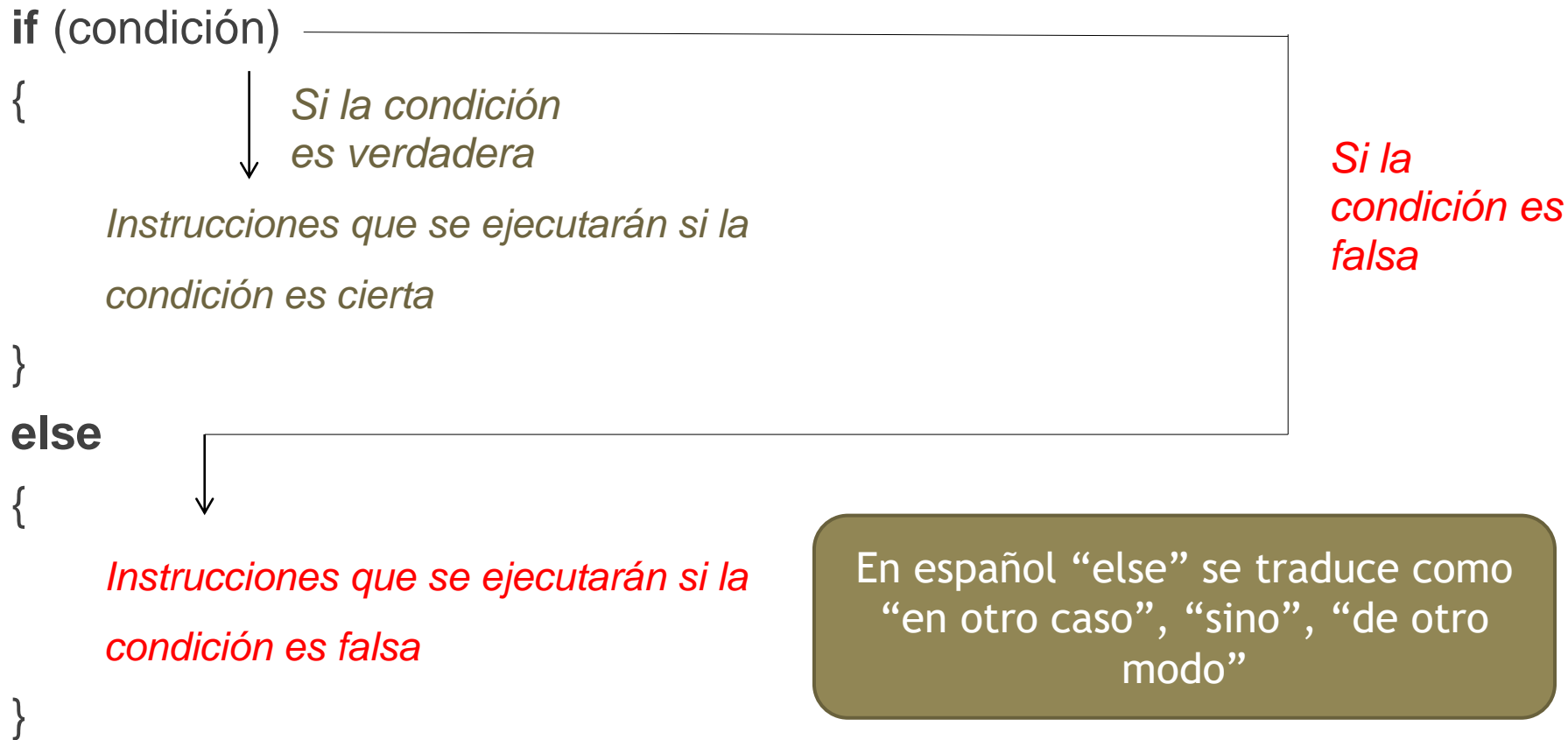
Nuestro programa va por el pasillo ejecutando las instrucciones que se encuentra. De pronto llega a dos puertas cerradas y se le hace una pregunta:

- Si el resultado de la pregunta es VERDADERO entonces pasará a la PRIMERA habitación y continuará ejecutando las instrucciones que allí se encuentre. Después saldrá de la habitación y continuará su camino por el pasillo.
- Si el resultado de la pregunta es FALSO entonces pasará a la SEGUNDA habitación y continuará ejecutando las instrucciones que allí se encuentre. Después saldrá de la habitación y continuará su camino por el pasillo.

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 2 - Quinta vuelta de espiral: condicionales

La **sintaxis** en Java de la condicional doble es:



# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 2 - Quinta vuelta de espiral: condicionales

### Ejemplo:

```
System.out.print("La bombilla está... ");  
if (encendida)  
{  
    System.out.println("encendida");  
}  
else  
{  
    System.out.println("apagada");  
}  
System.out.print("El programa sigue ejecutándose");
```

Si encendida == true es verdadero

La bombilla está... encendida  
El programa sigue ejecutándose

Si encendida == false es falso

La bombilla está... apagada  
El programa sigue ejecutándose

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 2 - Quinta vuelta de espiral: condicionales

En el caso anterior se usa 1 sola instrucción tanto en el “cuerpo” del if como en el “cuerpo” del else.



```
System.out.print("La bombilla está... ");  
if (encendida)  
{  
    System.out.println("encendida"); // 1 instruc.  
}  
else  
{  
    System.out.println("apagada"); // 1 instruc.  
}  
System.out.print("El programa sigue ejecutándose");
```



Solo en el caso de que un “cuerpo” tenga una sola instrucción podemos quitar las llaves. Sin embargo, no se considera buena práctica porque **puede inducir a errores futuros añadiendo más instrucciones debajo...**



```
System.out.print("La bombilla está... ");  
if (encendida)  
    System.out.println("encendida"); // 1 instruc.  
else  
    System.out.println("apagada"); // 1 instruc.  
System.out.print("El programa sigue ejecutándose");
```

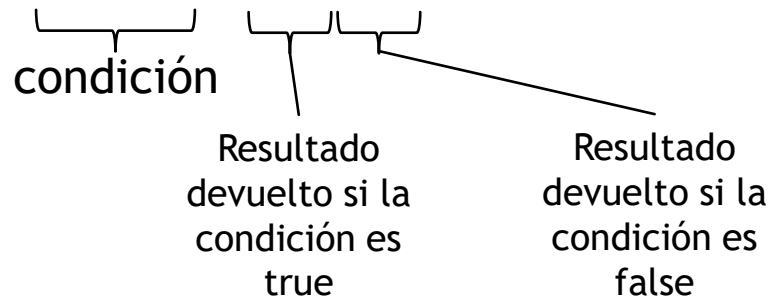


# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 2 - Quinta vuelta de espiral: condicionales

Cuando nos encontramos casos en los que en el “cuerpo” del if y del else solo hay una instrucción y esa instrucción es una asignación de una valor a la misma variable se suele usar la siguiente notación abreviada:

`elMayor = (x > y) ? x : y;`



Es equivalente a:

```
if (x > y)
    elMayor = x;
else
    elMayor = y;
```

El operador `?` afecta a tres operandos: la condición y los dos resultados posibles. Por eso, al `?` se le llama el **operador ternario condicional**.



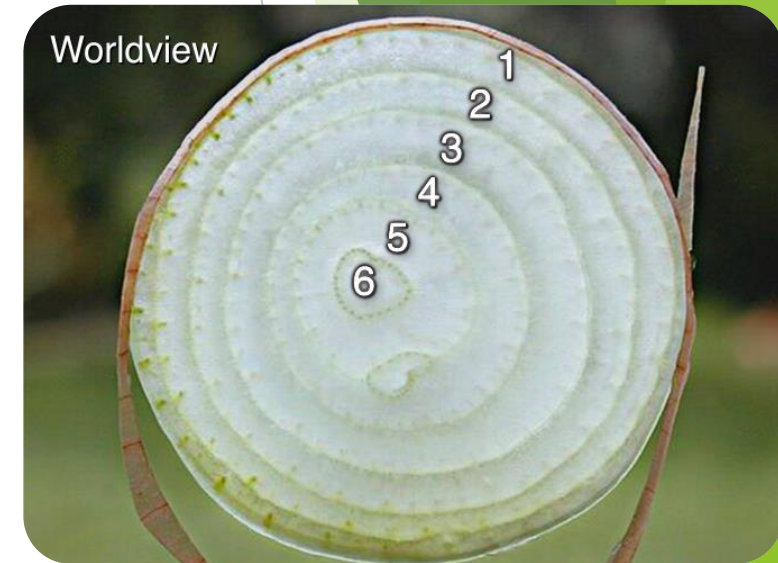
# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 2 - Quinta vuelta de espiral: condicionales

### Condicionales anidadas (nested)

Para rizar el rizo, son condicionales unas dentro de otras. Ejemplo:

```
if (edad < 18)
{
    System.out.println("Es menor de edad");
}
else
{
    if (tieneCarneConducir) {
        System.out.println("Es mayor de edad y puede conducir");
    }
    else {
        System.out.println("Es mayor de edad pero no puede conducir");
    }
}
```



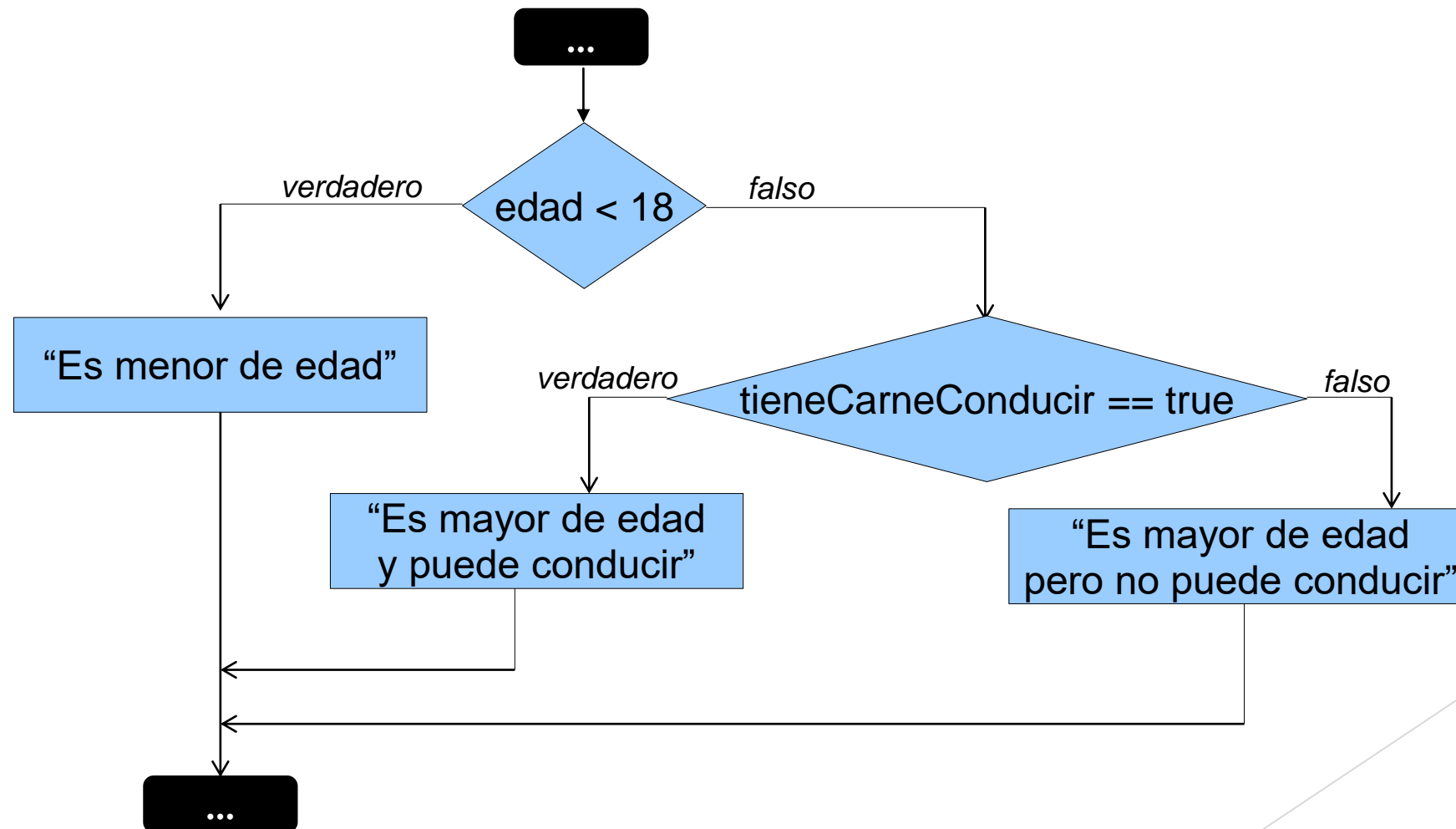
Anidamos condicionales como las capas de una cebolla



## UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

### 2 - Quinta vuelta de espiral: condicionales

- El ejemplo anterior se puede describir también mediante un diagrama de flujo como el siguiente:



# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 2 - Quinta vuelta de espiral: condicionales

- ▶ Cuando usamos sentencias condicionales anidadas es importantísimo aplicar **indentación** al código para que se vea qué partes contienen a otras.

Con Alt + Mayusc + F  
NetBeans formatea el  
código por nosotros

*Con indentación*

```
if (edad < 18)
{
    System.out.println("Es menor de edad");
}
else
{
    if (tieneCarnetConducir == true) {
        System.out.println("Es mayor de edad y puede conducir");
    }
    else {
        System.out.println("Es mayor de edad pero no puede conducir");
    }
}
```

*Sin indentación*

```
if (edad < 18)
{
    System.out.println("Es menor de edad");
}
else
{
    if (tieneCarnetConducir == true) {
        System.out.println("Es mayor de edad y puede conducir");
    }
    else {
        System.out.println("Es mayor de edad pero no puede conducir");
    }
}
```

Realiza los ejercicios del 11 al 13 del boletín de problemas

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 2 - Quinta vuelta de espiral: condicionales

### DE LA CABEZA AL CÓDIGO

- ▶ Cada vez vamos a tener más instrucciones en nuestra “paleta” y más formas de combinarlas.
- ▶ Si queremos aprender a programar no podemos afrontar un desarrollo mediante “ensayo y error”, a ver si “milagrosamente” damos con la combinación de instrucciones correcta.
- ▶ Necesitamos una metodología, es decir, una secuencia de tareas que nos vaya acercando a la solución del problema. Para ello, vamos a usar el documento DE LA CABEZA AL CÓDIGO.

LEEMOS EL DOCUMENTO “DE LA CABEZA AL CÓDIGO” Y EMPEZAMOS A APLICARLO EN LOS PRÓXIMOS EJERCICIOS

Realizamos entre todos el ejercicio 14 aplicando esta metodología



```
1 package Seleccio1;
2 import java.util.Scanner;
3 /**
4  * @author Kris
5  */
6
7 public class Seleccio1 {
8     // TODO code application logic here
9
10    int num;
11    Scanner ingreso=new Scanner(System.in);
12    System.out.print("Ingrese Número del 1 - 12 : ");
13    num = Integer.parseInt(ingreso.next());
14    switch (num) {
15        case 1: System.out.println("ENERO"); break;
16        case 2: System.out.println("FEBRERO"); break;
17        case 3: System.out.println("MARZO"); break;
18        case 4: System.out.println("ABRIL"); break;
19        case 5: System.out.println("MAYO"); break;
20        case 6: System.out.println("JUNIO"); break;
21        case 7: System.out.println("JULIO"); break;
22        case 8: System.out.println("AGOSTO"); break;
23        case 9: System.out.println("SEPTIEMBRE"); break;
24        case 10: System.out.println("OCTUBRE"); break;
25        case 11: System.out.println("NOVIEMBRE"); break;
26        case 12: System.out.println("DICIEMBRE"); break;
27        default : System.out.println("NÚMERO DEL MES INCORRECTO");
28    }
29 }
30 }
```

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 2 - Quinta vuelta de espiral: condicionales

### Formato “preferido” para la escritura del else if

Cuando el **else** va **seguido solo de un if** entonces se suele formatear el código de esta forma que es más compacta y se lee mejor:

```
if (edad < 18) {  
    System.out.println("Es menor de edad");  
}  
  
else if (tieneCarneConducir) {  
    System.out.println("Es mayor de edad y puede conducir");  
}  
  
else {  
    System.out.println("Es mayor de edad pero no puede conducir");  
}
```

**Versión anterior**

```
if (edad < 18)  
{  
    System.out.println("Es menor de edad");  
}  
else  
{  
    if (tieneCarneConducir == true) {  
        System.out.println("Es mayor de edad y puede conducir");  
    }  
    else {  
        System.out.println("Es mayor de edad pero no puede conducir");  
    }  
}
```

Conseguimos trabajar con menos { }  
y menos niveles de indentación

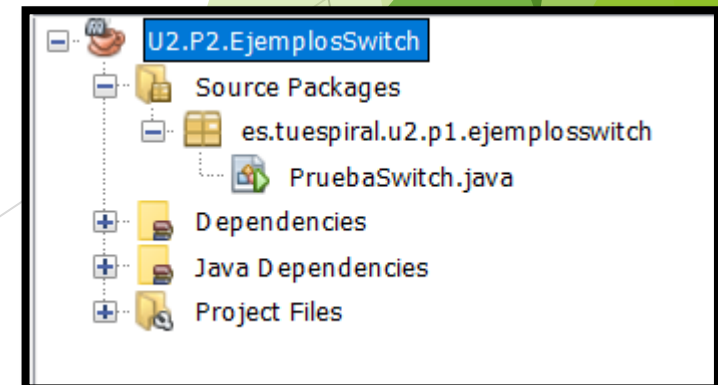
También es muy común poner la {  
de apertura en la misma línea que  
el if o el else

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 2 - Quinta vuelta de espiral: condicionales

### La condicional múltiple o de selección - switch.

- ▶ Funciona de una forma parecida al **selector del programa de una lavadora**.
  - ▶ Cuando pones una lavadora en el 5, empezará a realizar las tareas del programa 5, después continuará con el 6, 7... hasta llegar al último número.
  - ▶ Para que no pase por todos los números debes parar (break) manualmente la lavadora.
- ▶ El switch **no** utilizará una pregunta de tipo booleano (true/false), sino que se usará una **expresión** de tipo **entero** o **cadena de texto** que se corresponda con el “programa de inicio de la lavadora”.
- ▶ Veamos el código del proyecto **U2.P2.EjemplosSwitch** que está en el repositorio de código.

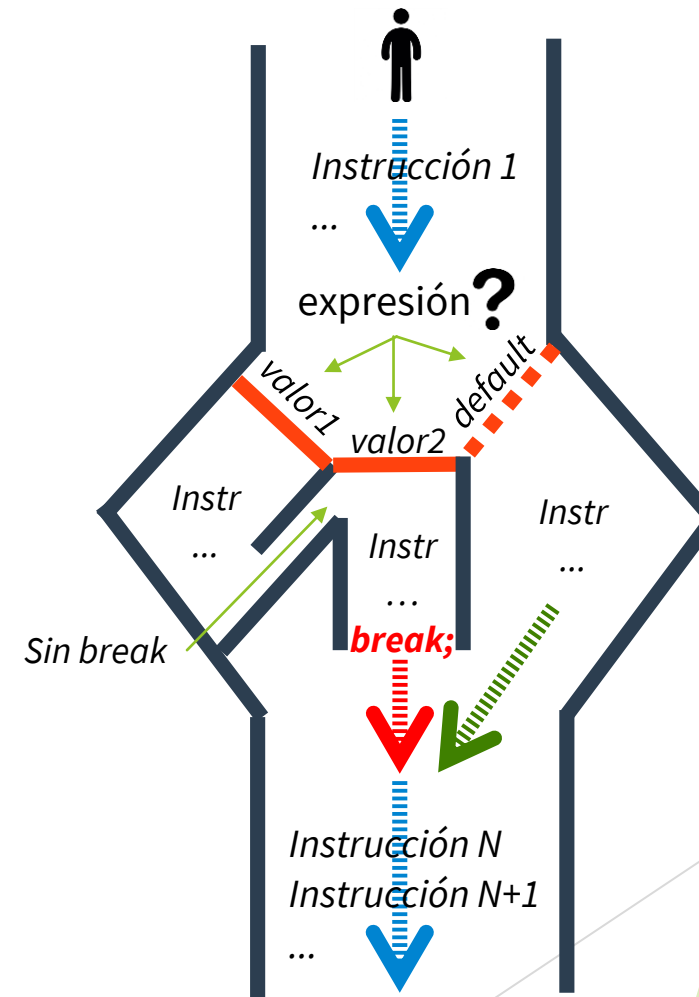


# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 2 - Quinta vuelta de espiral: condicionales

- La sintaxis y nuestro “símil” con el pasillo de instrucciones serían así:

```
switch( expresión_tipo_int_o_tipo_String ) {  
    case valor1:  
        instrucción1;  
        ...  
        instrucciónN;  
        break; // Opcional, dependiendo  
               // del efecto que se quiera conseguir  
    case valor2:  
        ...  
    default: // Opcional  
        ...  
}
```

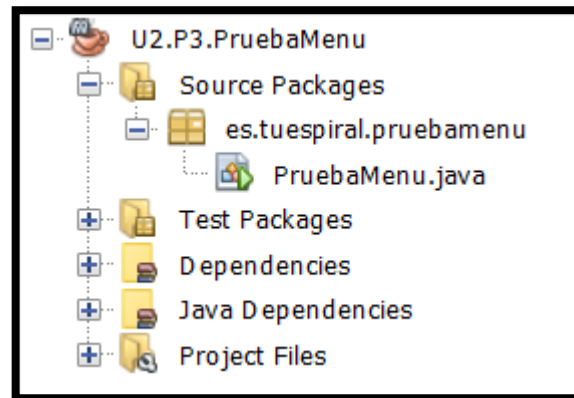


# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 2 - Quinta vuelta de espiral: condicionales

- El **switch** no se usa tanto como el **if** pero hay un caso en el que lo usaremos SIEMPRE: cuando hagamos **menús de opciones** para el usuario.

```
Dime un número entero (a):  
4  
Dime un número entero (b):  
5  
MENU DE OPCIONES:  
1 - Suma (a+b)  
2 - Multiplicación (a*b)  
3 - División entera (a/b)  
Escoge una opción:  
2  
El resultado es 20
```



Estudia el código de U2.P3.PruebaMenu que está en el repositorio

Estudia la chuleta de la 5ª vuelta a la espiral

Realiza los ejercicios del 15 al 18 del boletín de problemas

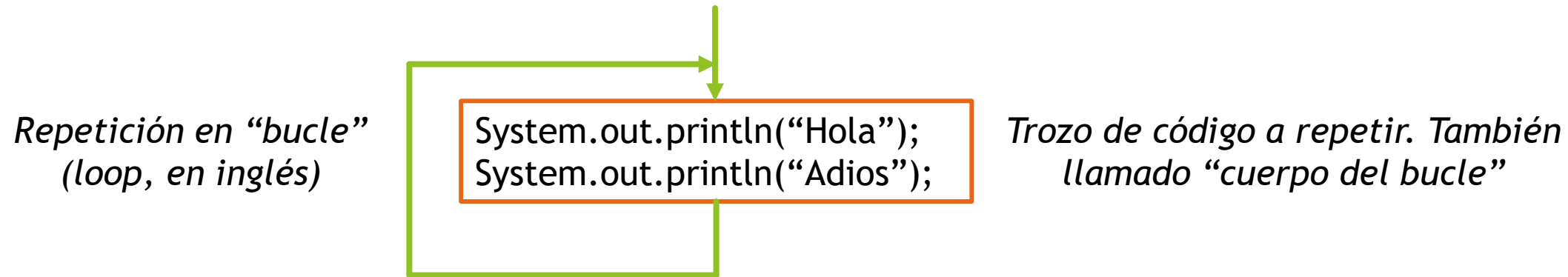


# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 3 - Sexta vuelta de espiral: repetitivas

### ► 3 - Sexta vuelta - Estructuras de control repetitivas

Ahora vamos a exprimir la potencia del procesador haciendo que ejecute **un mismo trozo de código** muchas veces para conseguir algún objetivo.



Cada vez que pasemos por ese trozo de código diremos que hemos dado **“una vuelta”** o **“una iteración”**.

**Pero y ¿cuándo terminamos?**

**¿Nos quedamos dando vueltas para siempre en un “bucle infinito”?**



## UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

### 3 - Sexta vuelta de espiral: repetitivas

Nos hace falta saber en **qué condiciones nos quedamos repitiendo** el trozo de código y cuándo dejamos de repetir.

Esta “**condición de permanencia**” hay que diseñarla con mucho cuidado para que el programa haga exactamente lo que queremos.

Dependiendo de si “conocemos de antemano” cuántas repeticiones tenemos que hacer, clasificamos las estructuras de control en dos grupos:

- ▶ **Tipo MIENTRAS (while, en inglés):** se usan normalmente cuando NO sabemos de antemano cuántas veces tenemos que repetir el trozo de código.
- ▶ **Tipo PARA (for, en inglés):** se usan cuando sabemos EXACTAMENTE cuántas veces tenemos que repetir el trozo de código.



while

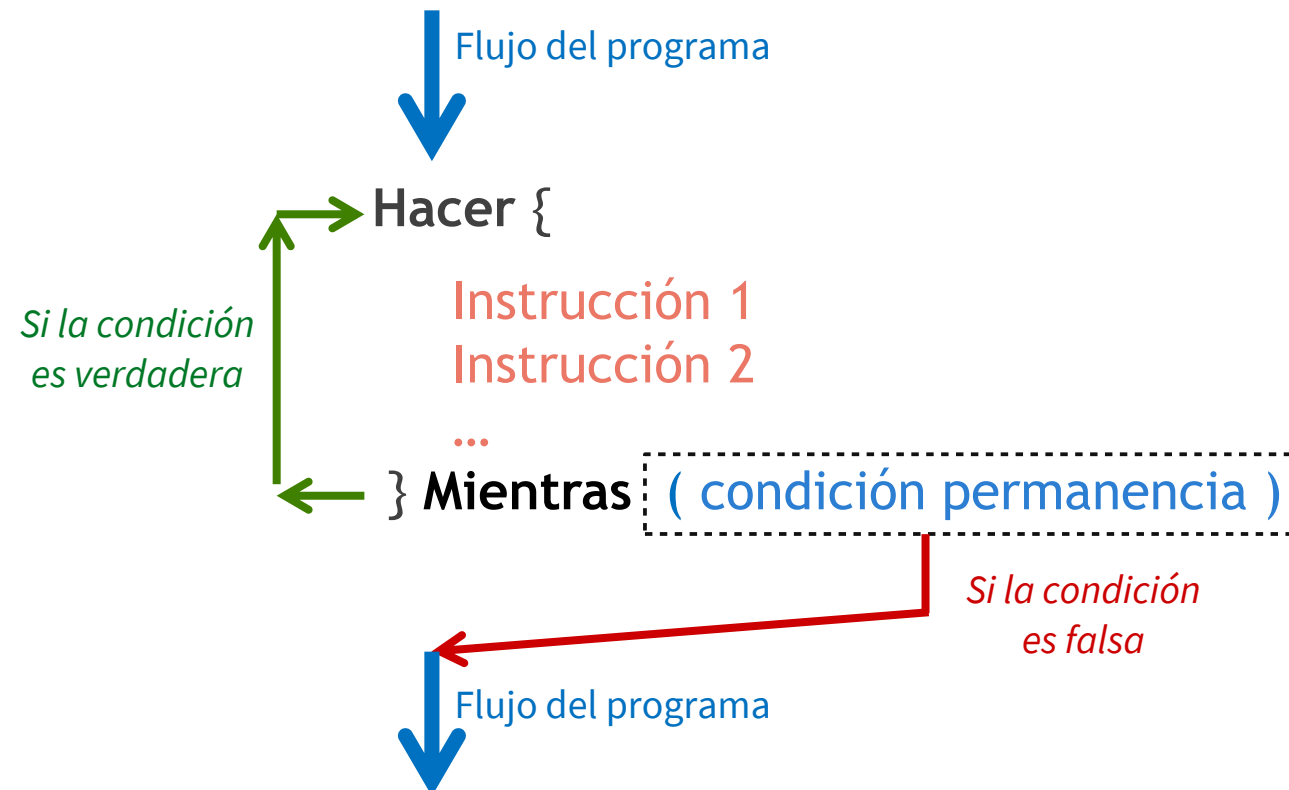
for

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 3 - Sexta vuelta de espiral: repetitivas

### Tipo HACER... MIENTRAS (do...while, en inglés)

- Responde a la frase: “**Repite este trozo de código mientras la condición de permanencia sea cierta**”.
- Queremos algo como:

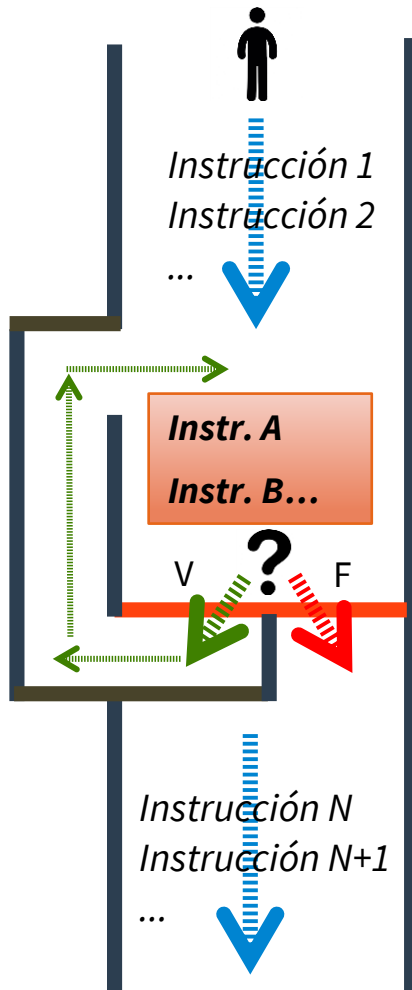


do...  
while

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 3 - Sexta vuelta de espiral: repetitivas

- Si volvemos al símil del “tío del pasillo” tendríamos:



Nuestro programa va por el pasillo ejecutando las instrucciones que se encuentra.

Ahora pasa por unas instrucciones (cuadro naranja) que terminan en una puerta con una pregunta:

- Si el resultado de la pregunta es VERDADERO entonces pasará al “pasillo secundario” y volverá a tener que realizar las instrucciones anteriores.
- Si el resultado de la pregunta es FALSO entonces volverá al camino del pasillo principal.

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 3 - Sexta vuelta de espiral: repetitivas

### ► Sintaxis en Java:

```
do {  
    // Instrucciones del cuerpo del bucle  
} while (condición) ; // Ojo: el ; se suele olvidar
```



Esta estructura se usa habitualmente cuando hay que **validar** lo que escribe el usuario

### ► Ejemplo:

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

```
int num;
```

```
do {
```

```
    System.out.println("Dime un número entre 1 y 10: ");
```

```
    num = sc.nextInt();
```

```
} while (num < 1 || num > 10) ;
```

Fíjate que al usuario se le pide que **num** cumpla:  
`num >= 1 && num <= 10`

Sin embargo, la condición de permanencia es justo la contraria:  
`!(num >= 1 && num <= 10)`

do...  
while

Realiza el ejercicio  
19 del boletín de  
problemas

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 3 - Sexta vuelta de espiral: repetitivas

- ▶ Cuando pensamos en la condición de permanencia, **es fácil confundir el “mientras” con el “hasta”**. Si pensamos en “modo hasta” estaremos haciendo la condición contraria a la que necesitamos.
- ▶ Si pensamos el ejemplo anterior en “modo hasta” caeríamos en una frase como “**Repito hasta que el número esté entre 1 y 10**”. Ejemplo:

```
Scanner sc = new Scanner(System.in);  
  
int num;  
  
do {  
    System.out.println("Dime un número entre 1 y 10: ");  
    num = sc.nextInt();  
  
} while (num >= 1 && num <= 10);
```

Es justo lo contrario de lo que queremos. Sólo saldríamos del bucle cuando el número NO esté entre 1 y 10



Hay lenguajes que tienen la estructura “hasta” (until). Pero en Java no existe.

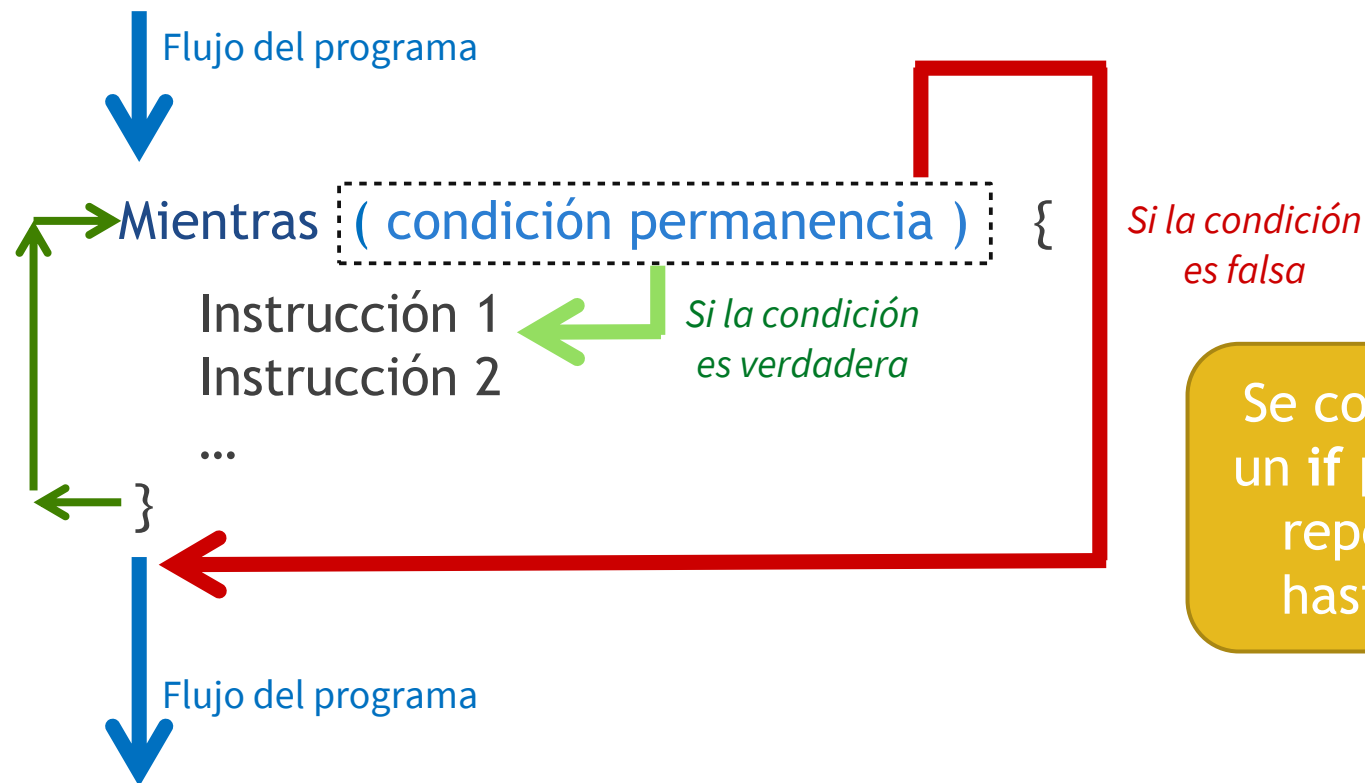
Realiza los ejercicios 20 y 21 del boletín de problemas

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 3 - Sexta vuelta de espiral: repetitivas

### Tipo MIENTRAS (while, en inglés)

- Responde a la frase: “mientras la **condición de permanencia sea cierta** **seguimos repitiendo** el trozo de código”.
- Queremos algo como:



Se comporta igual que un if pero que vuelve a repetir la pregunta hasta que sea falsa

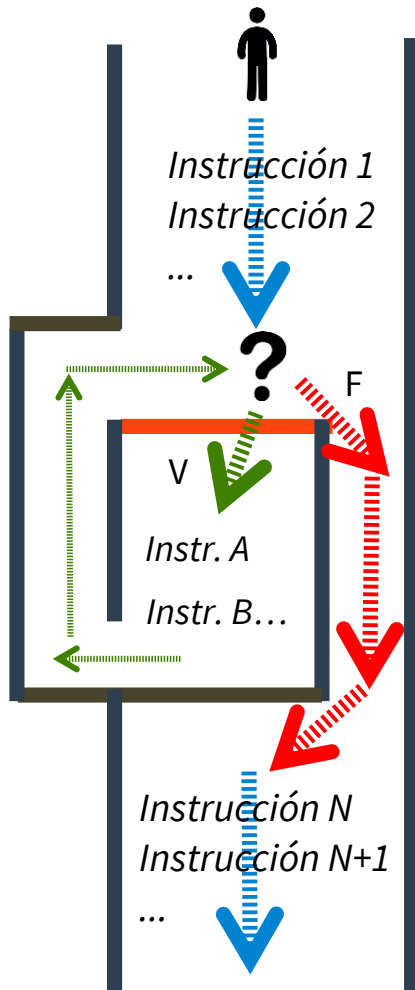
while



## UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

### 3 - Sexta vuelta de espiral: repetitivas

- Si volvemos al símil del “tío del pasillo” tendríamos:



Nuestro programa va por el pasillo ejecutando las instrucciones que se encuentra. De pronto llega a una puerta cerrada que le hace una pregunta, pudiendo pasar dos cosas:

- Si el resultado de la pregunta es VERDADERO entonces pasará a la habitación y continuará ejecutando las instrucciones que allí se encuentre. Después volverá al pasillo principal a través de otro pasillo “secundario. En el pasillo principal volverá a hacerse la pregunta de nuevo.
- Si el resultado de la pregunta es FALSO entonces no entrará en la habitación y continuará su camino por el pasillo principal.

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 3 - Sexta vuelta de espiral: repetitivas

### ► Sintaxis en Java:

```
while (condición) {  
    // Instrucciones del cuerpo del bucle  
}
```

Ahora preguntamos por la condición al principio. En el do... while, lo hacíamos al final

### ► Ejemplo:

```
System.out.println("Dime un número entre 1 y 10: ");
```

```
int num = sc.nextInt();
```

```
while (num < 1 || num > 10) {
```

```
    System.out.println("El "+num+" no está comprendido entre 1 y 10");
```

```
    System.out.println("Dime un número entre 1 y 10: ");
```

```
    num = sc.nextInt();
```

```
}
```

Fíjate que tenemos que preparar la entrada al while habiendo leído el valor que ponga el usuario.

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 3 - Sexta vuelta de espiral: repetitivas

### ► Consideraciones sobre los bucles **while** y **do...while**:

- **do...while** funciona muy bien cuando la condición de permanencia depende de alguna variable cuyo valor se concreta dentro del cuerpo del bucle y **necesitamos ejecutar el cuerpo del bucle al menos una vez**.
  - Cuando validamos la entrada de teclado de un usuario o hacemos un menú de opciones, **do...while** es la opción más adecuada.
  - Sin embargo, **do...while** parece que no le cae bien a la comunidad de programadores y se usa mucho más **while**, incluso en ocasiones donde un **do... while** hace mejor el trabajo
- **while** también se usa cuando sabemos exactamente cuántas repeticiones vamos a hacer.
  - En este caso, es mejor usar un bucle tipo **for** porque su sintaxis ayuda a que no te “olvides” de algún paso.



do...  
while

while

## UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

### 3 - Sexta vuelta de espiral: repetitivas

- **Uso de while cuando sabemos exactamente el número de repeticiones que va a hacer el bucle.**

Imagina que queremos imprimir 100 veces en pantalla la frase “Hola”.

Para no pasarnos ni quedarnos cortos, necesitamos “contar” de algún modo cuántas repeticiones llevamos.

Para ello, empleamos una “variable auxiliar” que actúa como **contador** y cuyo valor se corresponda con el número de “ejecuciones o vueltas” que le hemos dado al cuerpo del bucle.

Veamos cómo se hace:



## UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

### 3 - Sexta vuelta de espiral: repetitivas

...

```
int contador = 1;
```

```
while (contador <= 100) {
```

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

```
    contador++;
```

```
}
```

...

Iniciamos el contador

La condición de permanencia nos relaciona el contador con un límite

En el cuerpo del bucle modificamos el valor del contador de forma que nos vayamos acercando al límite de la condición de permanencia

Realiza los ejercicios del 22 al 31 del boletín de problemas

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 3 - Sexta vuelta de espiral: repetitivas

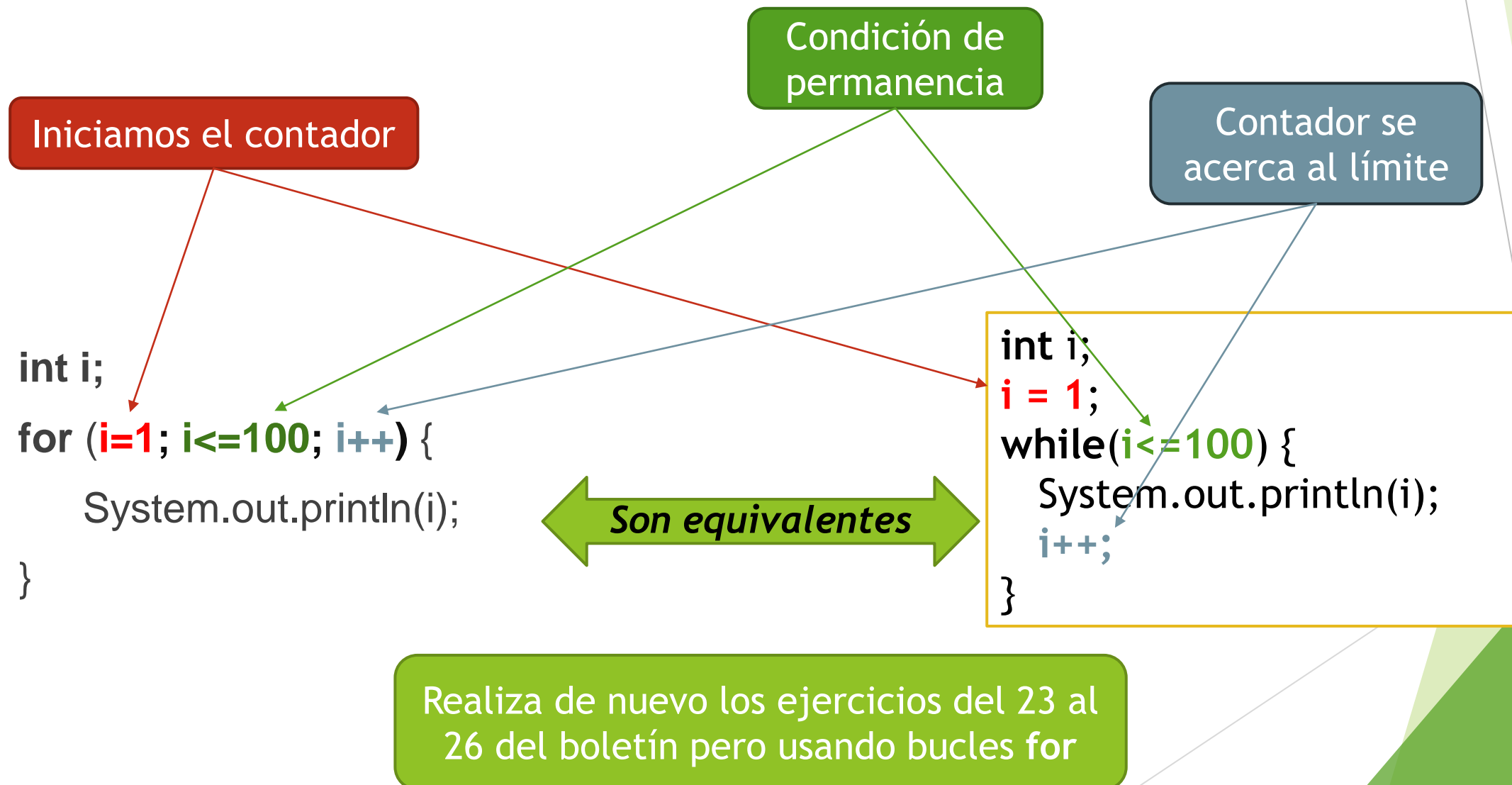
### Tipo PARA (for, en inglés)

- ▶ Llevamos varios ejercicios usando **while** usando una variable como **contador** para llevar la cuenta del número de repeticiones. Esto conlleva la realización de **tres acciones que “gestionan” el bucle**:
  - ▶ Iniciar el contador
  - ▶ Usar el contador en la condición de permanencia
  - ▶ Modificar el contador en el cuerpo del bucle para “acercarnos” a la salida del bucle y no caer en un bucle infinito
- ▶ La instrucción **for** nos obliga a escribir en una sola línea estas tres acciones. De este modo, es casi imposible que se nos olvide poner alguna de ellas.
- ▶ Usa **for** siempre que se pueda, es más “seguro” que usar **while** porque disminuye la probabilidad de que cometamos un error al programar.

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 3 - Sexta vuelta de espiral: repetitivas

► Veamos un ejemplo:





# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 3 - Sexta vuelta de espiral: repetitivas

### Bucles anidados

- ▶ Cuando ponemos un bucle dentro de otro, a los ojos del bucle externo, el bucle interno se comportará como una instrucción más.
- ▶ El siguiente código imprime las tablas de multiplicar del 1 al 10.

```
for (int num=1; num<=10; num++) {  
    System.out.println("La tabla del "+num+" es:");  
    for (int cont=0; cont <=10; cont++) {  
        System.out.println(num+" x "+cont+" = "+(num*cont));  
    }  
}
```

Escribe este código dentro de un *main* y depúralo poniendo watches a las variables *num* y *cont*



# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 3 - Sexta vuelta de espiral: repetitivas

### Romper un bucle con break

- ▶ Cuando vimos la sentencia **switch** poníamos un **break** al final de cada **case** para “romper el flujo descendente” dentro del **switch** y salir de él.
- ▶ Cuando usamos **break** dentro de un bucle hace exactamente lo mismo, se rompe el flujo natural del bucle y salimos de él.
- ▶ Veamos un ejemplo:

```
int numVecesSeis = 0;
Dado d = new Dado();
for (int i = 1; i <= 10; i++) {
    if (d.tirada() == 6)
        numVecesSeis++;
    if (numVecesSeis == 2)
        break;
}
```

El bucle está pensado para dar 10 vueltas pero si conseguimos un “seis doble” salimos inmediatamente del bucle

Realiza los ejercicios del 32 al 34 del boletín de problemas

#### Otras formas de hacer lo mismo:

- for (int i=1; i<=10 && numVecesSeis < 2; i++) en este caso se mete la cuenta del 6 doble en la condición de permanencia.
- for (int i=1; i<=10 && !seisDoble; i++) en este caso se pone una variable lógica a modo de “bandera” que nos indica si tenemos que cortar o no el bucle.

Cualquiera de las tres opciones es correcta.

Estudia la chuleta de la 6ª vuelta a la espiral

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 4 - Séptima vuelta de espiral: arrays

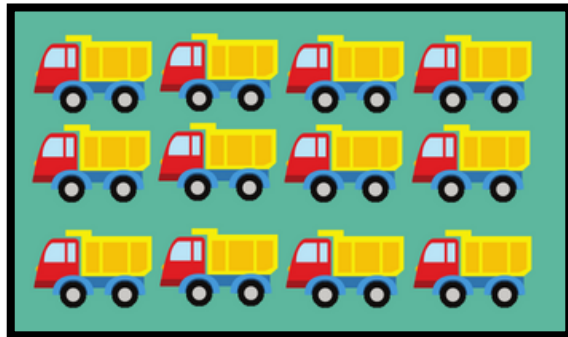
### ► 4 - Séptima vuelta - arrays

Imagina que tenemos que guardar las calificaciones del alumnado de una clase de 25 alumnos. ¿Definimos 25 variables? ¿nota1, nota2, ... nota25? ¿Y si ahora se matricula 1 alumno más? ¿Tendría que cambiar mi programa para añadir la nota26?

Necesitamos una **forma cómoda de manejar una “colección de variables” del mismo tipo.**

Los **arrays** son el elemento más simple del lenguaje que nos permite hacer esto.

La traducción al español de la palabra array es “**formación**”, en el sentido “militar” de la palabra.



Array de 3x4 camiones



Array de 4 camiones (1x4)



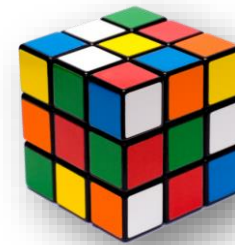
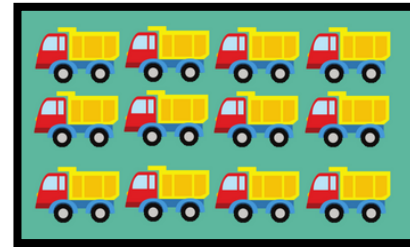
Array de NxM soldados

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 4 - Séptima vuelta de espiral: arrays

Según el tipo de “formación” hablamos de arrays:

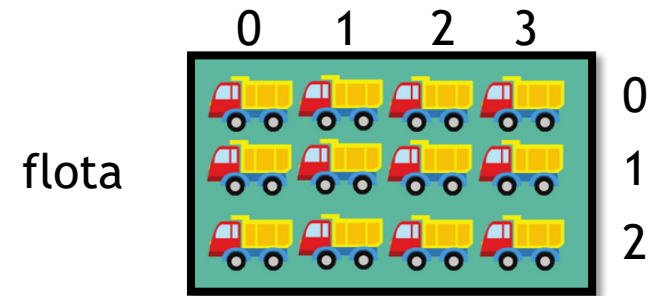
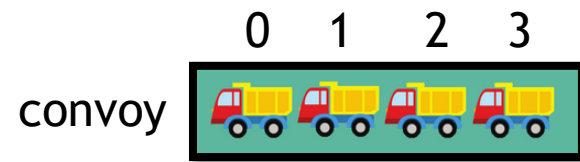
- ▶ **Unidimensionales:** tienen una sola dimensión.
  - ▶ Tienen forma de fila ( $1 \times N$ ) o de columna ( $N \times 1$ ).
  - ▶ También se les llama tablas o vectores.
- ▶ **Bidimensionales:** tienen dos dimensiones.
  - ▶ Tienen forma de **matriz** de N filas y M columnas ( $N \times M$ ).
  - ▶ Se les llama normalmente matrices.
- ▶ **Tridimensionales:** tienen tres dimensiones.
  - ▶ En forma de **cubo** (X, Y, Z)
- ▶ ...
- ▶ **N-dimensionales:** tiene n-dimensiones... Ya no se puede dibujar.



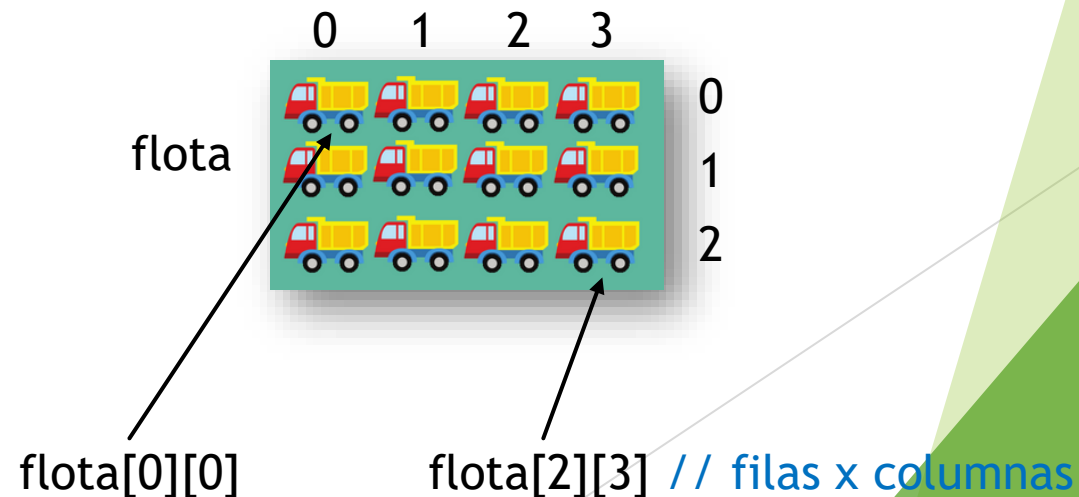
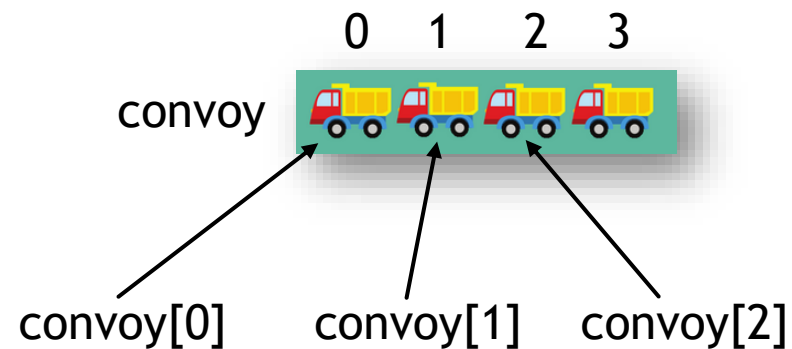
# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 4 - Séptima vuelta de espiral: arrays

La idea es **darle un nombre a la colección** de elementos y **manejarla con unos índices** que identifican la posición que ocupa un elemento de la colección:



De esta forma, para **referirme a un determinado elemento de la colección** solo tengo que usar el nombre de la colección y la posición que ocupa:



# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 4 - Séptima vuelta de espiral: arrays

En Java, un **array es una clase del lenguaje**, de modo que tendremos que usar el operador **new** para crear el objeto correspondiente.

Tiene una notación un tanto peculiar. Veamos un ejemplo:

```
int [] edades; // Declara una referencia capaz de apuntar a un array de int
edades = new int[10]; // Crea un objeto tipo array con capacidad para
                      // 10 int y lo asignamos a la referencia edades
```

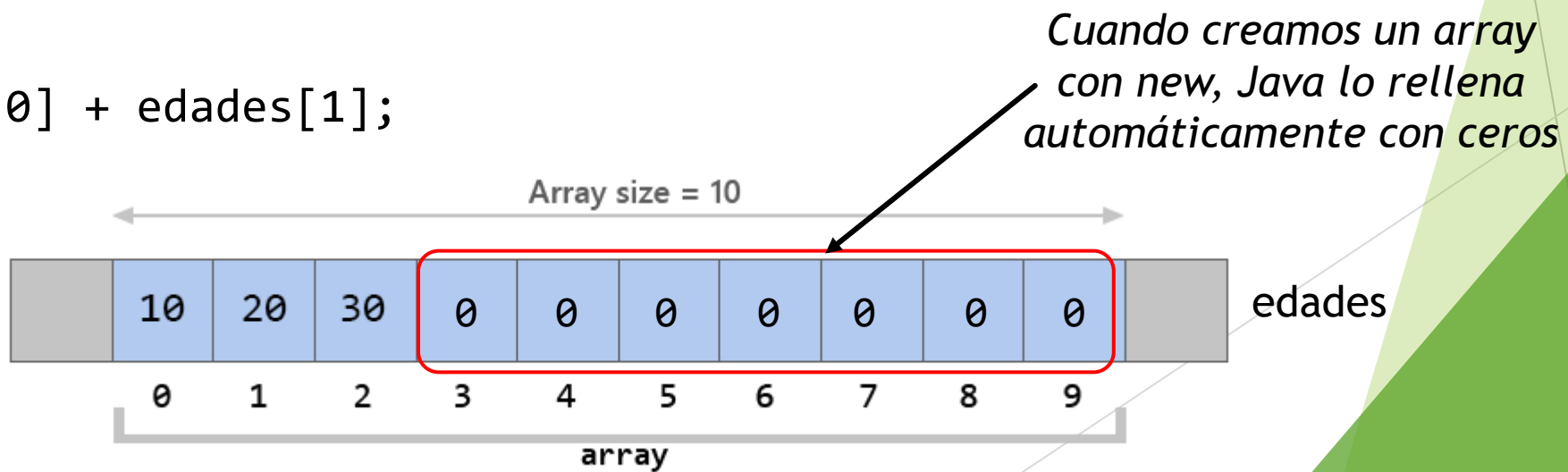
```
edades[0] = 10;
```

```
edades[1] = 20;
```

```
int suma = edades[0] + edades[1];
```

```
edades[2] = suma;
```

```
...
```

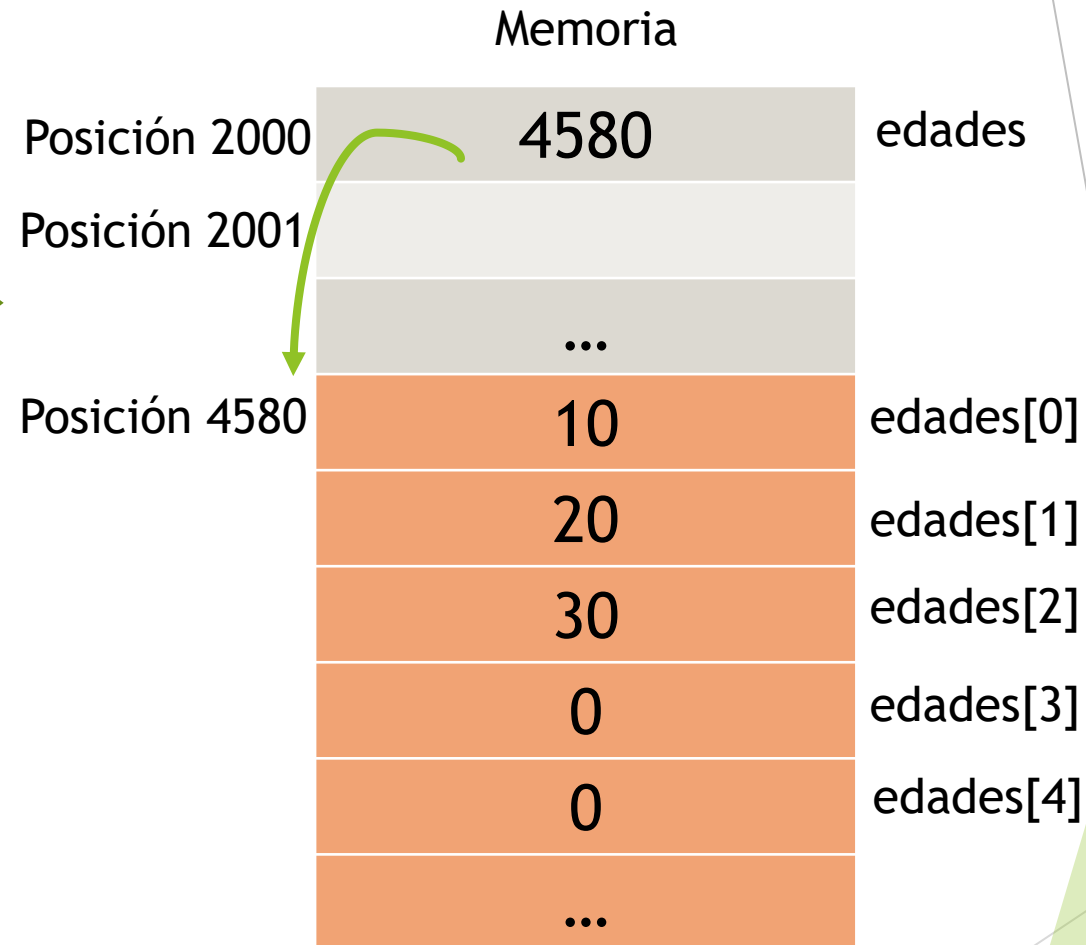
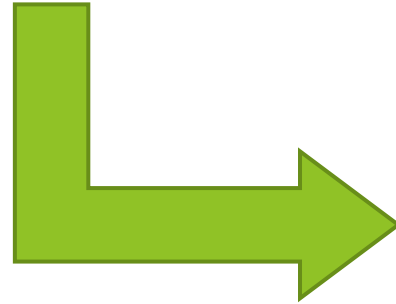




# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 4 - Séptima vuelta de espiral: arrays

Nuestro ejemplo, en la memoria quedaría algo así:



Por otro lado, hay **otra notación para declarar la referencia** a un array que es equivalente a la anterior:

```
int edades[];
```

es equivalente a:

```
int [] edades;
```

Las dos sintaxis son correctas y habituales.



# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 4 - Séptima vuelta de espiral: arrays

Normalmente, se usa la **declaración y la creación** del objeto **en la misma línea**:

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

Sabemos que cuando creamos un objeto, Java pone todas sus propiedades numéricas a cero, los booleanos a false y las referencias a objeto a **null**.

**¿Pero y si queremos darle otros valores de partida?** En este caso, tenemos la opción de **inicializar el array en el momento de su creación**.

La sintaxis usada es:

```
int[] enteros = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

Esto me crea un objeto array de 10 posiciones y le asigna los números que hay entre las llaves en las posiciones de la 0 a la 9.

Realiza el ejercicio 35 del boletín

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 4 - Séptima vuelta de espiral: arrays

### Array bidimensionales o matrices

Si queremos “modelar” un sudoku, podríamos utilizar un array de dos dimensiones para almacenar los números. Su declaración sería:

```
int[][] sudoku = new int[9][9]; // [filas][columnas]
```

```
sudoku[0][0] = 5;
```

```
sudoku[0][1] = 3;
```

```
sudoku[1][0] = 6;
```

...

*Filas* *Columnas*

*¿Qué instrucción escribirías para poner un 5 en la celda verde?*

	0	1	2	3	4	5	6	7	8
0	5	3			7				
1	6			1	9	5			
2		9	8					6	
3	8				6				3
4	4			8		3			1
5	7				2				6
6		6					2	8	
7				4	1	9			5
8					8			7	9

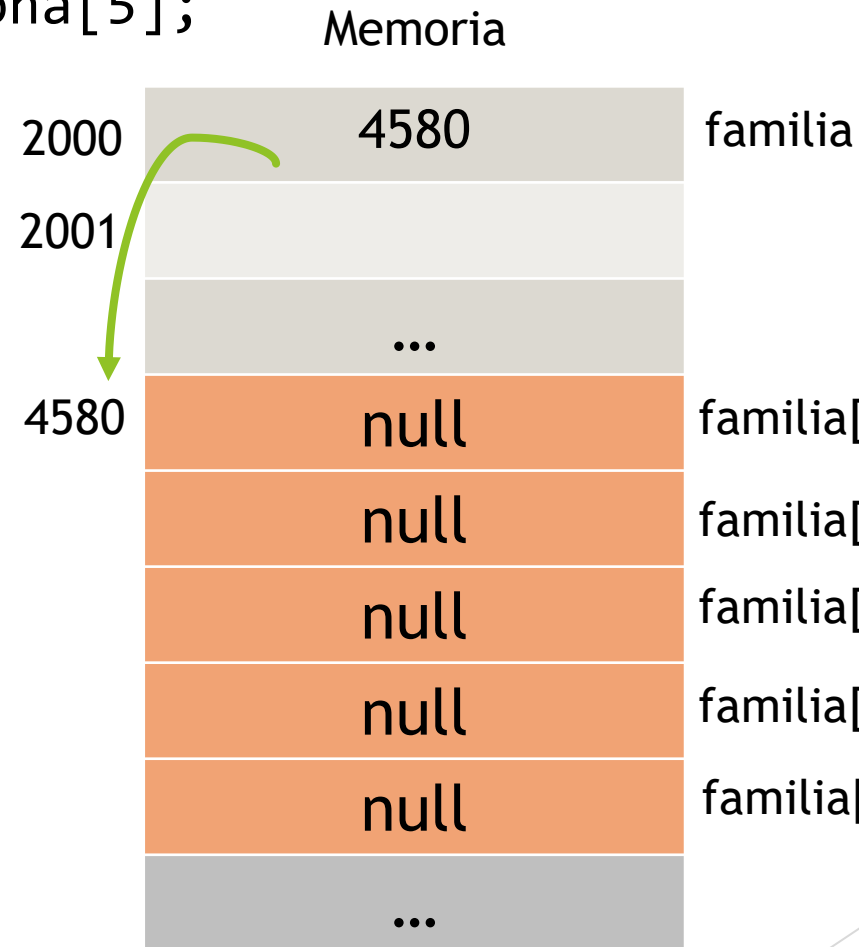
# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 4 - Séptima vuelta de espiral: arrays

### Arrays de referencias a objetos

También podemos crear un array de referencias a objetos del tipo que queramos:

```
Persona[] familia = new Persona[5];
```



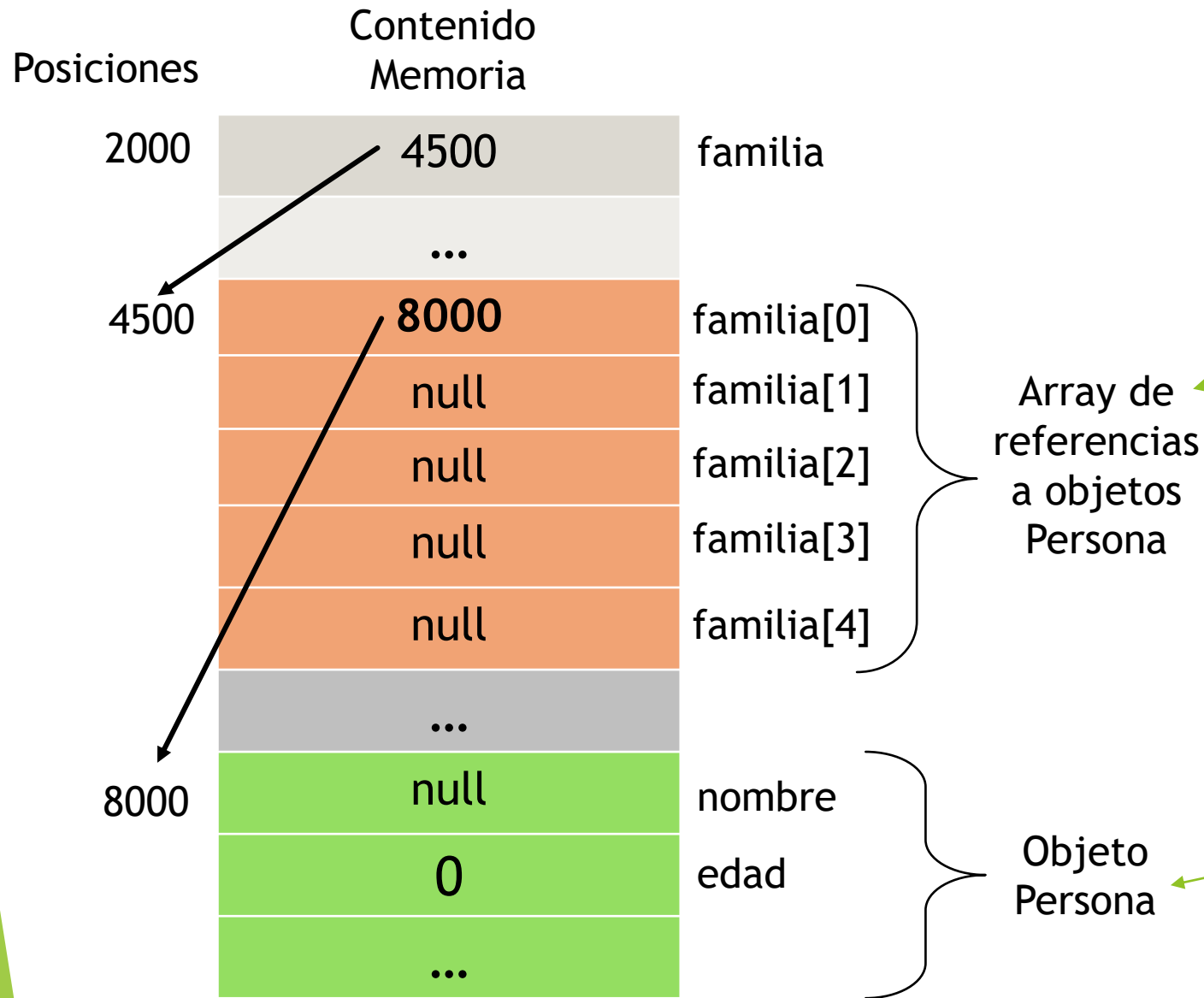
*¿Qué crees que pasa en la memoria cuando hacemos la siguiente instrucción?*

```
familia[0] = new Persona();
```

Objeto  
array de  
referencias  
a objetos  
Persona

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 4 - Séptima vuelta de espiral: arrays



Con la instrucción  
`Persona[] familia = new Persona[5];`  
se crea el objeto array

Con la instrucción  
`Familia[0] = new Persona();`  
se crea el objeto Persona

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 4 - Séptima vuelta de espiral: arrays

Ejemplos con referencias a **objetos String**:

```
String[] semana = {"L", "M", "X", "J", "V", "S", "D"};
```

```
System.out.println("El primer día de la semana es: "+semana[0]);
```

```
System.out.println("El último día es: "+semana[6]);
```

*/\* Si esta notación se te hace complicada, asigna el contenido de una posición del array a una variable auxiliar. Ejemplo: \*/*

```
String segundoDia = semana[1];
```

```
System.out.println("Y el segundo día es: "+segundoDia);
```



**Se imprime:**  
*El primer día de la semana es: L*  
*El último día es: D*  
*Y el segundo día es: M*

Los arrays de referencias a objetos tienen una **notación "engorrosa"**. Por el momento tenemos que seguir con ellos, en la unidad 4 aprenderemos soluciones mejores.

**Realiza el ejercicio 36 del boletín**

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 4 - Séptima vuelta de espiral: arrays

### Tratamiento de arrays usando bucles

Normalmente cuando tenemos una colección de elementos en un array vamos a tener que realizar tres acciones en repetidas ocasiones:

- ▶ **Recorrer todos los elementos** de un array para usarlos en alguna operación o aplicarles algún tratamiento.
- ▶ **Buscar un elemento concreto** del array o algún elemento que cumpla una condición.
- ▶ **Ordenar todos los elementos** del array según un criterio de orden determinado

Para los 3 casos anteriores **necesitamos conocer el tamaño del array a tratar**. Para ello los arrays tienen una **propiedad pública** que almacena la “longitud” (length) o número de posiciones del array y que podemos acceder mediante la sintaxis: **nombreArray.length**

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 4 - Séptima vuelta de espiral: arrays

### ► Recorrer un array

La idea es pasar por todos los elementos del array para hacerles algún tratamiento. Una posible estructura es:

```
for (int i = 0; i < nombreArray.length; i++) { // OJO: es < y no <=
    // Tratamiento de cada elemento
}
```

Ejemplo:

```
int[] pares = new int[100];
for (int i=0; i< pares.length; i++) {
    pares[i] = i*2;
}
```

Guarda en el array los 100  
primeros números pares  
empezando en 0

Realiza los ejercicios 37  
al 39 del boletín



# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 4 - Séptima vuelta de espiral: arrays

### Sintaxis alternativa: for (each) – “para cada elemento”

Cuando queremos **recorrer los elementos sin modificar el contenido de la colección** se usa una sintaxis alternativa **más cómoda y natural**. Veamos un ejemplo:

```
int[] pares = new int[100];

for (int i=0; i< pares.length; i++) {
    pares[i] = i*2;
}
```

```
for(int elem: pares) {
    System.out.println(elem);
}
```

La lectura de la instrucción sería:  
“**Para cada elemento** del array *pares* haces lo siguiente:  
1) Lo guardas en la **variable auxiliar elem**  
2) Y haces lo que se indique en el cuerpo del bucle”

En otros lenguajes esta sintaxis se corresponde con la instrucción **foreach**

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 4 - Séptima vuelta de espiral: arrays

### ► Ventajas de esta sintaxis:

- La gestión del bucle se hace automáticamente, no tenemos que manejar el índice (menos errores humanos).
- Volcar el elemento del array en una variable auxiliar nos permite una escritura más cómoda del cuerpo del bucle, ya que usamos **elem** en vez de **pares[i]**, que es más incómoda.
- También se puede usar para “buscar” un elemento determinado en una colección usando un **break** para salir del bucle en cuanto lo encontremos.

### ► Inconvenientes de esta sintaxis:

- El hecho de que se utilice la variable auxiliar **impide que podamos modificar el contenido del array**. Así que solo sirve para “leer” de la colección y no para “escribir” en ella.

Realiza el ejercicio 40 del boletín

Tras hacer estos ejercicios ve al final del tema y lee con atención el Anexo I sobre la clase String

## UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

### 4 - Séptima vuelta de espiral: arrays

#### ► Buscar un elemento en un array

La idea es pasar por todos los elementos del array **mientras no encontremos el elemento buscado, saliendo del bucle en cuanto lo encontramos.**

Normalmente se busca **un elemento que cumpla una condición.**  
Por ejemplo: *“El objeto persona con DNI 43123534K”* o *“Un número mayor que 100”* o *“La primera persona que encuentre cuyo nombre sea Juan”*

► **OJO:** si buscamos **“*TODAS* las personas cuyo nombre sea Juan”** ya **no se trata de una búsqueda sino un recorrido**, porque siempre tendríamos que comprobar todos los elementos de la colección”.

Otras veces no nos interesa el elemento como tal sino **la posición que ocupa en el array.**



# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 4 - Séptima vuelta de espiral: arrays

Al llevar todo esto al código tenemos que tomar decisiones sobre:

- ▶ **¿Qué tipo de bucle usamos?** ¿usamos un while, for, do...while?...
- ▶ **¿Cómo salimos del bucle si encontramos el elemento?** ¿usamos un break o creamos una condición del tipo “mientras no encontrado”?
- ▶ **¿Qué pasa si no encuentro el elemento?** ¿Qué devuelvo? ¿Un error, un elemento “vacío”, un **null**...?

A la última pregunta normalmente se le da una respuesta “a nivel de proyecto” para que todas las personas del equipo de desarrollo sigan un mismo criterio.

Además, la búsqueda de un elemento habitualmente se ofrece como un **método de una clase** que contiene como propiedad el array de elementos.

**Veamos ejemplos con DISTINTAS COMBINACIONES...**



## UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

### 4 - Séptima vuelta de espiral: arrays

**Buscar la posición de un número determinado en un array:**

```
public int buscaNumero (int numBuscado) {  
    int pos = -1;  
    for (int i=0; i < array.length; i++) {  
        int elem = array[i];  
        if (elem == numBuscado) {  
            pos = i;  
            break;  
        }  
    }  
    return pos;  
}
```

Decisiones tomadas:

- Usamos un bucle for
- Salimos con un break
- Si no se encuentra el elemento buscado la posición devuelta será -1

Realiza los ejercicios 41 y 42 del boletín

## UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

### 4 - Séptima vuelta de espiral: arrays

**Variante del anterior pero usando un array de cadenas:**

```
public int buscaNombre (String nombreBuscado) {  
    int pos = -1;  
    for (int i=0; i < array.length; i++) {  
        String nombreActual = array[i];  
        if(nombreActual.equals(nombreBuscado)) {  
            pos = i;  
            break;  
        }  
    }  
    return pos;  
}
```

Para comparar cadenas no se usa == sino equals. Consulta el Anexo I al final de este documento.

Decisiones tomadas:

- Usamos un bucle for
- Salimos con un break
- Si no se encuentra el elemento buscado la posición devuelta será -1

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 4 - Séptima vuelta de espiral: arrays

**Otra versión con otras decisiones:**

```
public int buscaNombre (String nombreBuscado) {  
    int pos = -1;  
    for (int i=0; i < array.length && pos==-1; i++) {  
        String nombreActual = array[i];  
        if(nombreActual.equals(nombreBuscado))  
            pos = i;  
    }  
    return pos;  
}
```

Decisiones tomadas:

- Usamos un bucle for
- Salimos usando una condición de permanencia tipo “mientras no encontrado”
- Posición -1, si no se encuentra

Realiza el ejercicio  
43 del boletín



# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 4 - Séptima vuelta de espiral: arrays

### Otra versión más:

```
public int buscaPorDNI (String dniBuscado) {  
    int pos = -1;  
    boolean encontrado = false;  
    int i = 0;  
    while (i < arrayPersona.length && !encontrado) {  
        String dni = arrayPersona[i].getDni();  
        if(dni.equals(dniBuscado)) {  
            pos = i;  
            encontrado = true;  
        }  
        i++;  
    }  
    return pos;  
}
```

#### Decisiones tomadas:

- Usamos un bucle while
- Salimos usando una condición de permanencia tipo “mientras no encontrado” pero usando una variable auxiliar booleana
- Posición -1, si no se encuentra

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 4 - Séptima vuelta de espiral: arrays

### Devolviendo un objeto (no una posición):

```
public Persona buscaPorDNI (String dniBuscado) {  
    Persona persEncontrada = null;  
    for (Persona elem: arrayPersona) {  
        String dni = elem.getDni();  
        if(dni.equals(dniBuscado)) {  
            persEncontrada = elem;  
            break;  
        }  
    }  
    return persEncontrada;  
}
```

#### Decisiones tomadas:

- Usamos un bucle for (each)
- Salimos con un break
- Si no se encuentra el elemento buscado devolvemos null

## UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

### 4 - Séptima vuelta de espiral: arrays

**Otra variante cambiando la marca de “elemento no encontrado”:**

```
public Persona buscaPorDNI (String dniBuscado) {  
    Persona persEncontrada = new Persona();  
    for (Persona elem: arrayPersona) {  
        String dni = elem.getDni();  
        if(dni.equals(dniBuscado)) {  
            persEncontrada = elem;  
            break;  
        }  
    }  
    return persEncontrada;  
}
```

Decisiones tomadas:

- Usamos un bucle for (each)
- Salimos con un break
- Si no se encuentra el elemento buscado devolvemos un objeto Persona “recién creado” que actuaría como marca de “elemento no encontrado”. Con este tipo de decisión, tendríamos que garantizar que en el array no se puede guardar nuestra marca de “elemento no encontrado”.

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 4 - Séptima vuelta de espiral: arrays

**Como vemos hay muchas variantes a la hora de buscar...**

Sin embargo, considero que las siguientes combinaciones son las que reducen las posibilidades de cometer errores:

- ▶ **Usa for o for each en vez de while** porque hace una gestión del bucle más automatizada y menos tendiente a errores.
- ▶ **Usa un break para salir** del bucle, es más limpio que complicar la condición del bucle añadiendo la parte de “mientras no encontrado”
- ▶ **Usa -1 como marca de no encontrado**, si lo que estamos buscando la posición de un elemento.
- ▶ **Usa un objeto creado pero “inicializado con valores vacíos” como marca de no encontrado**, si lo que buscamos es un objeto. Devolver **null**, es una práctica habitual pero tiende a generar errores en tiempo de ejecución (NullPointerException)

Realiza el  
ejercicio 44  
boletín

NullPointerException!!  
Ohh not again

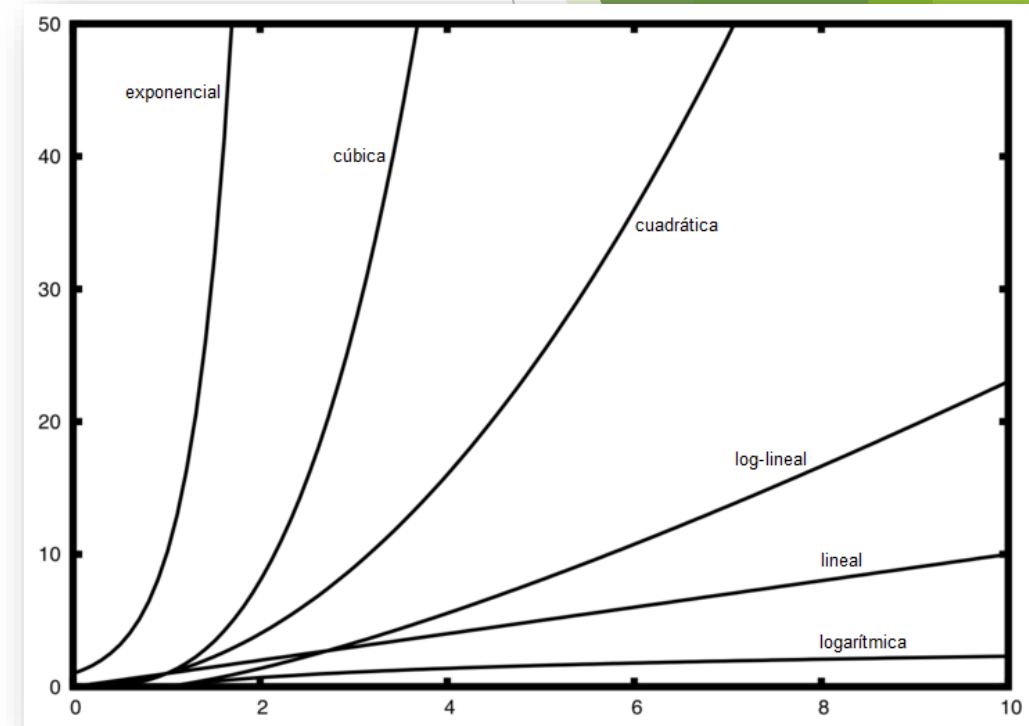


# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 4 - Séptima vuelta de espiral: arrays

### Buscar un elemento en un array ordenado.

- ▶ Cuando tenemos arrays desordenados en el peor de los casos tenemos que recorrer el array completo y no encontramos el elemento. A este tipo de búsqueda se le llama **búsqueda secuencial** y se dice, además, que su eficiencia es de “**Orden n**”, **O(n)** o **lineal**. Es decir para un array de **n** elementos, en el peor de los casos tenemos que tratar los **n** elementos.
- ▶ Digamos que para resolver un problema dado, podemos usar distintos algoritmos/soluciones y ordenarlas según su eficiencia, estableciendo un “Orden de magnitud” que representa el costo de cómputo del algoritmo aplicado.



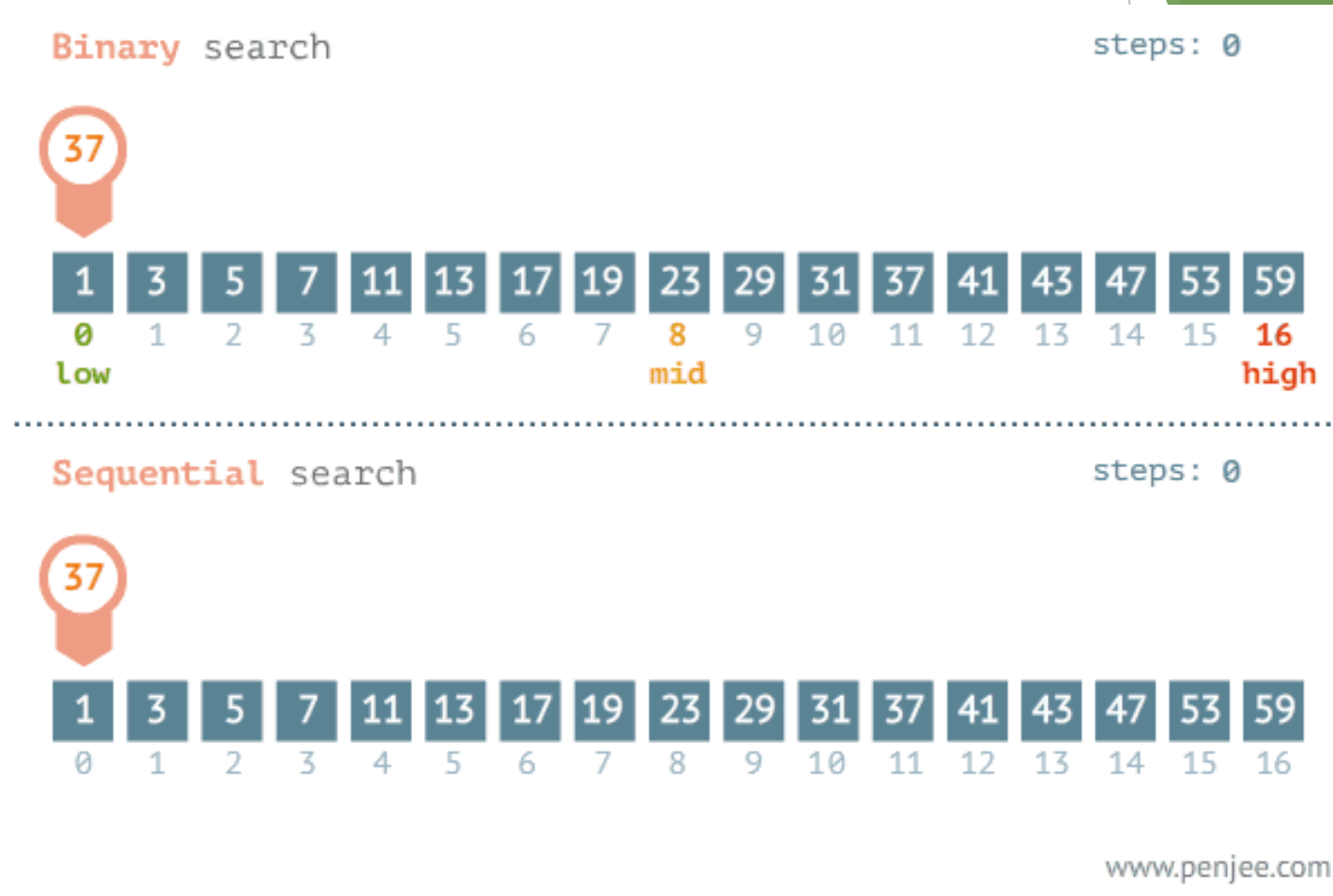
$$O(\log n) < O(n) < O(n^2) < O(n^3) < O(a^n)$$

## UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

### 4 - Séptima vuelta de espiral: arrays

- Cuando tenemos un array ordenado podemos utilizar el orden de los elementos para realizar una **búsqueda binaria** que es más eficiente, obteniendo un algoritmo de  **$O(\log n)$** . La lógica del algoritmo se muestra en el siguiente artículo:

<https://medium.com/@Emmitta/b%C3%BAsqueda-binaria-c6187323cd72>



No necesito que sepáis escribir este algoritmo (aunque es “bonito” y recomendable intentarlo). Solo quiero que sepáis que existe y que si tenéis una colección ordenada es el algoritmo de búsqueda que hay que usar.

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 4 - Séptima vuelta de espiral: arrays

### ► Ordenar los elementos de un array

Otro tratamiento habitual es ordenar los elementos de un array respetando un cierto criterio de orden.

Para resolver este problema también nos encontramos con distintos enfoques que se traducen en distintos algoritmos, cada uno con su “Orden de magnitud”.

En el siguiente vídeo se muestran distintas soluciones a los problemas de ordenación y búsqueda.

<https://www.youtube.com/watch?v=t-ige01xxEg>

En el siguiente artículo se muestra el código Java asociado a distintas técnicas de ordenación de arrays:

<https://www.discoduroderoer.es/formas-de-ordenar-un-array-en-java/>

```
Introduce la dimension del array: 5
Inserta el valor 1: 83
Inserta el valor 2: 13
Inserta el valor 3: 71
Inserta el valor 4: 4
Inserta el valor 5: 81
```

```
Array
83 13 71 4 81
```

```
Array ordenado
4 13 71 81 83
```

No necesito que sepáis escribir estos algoritmos.

Vamos a usar la clase estática `java.util.Arrays` para automatizar todas estas tareas.



# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 4 - Séptima vuelta de espiral: arrays

### ► La clase `java.util.Arrays`

Esta clase contiene un montón de métodos estáticos útiles para el trabajo con arrays. De modo que podremos fácilmente:

- Ordenar un array (*sort*)
- Comparar dos arrays (*equals*)
- Copiar un array en otro de distinto tamaño (*copyOf*)
- Buscar un elemento con búsqueda binaria (*binarySearch*)
- Obtener una representación de texto del array (*toString*)
- ...

Podemos ver la documentación oficial de la clase aquí:

- <https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html>

¿Qué es esto de **estático**?  
Ver ANEXO II al  
final de las  
diapositivas

Descarga el proyecto  
U2.P4.PruebaArrays del  
repositorio y estúdialo

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 4 - Séptima vuelta de espiral: arrays

### ANEXO I. La clase String

La clase String nos sirve para crear objetos de tipo texto y está integrada en el núcleo del lenguaje Java.

Como es muy usada, el lenguaje ofrece una notación simplificada para la creación de objetos. Basta con encerrar un texto entre comillas dobles y ya estaríamos creando un objeto String. Por ejemplo:

```
System.out.println("Hola");  equivale a  String s = new String("Hola");  
                                     System.out.println(s);
```

La clase contiene una **propiedad que es un array de caracteres (char)** pero que no es público, por tanto no puede ser accedido directamente, si no a través de los métodos disponibles.

#### String

- char[] texto
+ métodos

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 4 - Séptima vuelta de espiral: arrays

En el siguiente enlace tenemos la documentación oficial de la clase:

<https://docs.oracle.com/javase/9/docs/api/java/lang/String.html>

Veamos los métodos más usuales de la clase:

### MÉTODOS PARA COMPARAR DOS CADENAS:

- **public boolean equals(String s)** devuelve **true** si ambas cadenas son iguales lexicográficamente. Ejemplo:

```
String s1 = "Hola", s2 = "Hola", s3;
```

```
s3 = s1;
```

```
if (s1.equals(s2))
```

```
    System.out.println("Las cadenas tienen el mismo contenido");
```

```
if (s1 == s3)
```

```
    System.out.println("Las referencias apuntan al mismo objeto");
```

Devuelve **true** porque ambos objetos contienen exactamente el mismo texto

Devuelve **true** porque ambas referencias apuntan al mismo objeto, es decir, a la misma dirección de memoria.

Descarga el proyecto  
U2.DemoString.zip  
y analiza el código

## UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

### 4 - Séptima vuelta de espiral: arrays

- ▶ **public boolean equalsIgnoreCase(String s)** devuelve true si ambas cadenas contienen el mismo texto sin tener en cuenta las diferencias provocadas por las mayúsculas y minúsculas. Ejemplo:

String s = "Hola";

if (s.equalsIgnoreCase("hOLa")) devolvería **true**

- ▶ **public int compareTo (String s)** permite ordenar alfabéticamente dos cadenas, del siguiente modo:
  - ▶ devuelve 0 si ambas cadenas contienen el mismo texto
  - ▶ devuelve un entero  $< 0$  si el objeto desde el que se usa el método es menor que el parámetro de entrada
  - ▶ devuelve un entero  $> 0$  si el objeto desde el que se usa el método es mayor que el parámetro de entrada

Ejemplo:

## UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

### 4 - Séptima vuelta de espiral: arrays

```
String s1 = "Ana", s2 = "Jose";
```

```
if (s1.compareTo(s2) < 0) // Sería el equivalente a if (s1 < s2)
```

```
    System.out.println(s1+" es menor que "+s2);
```

```
if (s2.compareTo(s1) > 0) // Sería el equivalente a if (s2 > s1)
```

```
    System.out.println(s2+" es mayor que "+s1);
```

```
if (s2.compareTo(s1) != 0) // Sería el equivalente a if (s1 != s2)
```

```
    System.out.println(s2+" es distinta a "+s1);
```

- **public boolean compareToIgnoreCase(String s)** se comporta igual que la anterior pero ahora ignorando las diferencias provocadas entre mayúsculas y minúsculas.

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 4 - Séptima vuelta de espiral: arrays

### MÉTODOS PARA AVERIGUAR CIERTAS CARACTERÍSTICAS:

- ▶ **public int length()** devuelve la longitud de la cadena, es decir, el número de caracteres que tiene.
- ▶ **public boolean isEmpty()** devuelve true si la cadena es la cadena vacía ("")
- ▶ **public char charAt(int pos)** devuelve el carácter que hay en la posición *pos* del array. Las posiciones válidas serán, por tanto, desde la 0 a la *length() - 1*

### MÉTODOS PARA PASAR A MAYÚSCULAS/MINÚSCULAS:

- ▶ **public String toUpperCase()** y **public String toLowerCase()** devuelven una nueva cadena con todas las letras pasadas a mayúsculas o minúsculas, respectivamente.

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 4 - Séptima vuelta de espiral: arrays

Por último, es importante saber que **los objetos de la clase String son INMUTABLES**, es decir, una vez creado el objeto no lo podemos modificar (añadir, borrar o cambiar algún carácter).

**IMMUTABILITY**

- ▶ Algunos métodos de la clase String **dan la sensación** de que estamos modificando el contenido del objeto (por ejemplo: `toLowerCase()`). Sin embargo, lo que hacen es **devolver un nuevo objeto String** en el que se reflejan los cambios, pero nunca se altera el objeto original.
- ▶ Esto se hace así por motivos de rendimiento y de seguridad en entornos multihilo.

La “inmutabilidad” de los objetos String cumple con las necesidades de la mayoría de las aplicaciones. En los momentos puntuales en los que una aplicación tenga que hacer mucho trabajo de edición de texto (procesadores de textos, analizadores de documentos...) entonces se usa la clase **StringBuilder** que permite modificar el contenido de la cadena de texto sobre el mismo objeto.

**Class StringBuilder**



# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 4 - Séptima vuelta de espiral: arrays

### ANEXO II. Propiedades o métodos estáticos

Cuando usamos los siguientes elementos en nuestras expresiones estamos usando propiedades o métodos estáticos:

- ▶ Math.PI

- ▶ Es una propiedad estática definida dentro de la clase Math como:  
`public static final double PI = 3.1415...;`

- ▶ Math.random()

- ▶ Es un método estático definido dentro de la clase Math como:  
`public static double random() {...}`

- ▶ A efectos prácticos, cuando declaramos una propiedad o un método de una clase como **estáticos significa que podremos usarlos sin tener que crear un objeto de dicha clase.**



# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 4 - Séptima vuelta de espiral: arrays

Es decir, sería incorrecto hacer lo siguiente:

```
Math m = new Math();  
m.random();
```



El modificador **static** se usa en dos situaciones:

1. **Para tener una clase tipo “caja de herramientas”**: es decir una colección de propiedades o métodos que tienen poca conexión entre sí pero que sirven como "utilidades" a otras clases .
  - ▶ La clase **java.util.Math** tiene elementos estáticos poco relacionados entre sí (propiedades: PI, E y métodos: sin, cos, random, sqrt...) pero el conjunto sirve como “caja de herramientas matemáticas”.
  - ▶ A este tipo de clases en las que todos los elementos son estáticos se les llaman “clases degeneradas” porque rompen con el principio de la POO de que “Todo es un objeto”.
  - ▶ Esta es la **forma de uso más extendida** del modificador **static**.



# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 4 - Séptima vuelta de espiral: arrays

2. Para tener una propiedad de clase: en este caso queremos tener una **propiedad compartida** entre todos los objetos de una clase.

```
public class U2DemoStatic {  
  
    public static void main(String[] args) {  
        Bombilla b1 = new Bombilla();  
        Bombilla b2 = new Bombilla();  
  
        System.out.println("El consumo actual es " + Bombilla.getConsumoTotalBombillas() + " W");  
  
        b1.setPotencia(100);  
        b2.setPotencia(60);  
        b1.encender();  
        b2.encender();  
  
        System.out.println("El consumo actual es " + Bombilla.getConsumoTotalBombillas() + " W");  
    }  
}
```

El consumo actual es 0 W  
El consumo actual es 160 W  
BUILD SUCCESSFUL (total time: 0 seconds)

```
public class Bombilla {  
    // La propiedad estática consumoTotal será compartida entre todos  
    // los objetos. Es obligatorio inicializarla a la vez que se declara  
    public static int consumoTotal = 0;  
    public boolean encendida;  
    public int potencia;  
  
    public void encender() {  
        encendida = true;  
        consumoTotal = consumoTotal + this.potencia;  
    }  
  
    public void apagar() {  
        encendida = false;  
        consumoTotal = consumoTotal - this.potencia;  
    }  
  
    public static int getConsumoTotalBombillas() {  
        return consumoTotal;  
    }  
  
    public void setPotencia(int potencia) {  
        this.potencia = potencia;  
    }  
  
    public int getPotencia() {  
        return potencia;  
    }  
}
```

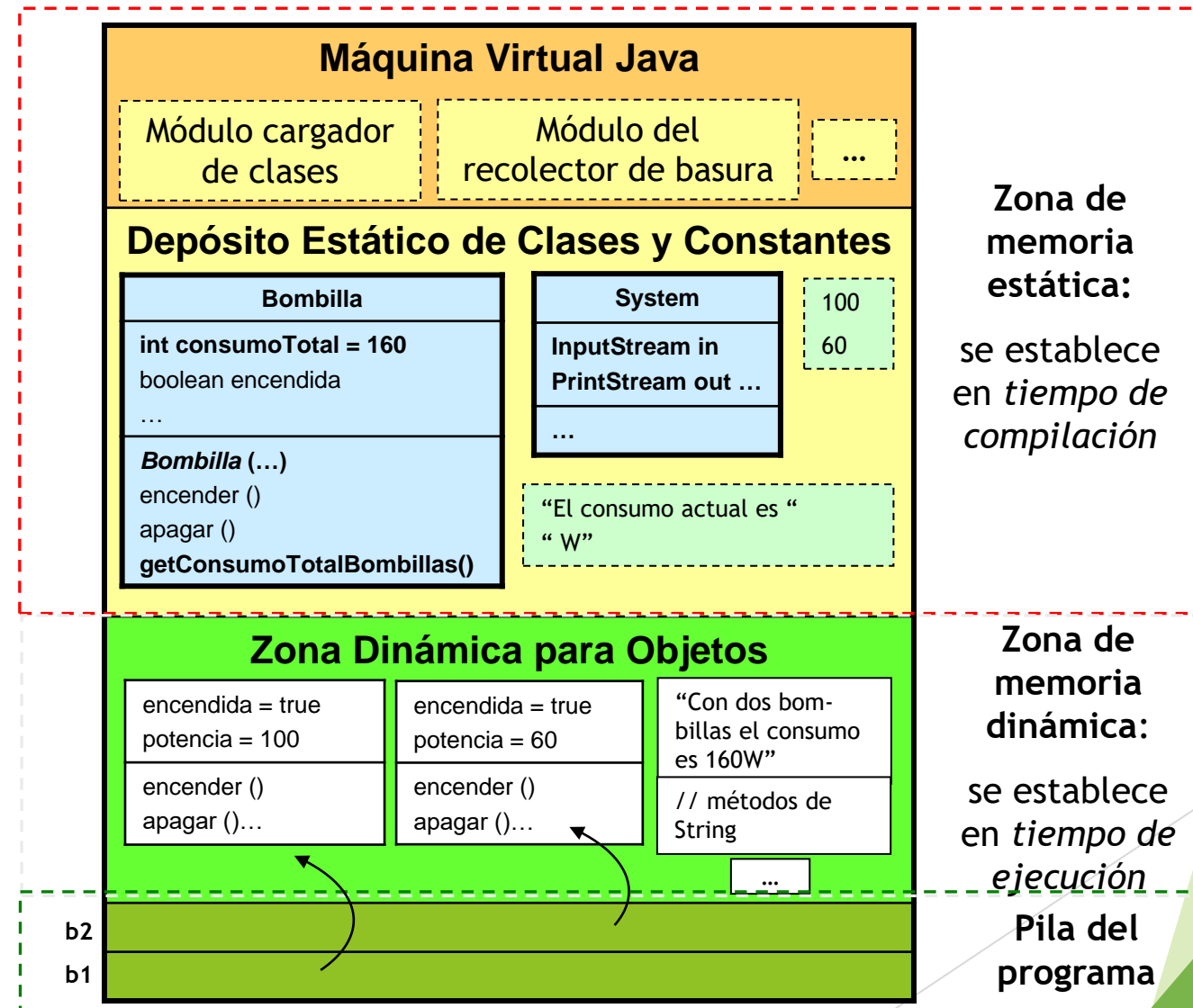
Descarga el proyecto  
U2.DemoStatic.zip y  
analiza el código

# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 4 - Séptima vuelta de espiral: arrays

Como dato "curioso":

- Las propiedades y los métodos **static** se **almacenan en la zona estática de la memoria**. De ahí viene el nombre.
- Java organiza la memoria como se expresa en el diagrama siguiente:



# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 4 - Séptima vuelta de espiral: arrays

A efectos prácticos en nuestro día a día en este módulo profesional:

- ▶ Usaremos **static** para usar o crear clases tipo “caja de herramientas”.
- ▶ Casi nunca usaremos **static** para compartir propiedades entre objetos de una clase.
- ▶ Usaremos **métodos estáticos** para hacer más "legible" nuestros **métodos *main***. Analiza el proyecto **U2.DemoStaticMain.zip**

- ▶ Ahora nuestro *main* se lee mejor con cosas como:

```
case 1:
    imprimirElementos(array);
    break;
```

- ▶ Observa como creamos un **método estático** que sirve como apoyo o auxilio al método *main*.

```
public static void imprimirElementos(int[] array) {
    for(int elem : array) {
        System.out.println(elem);
    }
}
```



Si se olvida poner static en el método dará el siguiente error...



# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

## 4 - Séptima vuelta de espiral: arrays

```
public void imprimirElementos(int[] array) {  
    for(int elem : array) {  
        System.out.println(elem);  
    }  
}
```



```
25         switch(opcion) {  
26             non-static method imprimirElementos(int[]) cannot be referenced from a static context  
27             ----  
28             (Alt-Enter shows hints)  
29             imprimirElementos(array);  
                break;
```

*Esto se produce porque main es un método static y solo puede invocar a métodos auxiliares que también sean static*

### Comparativa static vs non static

Elementos static	Elementos non static
No necesitan la creación de un objeto para poderse utilizar	Necesitan la creación de un objeto para poderse utilizar.
Una <b>propiedad estática</b> representa un valor compartido por todos los objetos.	Una <b>propiedad no estática</b> representa un valor que está ligado al objeto al que pertenece. Solo tiene sentido para el objeto que lo contiene.
Un <b>método estático</b> solo puede realizar acciones con las propiedades estáticas de la clase o bien con los parámetros de entrada pero NUNCA podría realizar cambios sobre propiedades NO estáticas	Un <b>método no estático</b> normalmente realiza una acción con las propiedades del objeto.



# UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.


## Cierre de la unidad

En esta unidad hemos aprendido **poner los “ladrillos”** de una aplicación que nos permitirán en un futuro “levantar el edificio”.

Comprender el ámbito de las variables, saber cuál es el tipo de una expresión, manejar con precisión los operadores booleanos, así como las estructuras condicionales y repetitivas son **requisitos absolutamente necesarios** para superar este módulo profesional. No dejes NADA sin comprender.

Los arrays son las colecciones de elementos más básicas de un lenguaje y serán reemplazados por otras estructuras más potentes más adelante, sin embargo, **tenemos que saber recorrerlos y hacer búsquedas sobre ellos.**

Aunque solo “hemos presentado” el manejo de String y de los elementos estáticos, **ambos conceptos se seguirán trabajando en las próximas unidades** y seguiremos profundizando en ellos.



Hay 2 tipos de personas en el mundo: **las que hacen que las cosas ocurran** y las que ven cómo ocurren las cosas.



## UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM.

The words "The End" are written in a large, white, 3D sans-serif font. The text is centered and appears to be floating above a series of overlapping, organic, cloud-like shapes in red, orange, yellow, and green. The background is white, with a decorative green geometric pattern on the right side.

The End