

```

import threading, socket, time
class sock(threading.Thread):
    def __init__(self):
        self.sck=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        threading.Thread.__init__(self)
        self.flag=1
    def connect(self, addr, port, func):
        try:
            self.sck.connect((addr, port))
            self.handle=self.sck
            self.todo=2
            self.func=func
            self.start()
        except:
            print "Error: could not connect"
    def listen(self, host, port, func):
        try:
            self.sck.bind((host, port))
            self.sck.listen(5)
            self.todo=1
            self.func=func
            self.start()
        except:
            print "Error: Could not bind"
    def run(self):
        while self.flag:
            if self.todo==1:
                x, ho=self.sck.accept()
                self.todo=2
                self.handle=x
            else:
                dat=self.handle.recv(4096)
                self.data=dat
                self.func()
    def send(self, data):
        self.handle.send(data)
    def close(self):
        self.flag=0
        self.sck.close()

```

DAM/DAW

PROGRAMACIÓN

04.1

Estructuras de control

I

Indice

- Resolución de problemas
- Estructuras de selección
 - La sentencia *if*
 - La construcción *if - else*
 - El operador condicional (*?:*)
 - La sentencia *switch*
- Estructuras iterativas
 - El bucle *while*
 - El bucle *do-while*
 - El bucle *for*
 - Las sentencias *break* y *continue*
- Estructuras y ámbito de las variables

```
import threading, socket, time
class sock(threading.Thread):
    def __init__(self):
        self.sock=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        threading.Thread.__init__(self)
        self.flag=1
    def connect(self, addr, port, func):
        try:
            self.sock.connect((addr, port))
            self.handle=self.sock
            self.todo=2
            self.func=func
            self.start()
        except:
            print "Error: Could not connect"
            return
    def listen(self, host, port, func):
        try:
            self.sock.bind((host, port))
            self.sock.listen(2)
            self.todo=1
            self.func=func
            self.start()
        except:
            print "Error: Could not bind"
    def run(self):
        while self.flag:
            if self.todo==1:
                x, ho=self.sock.accept()
                self.todo=2
                self.client=ho
                self.handle=x
            else:
                dat=self.handle.recv(4096)
                self.data=dat
                self.func()
            self.send(self.data)
            self.handle.send(data)
        self.close(self)
        self.close()
```

Resolución de problemas (I)

- Para crear un solución computacional de un problema, necesitamos :
 - entender en detalle el problema a resolver y su contexto
 - comprender los bloques de construcción disponibles para expresar la solución
- Cualquier problema computacional puede resolverse ejecutando una serie de acciones en un orden específico
- Denominamos **algoritmo** al *procedimiento* para resolver un problema en términos de:
 - las **acciones** a ejecutar, y
 - el **orden** en que se ejecutan esas acciones
- Se llama **control del programa** al proceso de especificar el orden en el que se ejecutan las instrucciones de un programa

Resolución de problemas (II)

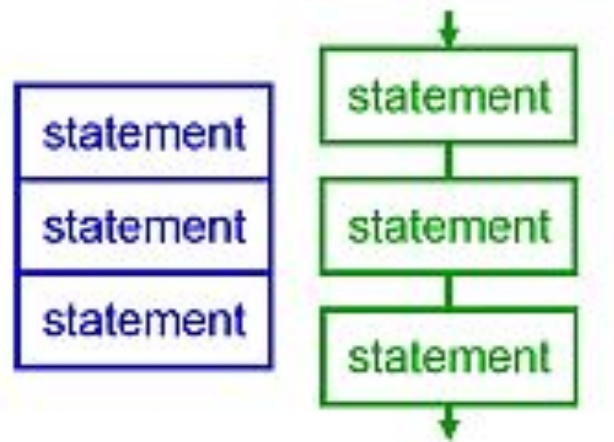
❑ Programación estructurada

- En general, un programa informático se ejecutará de forma **secuencial**, es decir, en el mismo orden que están escritas
- Sin embargo, desde la aparición de los primeros lenguajes máquina, se incorporan al lenguaje instrucciones que permitan “*romper*” este orden de ejecución y “*saltar*” a diferentes secciones del código. Esto se conoce como **transferencia de control**
- Ya durante la década de los 60 se observa como el uso indiscriminado de las *transferencias de control* (el “infame” *goto*), genera notables dificultades tanto en la escritura como el mantenimiento de los programas
- Los trabajos de *Corrado Böhm* y *Jacopini* de esa época, demuestran que tales instrucciones no son necesarias y sientan las bases de lo que se conoce como **programación estructurada**

Resolución de problemas (III)

❖ Teorema del Programa Estructurado (*Böhm-Jacopini*)

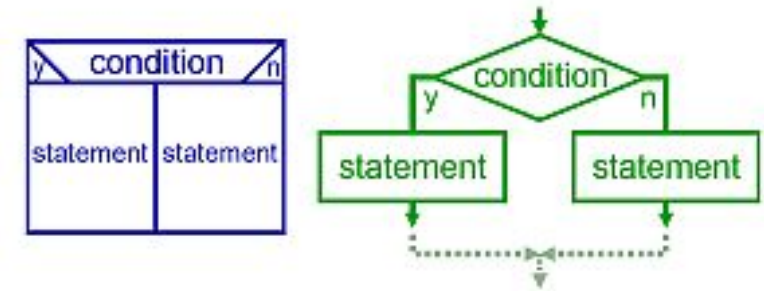
- El teorema del programa estructurado establece que toda función **computable** puede ser implementada en un lenguaje de programación que combine sólo tres estructuras lógicas (o **estructuras de control**) :
 - *Estructura de secuencia*
Ejecución de una instrucción tras otra, según el orden de secuencia



Resolución de problemas (IV)

- ***Estructura de selección o condicional***

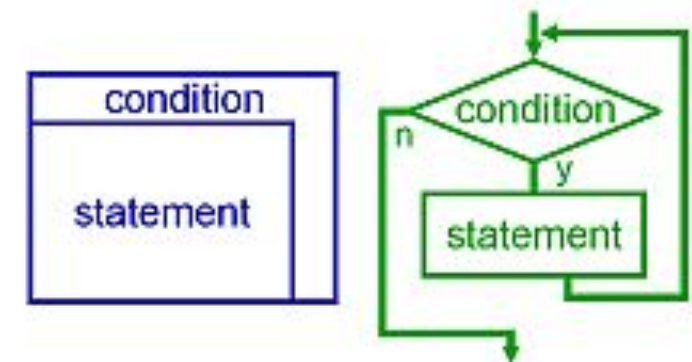
En base al resultado de una expresión *booleana*, podemos establecer que se ejecuten (o no) determinadas instrucciones del programa



- ***Estructura iterativa o de repetición***

Permiten repetir la ejecución de una serie de instrucciones mientras se cumpla determinadas condiciones (expresión *booleana*)

Esta estructura de control también se conoce como *ciclo* o *bucle*



Resolución de problemas (V)

❏ Representación de algoritmos

- A la hora de diseñar y representar algoritmos se utilizan principalmente dos técnicas: **pseudocódigo** y **diagramas de flujo**

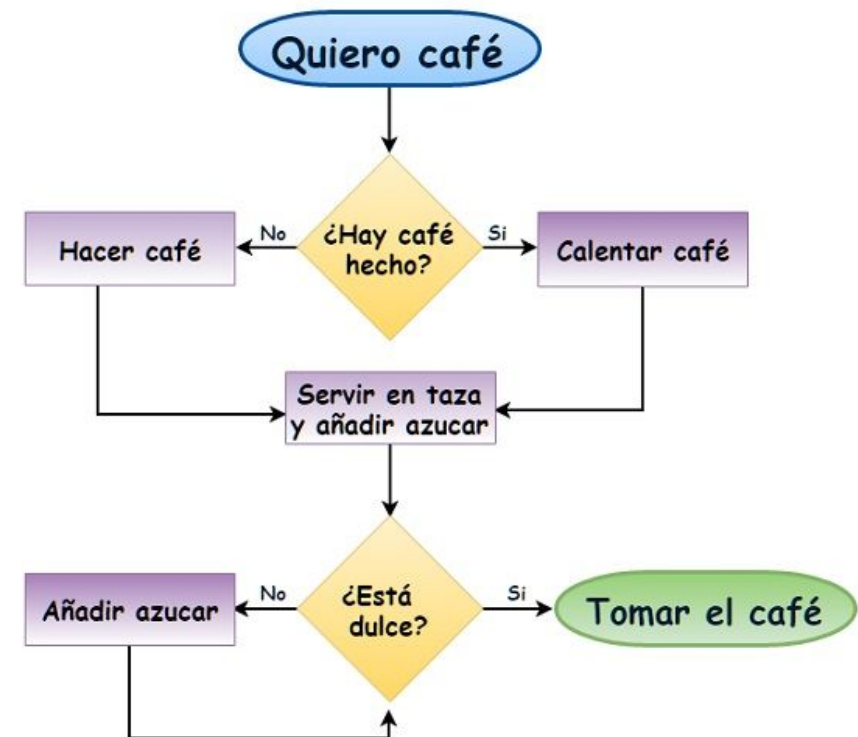
❖ Pseudocódigo

- Descripción de alto nivel en **lenguaje natural**, compacta e **informal** del principio operativo de un programa informático u otro algoritmo
- Utiliza las convenciones estructurales de un lenguaje de programación real (“si”, “mientras”, “repetir”, “ir a”,...), pero diseñado para la lectura humana en lugar de mediante computadoras, y con independencia de cualquier otro lenguaje de programación
- No existe una sintaxis estándar para el pseudocódigo

Resolución de problemas (VI)

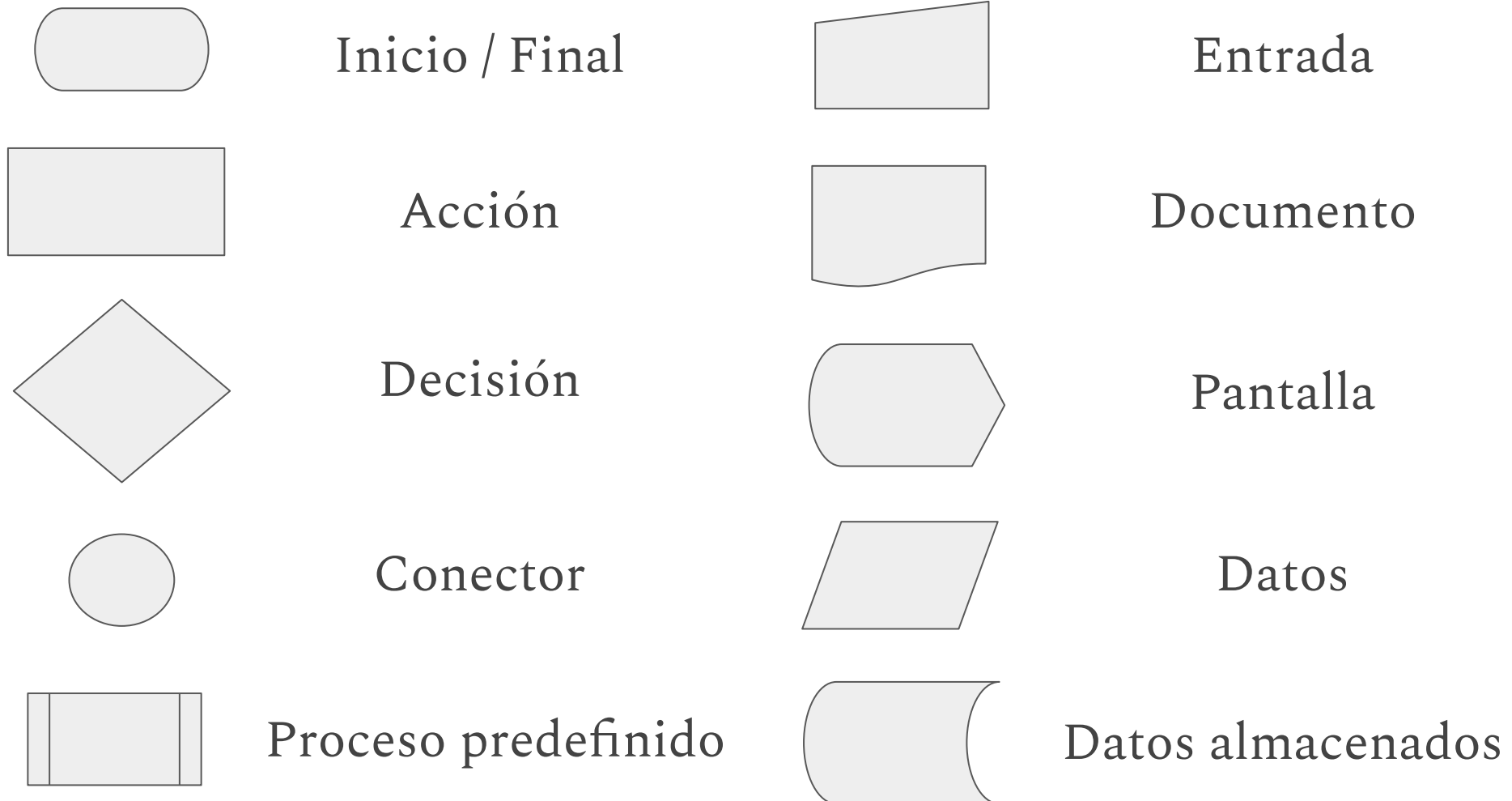
❖ Diagramas de Flujo

- Representación gráfica de un programa informático u otro algoritmo
- Emplean rectángulos, óvalos, diamantes y otras **figuras**, junto con **flechas** conectoras, para establecer el **flujo** y la **secuencia** de ejecución de los distintos pasos del algoritmo
- Los diagramas de flujo se emplean en diversos campos, no sólo programación. Para cualquier proceso productivo, de negocio o fabricación, facilita:
 - Documentar, analizar, comunicar y estandarizar procesos
 - Identificar cuellos de botellas, redundancias y pasos innecesarios



Resolución de problemas (VII)

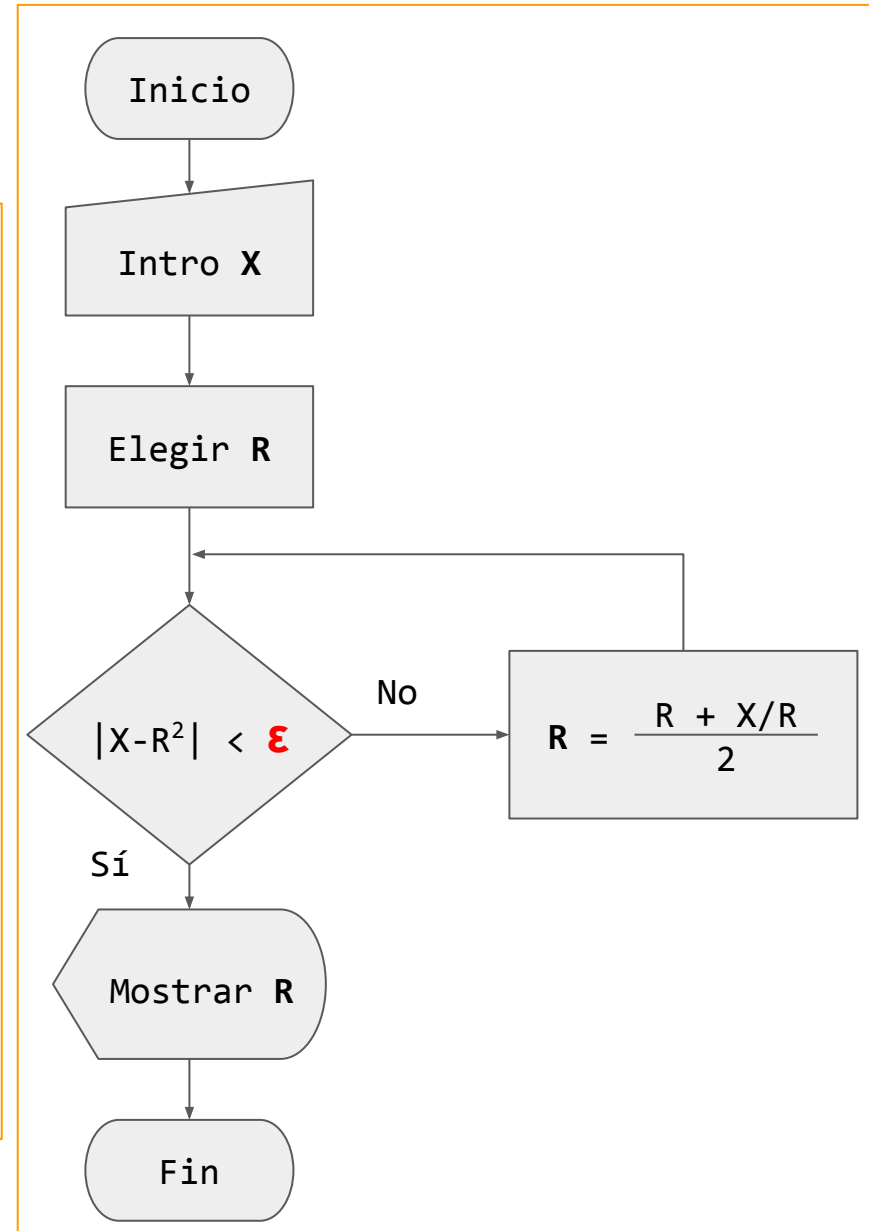
❖ Formas básicas de los diagramas de Flujo



Resolución de problemas (y VIII)

Cálculo raíz cuadrada^(*)
(pseudocódigo y diagrama de flujo)

- Leer el número (X) del que queremos obtener su raíz
- Elegir al azar una estimación del resultado, (R)
- 1 - Si $R \times R$ es suficientemente próximo a X entonces,
 - Mostrar R como solución
 - Finalizar
- Si no,
 - Realizar una nueva estimación con la media de R y X/R
 - Volver a (1) y repetir el proceso



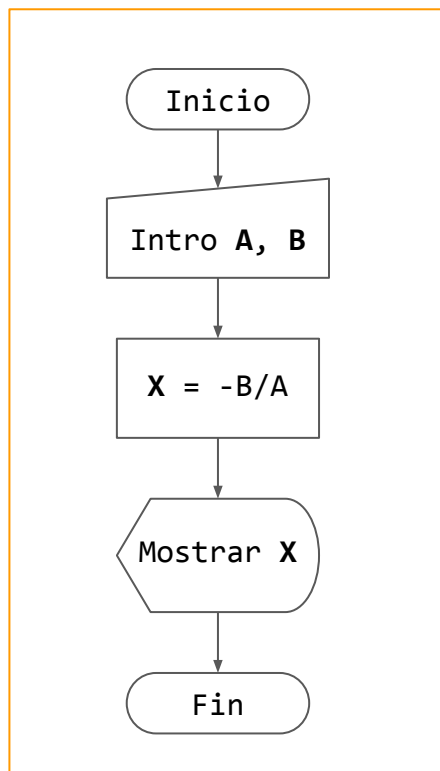
^(*) Algoritmo de Herón de Alejandría, s. I d.C.

Estructuras de selección (I)

- Supongamos que diseñamos un programa para la resolución de ecuaciones de 1^{er} grado de la forma:

$$a \cdot x + b = 0$$

- Una posible solución podría ser:



```
//: Ecu1GSolver.java
import java.util.Scanner;

class Ecu1GSolver {
    public static void main(String[] args) {
        Scanner entrada = new Scanner(System.in);

        System.out.print("Introduce el coeficiente A: ");
        double a = entrada.nextFloat();
        System.out.print("Introduce el coeficiente B: ");
        double b = entrada.nextFloat();

        System.out.printf("La solución es X = %.2f\n", -b/a);
        System.out.println("Fin programa");
    }
}
```

Estructuras de selección (II)

- Vamos a ejecutar nuestro programa:

```
~$ java Ecu1GSolver  
Introduce el coeficiente A: 2  
Introduce el coeficiente B: 5  
La solución es x = -2.50  
Fin programa
```



- Sin embargo, vamos a ejecutarlo de nuevo con estos otros coeficientes:

```
~$ java Ecu1GSolver  
Introduce el coeficiente A: 0  
Introduce el coeficiente B: 4  
La solución es X = -Infinity  
Fin programa
```



- Como era de esperar, se ha intentado una división por 0, dando lugar al correspondiente valor de *infinito*. Tal vez, deberíamos informar al usuario de que sólo es posible obtener una solución real **si** el coeficiente **A es distinto de 0**

Estructuras de selección (III)

La sentencia *if* (selección simple)

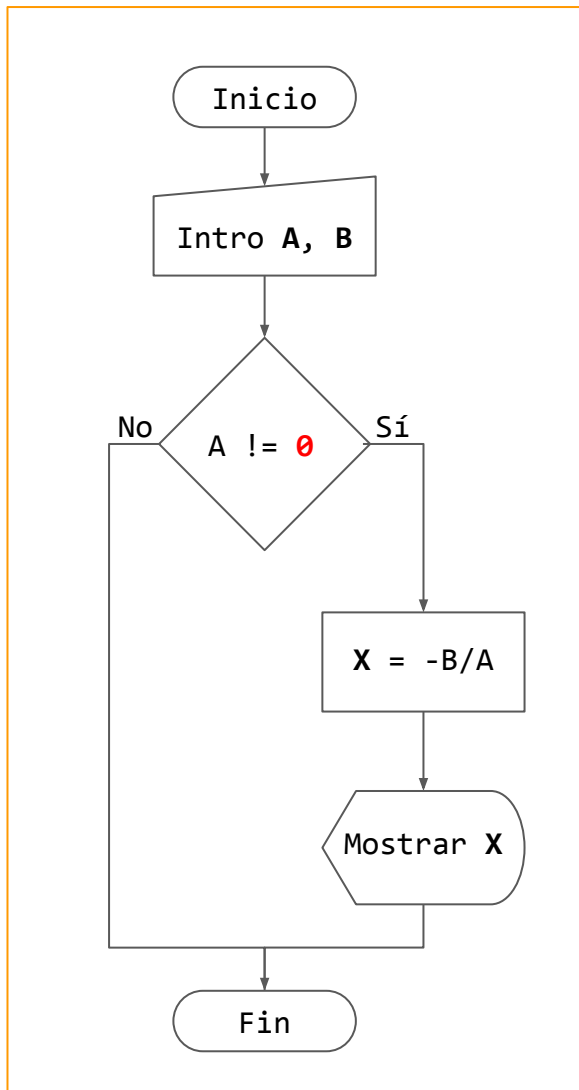
- Java nos proporciona la instrucción *if* para evaluar una **condición** y, **sólo en caso de que sea cierta**, ejecutar las acciones que consideremos. Su formato es:

```
if (condición) {  
    // si condición es true  
    bloque de acciones;  
}
```

- **condición** será cualquier *expresión booleana* que devuelva un valor lógico.
 - Si el resultado de evaluar dicha expresión es *verdadero* (**true**), se ejecutarán aquellas **acciones** (sentencias) definidas a continuación
 - Si se ha definido más de una acción, dicho **bloque de instrucciones** deberá ir necesariamente delimitado por unas llaves (**{ }**)

Estructuras de selección (IV)

- Modifiquemos nuestro programa para tener en cuenta la división por 0:



```
//: EculGSolver.java
import java.util.Scanner;

class EculGSolver {
    public static void main(String[] args) {
        Scanner entrada = new Scanner(System.in);

        System.out.print("Introduce el coeficiente A: ");
        double a = entrada.nextFloat();

        System.out.print("Introduce el coeficiente B: ");
        double b = entrada.nextFloat();

        if( a != 0) {
            System.out.printf("La solución es X = %.2f\n", -b/a);
        }

        System.out.println("Fin programa");
    }
}
```

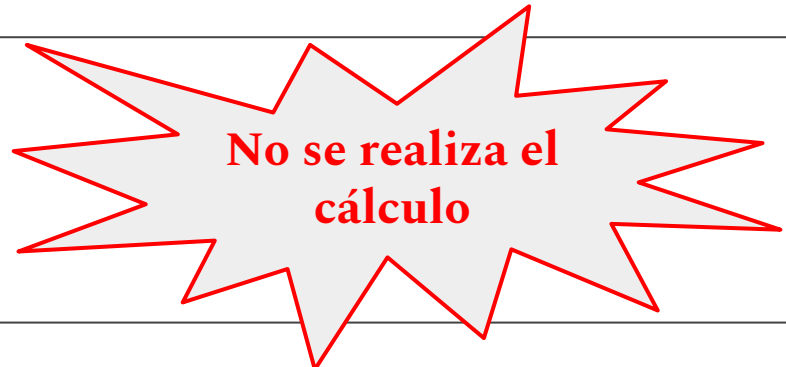
Estructuras de selección (V)

- Vamos a ejecutar de nuevo nuestro programa:

```
~$ java Ecu1GSolver
Introduce el coeficiente A: 2
Introduce el coeficiente B: 5
La solución es x = -2.50
Fin programa
```

- Sin embargo, vamos a ejecutarlo de nuevo con estos otros coeficientes:

```
~$ java Ecu1GSolver
Introduce el coeficiente A: 0
Introduce el coeficiente B: 4
Fin programa
```



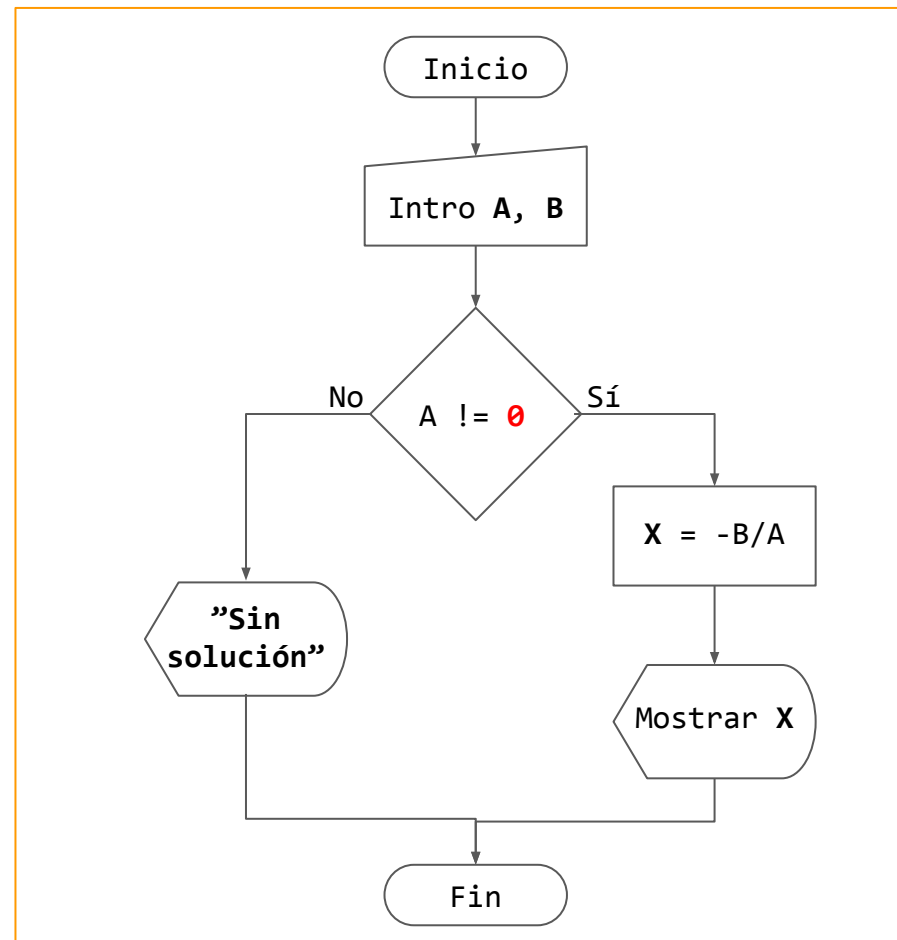
**No se realiza el
cálculo**

- En el segundo caso, al no cumplirse la condición (la expresión **a != 0** devuelve un valor *false*), se **salta** el bloque de instrucciones incluido en el **if** y se continúa en la siguiente instrucción:

```
System.out.println("Fin programa");
```


Estructuras de selección (VI)


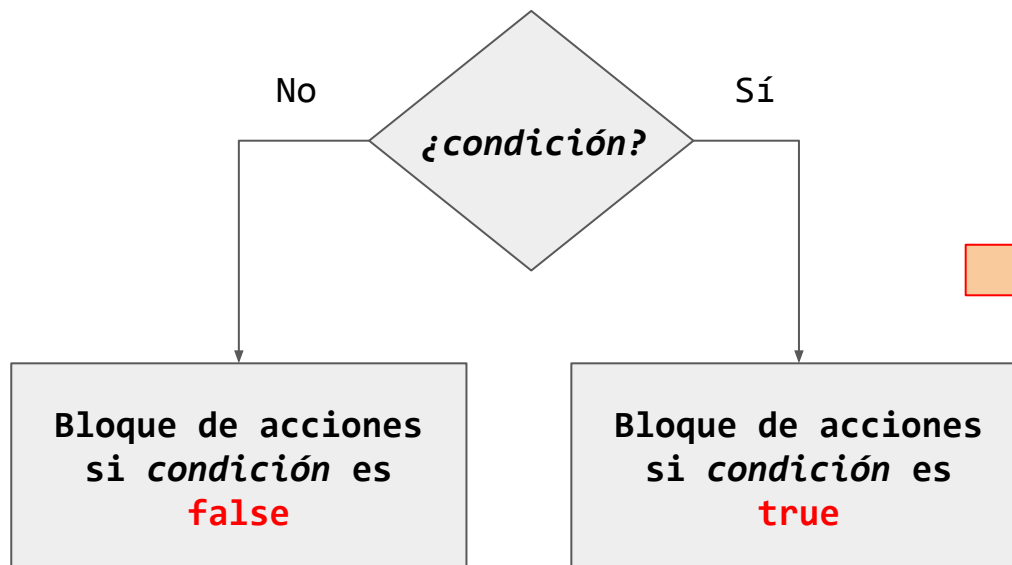
- Hemos evitado la división por 0 pero deberíamos informar de dicha situación al usuario en lugar de finalizar sin más. Por ejemplo:



Estructuras de selección (VI)

La construcción *if - else* (selección doble)

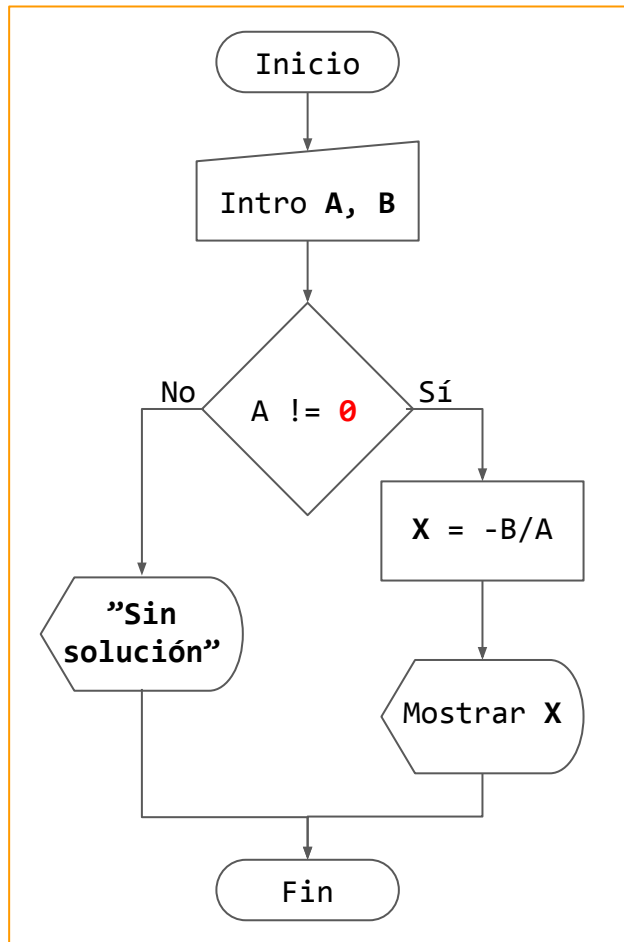
- Para situaciones mutuamente excluyentes, como en el caso anterior, donde debemos realizar unas determinadas acciones en caso de que se verifique una condición y, otras acciones diferentes en el caso contrario, Java nos proporciona la construcción *if - else* con el siguiente formato:



```
if (condición) {  
    // si condición es true  
    bloque de acciones;  
}  
else {  
    // si condición es false  
    bloque de acciones;  
}
```

Estructuras de selección (VII)

- Finalmente, nuestro código quedará de la siguiente manera:



```
//: Ecu1GSolver.java
import java.util.Scanner;

class Ecu1GSolver {
    public static void main(String[] args) {
        Scanner entrada = new Scanner(System.in);

        System.out.print("Introduce el coeficiente A: ");
        double a = entrada.nextFloat();

        System.out.print("Introduce el coeficiente B: ");
        double b = entrada.nextFloat();

        if( a != 0) {
            System.out.printf("La solución es X = %.2f\n", -b/a);
        }
        else {
            System.out.println("La ecuación no tiene solución");
        }

        System.out.println("Fin programa");
    }
}
```

Estructuras de selección (VIII)

- Y el resultado de ejecutarlo sería:

➤ con el coeficiente $a \neq 0$

```
~$ java Ecu1GSolver
Introduce el coeficiente A: 2
Introduce el coeficiente B: 5
La solución es x = -2.50
Fin programa
```

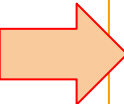
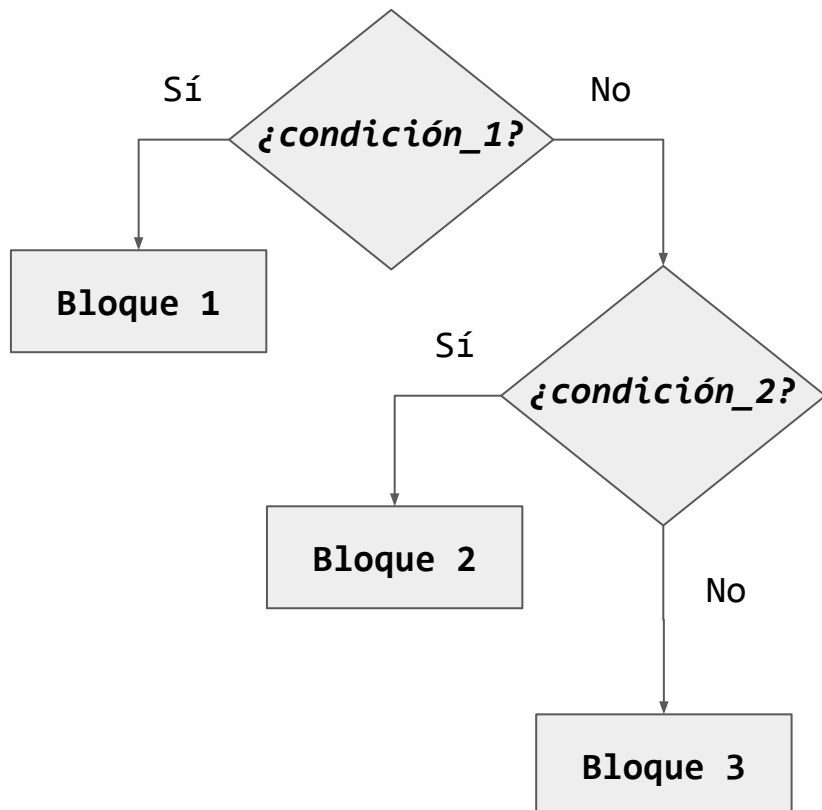
➤ con el coeficiente $a == 0$

```
~$ java Ecu1GSolver
Introduce el coeficiente A: 0
Introduce el coeficiente B: 4
La ecuación no tiene solución
Fin programa
```

Estructuras de selección (IX)

❏ Condicionales anidados

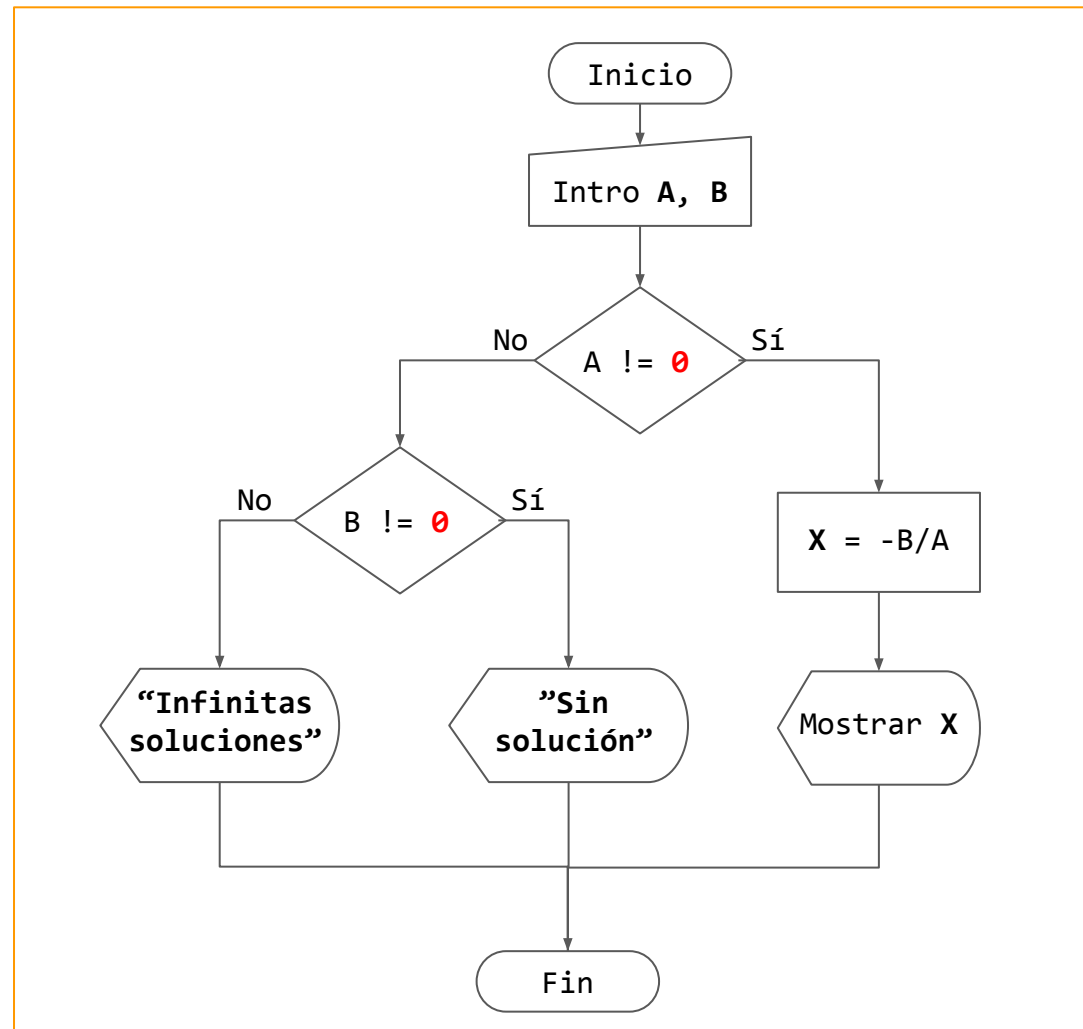
- Pueden darse casos en los que necesitamos “insertar” condicionales dentro de otros (*anidamiento*), creando estructuras como la siguiente:



```
if (condición_1) {  
    // Se ejecuta si condición_1 es true  
    bloque de acciones 1;  
}  
else {  
    // Se ejecuta si condición_1 es false  
    if (condición_2) {  
        // Se ejecuta si condición_2 es true  
        bloque de acciones 2;  
    }  
    else {  
        // Se ejecuta si ambas condiciones son false  
        bloque de acciones 3;  
    }  
}
```

Estructuras de selección (X)

- Volvamos al ejemplo. Queremos que, si a es 0, distinguir cuando b es 0 (infinitas soluciones) y b es distinto de 0 (no tiene solución)



Estructuras de selección (XI)

```
//: Ecu1GSolver.java
import java.util.Scanner;
class Ecu1GSolver {
    public static void main(String[] args) {
        Scanner entrada = new Scanner(System.in);

        System.out.print("Introduce el coeficiente A: ");
        double a = entrada.nextFloat();

        System.out.print("Introduce el coeficiente B: ");
        double b = entrada.nextFloat();

        if( a != 0) {
            System.out.printf("La solución es X = %.2f\n", -b/a);
        }
        else {
            if( b != 0) {
                System.out.println("La ecuación no tiene solución");
            }
            else {
                System.out.println("Indeterminado. La ecuación tiene infinitas soluciones");
            }
        }
        System.out.println("Fin programa");
    }
}
```


Estructuras de selección (XII)

- En situaciones de *if-else* anidados como la anterior, podemos escribirlos de una manera un poco más “*elegante*”:

```
if (condición_1) {  
    // Se ejecuta si condición_1 es true  
    bloque de acciones 1;  
}  
else if (condición_2) {  
    // Se ejecuta si condición_2 es true  
    bloque de acciones 2;  
}  
else {  
    // Se ejecuta si ambas condiciones son false  
    bloque de acciones 3;  
}
```

Estructuras de selección (XIV)

- El código final de nuestro pequeño programa será:

```
//: Ecu1GSolver.java
import java.util.Scanner;
class Ecu1GSolver {
    public static void main(String[] args) {
        Scanner entrada = new Scanner(System.in);

        System.out.print("Introduce el coeficiente A: ");
        double a = entrada.nextFloat();

        System.out.print("Introduce el coeficiente B: ");
        double b = entrada.nextFloat();

        if( a != 0) {
            System.out.printf("La solución es X = %.2f\n", -b/a);
        }
        else if( b != 0) {
            System.out.println("La ecuación no tiene solución");
        }
        else {
            System.out.println("Indeterminado. La ecuación tiene infinitas soluciones");
        }
        System.out.println("Fin programa");
    }
}
```

Estructuras de selección (XV)

El operador condicional (?:)

- Java cuenta con el **operador condicional (?:)** que, en ocasiones, puede utilizarse en lugar de una instrucción if...else, haciendo el código más corto y claro. Es el único operador **ternario** de Java. Su formato es:
- Es el único operador **ternario** de Java. Su formato es:

```
(condición) ? valor_si_expr_true : valor_si_expr_false;
```


- **condición** será cualquier *expresión booleana* que devuelva un valor lógico.
 - Si el resultado de evaluar dicha expresión es verdadero (**true**), se devolverá el valor *valor_si_expr_true*
 - En caso contrario, es decir, *condición* es **false**, se devolverá el valor *valor_si_expr_false*

Estructuras de selección (XVI)

❖ Algunos ejemplos

```
maxValue = (a > b) ? a : b;
```


es equivalente a:



```
if (a > b)
    maxValue = a;
else
    maxValue = b;
```

```
System.out.println( (notaMedia >= 5.0) ? "Aprobado" : "Suspenso" );
```

es equivalente a:



```
if (notaMedia >= 5.0)
    System.out.println("Aprobado");
else
    System.out.println("Suspenso");
```

Estructuras de selección (XVII)

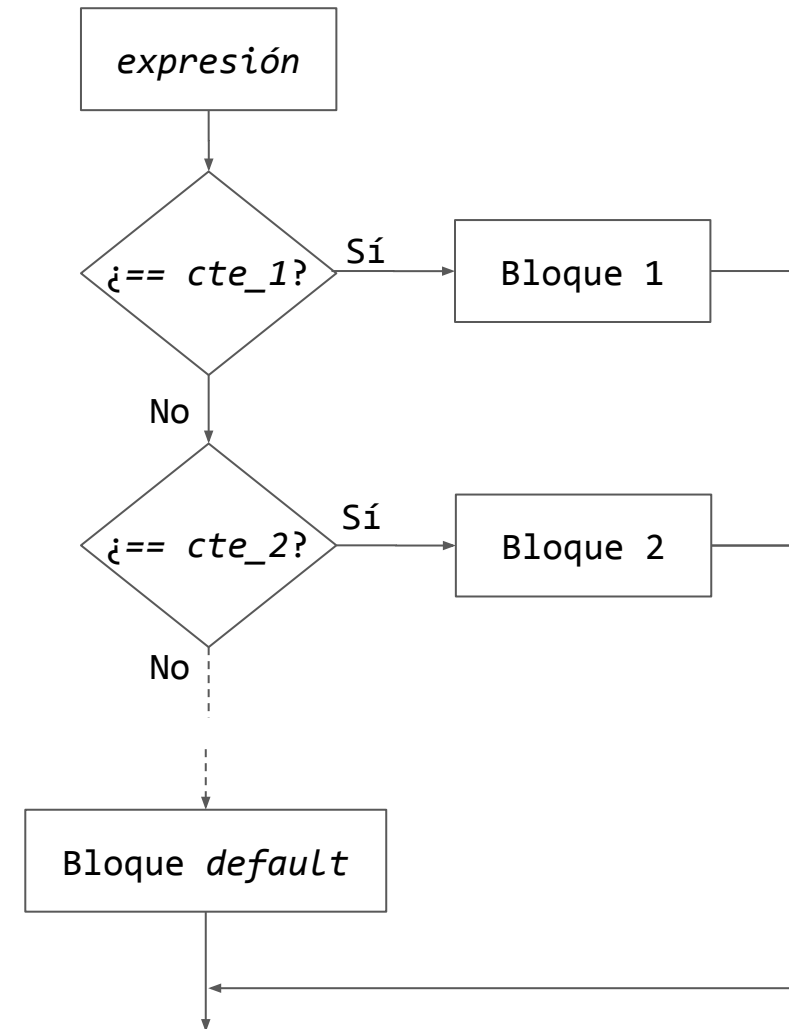
La sentencia *switch* (selección múltiple)

- La sentencia *switch* proporciona un mecanismo de selección múltiple, es decir, nos permite **seleccionar una opción** (*case*) **entre varias alternativas**
- Aunque una serie de *if*'s anidados nos permiten realizar múltiples tests, en ocasiones el uso de *switch* será más eficiente y legible
- Su funcionamiento se basa en proporcionar una **expresión** y una serie de valores posibles o **casos**. Si el resultado de evaluar la expresión coincide con alguno de los valores o *casos* propuestos, se ejecutarán aquellas acciones que estén definidas para dicho *caso de uso*
- Los diferentes *casos de uso* se comprueban de modo secuencial. En el momento en que uno de ellos sea válido, es decir, su valor coincida con el de la expresión evaluada, se ejecutarán las instrucciones que tenga definidas

Estructuras de selección (XVIII)

- Su formato es el siguiente:

```
switch (expresión) {  
    case constant_1:  
        bloque de acciones 1;  
        break;  
    case constant_2:  
        bloque de acciones 2;  
        break;  
    . . .  
    default:      // opcional  
        bloque de acciones default;  
}
```



Estructuras de selección (XIX)

- En las versiones anteriores a JDK 7, la *expresión* que controla el *switch* debía devolver un valor *byte*, *short*, *int*, *char* o una *enumeración*. Desde el JDK 7 ya se pueden utilizar valores de tipo *String*
- Cada valor especificado en una sentencia *case* debe ser un valor/expresión constante en tiempo de compilación (**literal** o **constante final**) y del mismo tipo (o compatible) que la *expresión* evaluada. No puede haber dos cláusulas *case* con el mismo valor
- El bloque de instrucciones asociado a la cláusula **default** se ejecutará si ninguno de los valores de las cláusulas *case* coincide con el resultado de la expresión evaluada. La cláusula *default* es **opcional**
- Técnicamente, la sentencia *break* de cada *case* es opcional. De no usarse, tras ejecutar las instrucciones del *case* correspondiente, se ejecutarían las instrucciones del *case posterior* (y sucesivamente de no haber más *break*)

Estructuras de selección (XIX)

❖ Ejemplo:

```
public static void imprimeTextoNota(int nota) {  
    switch(nota) {  
        case 10:  
            System.out.println("Sobresaliente");  
            break;  
        case 9:  
            System.out.println("Sobresaliente");  
            break;  
        case 8:  
            System.out.println("Notable");  
            break;  
        case 7:  
            System.out.println("Notable");  
            break;  
        case 6:  
            System.out.println("Bien");  
            break;  
        case 5:  
            System.out.println("Aprobado");  
            break;  
        case 4:  
            System.out.println("Insuficiente");  
            break;  
    }
```

```
        case 3:  
            System.out.println("Muy deficiente");  
            break;  
        case 2:  
            System.out.println("Muy deficiente");  
            break;  
        case 1:  
            System.out.println("Muy deficiente");  
            break;  
        case 0:  
            System.out.println("Muy deficiente");  
            break;  
        default:  
            System.out.println("Nota no válida");  
    }  
}
```

Estructuras de selección (y XX)

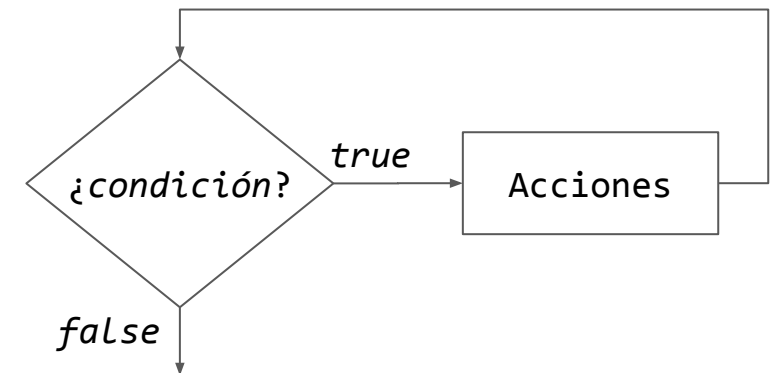
❖ Podemos simplificar el ejemplo anterior...

```
public static void imprimeTextoNota(int nota) {  
    switch(nota) {  
        case 10:  
        case 9:  
            System.out.println("Sobresaliente");  
            break;  
        case 8:  
        case 7:  
            System.out.println("Notable");  
            break;  
        case 6:  
            System.out.println("Bien");  
            break;  
        case 5:  
            System.out.println("Aprobado");  
            break;  
        case 4:  
            System.out.println("Insuficiente");  
            break;  
    }
```

```
        case 3:  
        case 2:  
        case 1:  
        case 0:  
            System.out.println("Muy deficiente");  
            break;  
        default:  
            System.out.println("Nota no válida");  
    }  
}
```

Estructuras iterativas (I)

- Las *estructuras iterativas* o *de repetición* nos van a permitir repetir la ejecución de una serie de instrucciones mientras se cumpla una determinada **condición**, descrita mediante una expresión *booleana*
- Esta estructura de control también se conoce como *ciclo* o *bucle*
- Al término de cada iteración del bucle, se reevaluará la condición para determinar si se sigue cumpliendo o no. En caso de hacerlo, se procederá a realizar una nueva iteración y se ejecutarán nuevamente las sentencias incluidas en el bucle
- Debemos **verificar** siempre que los parámetros sobre los que se establece la condición, se modifican dentro del bucle. De lo contrario, una vez dentro, nunca saldríamos (**bucle infinito**)



Estructuras iterativas (II)

El bucle *while*

- Los bucles *while* son los tipos más simples de bucle. En ellos se establece una *condición* y, mientras esa condición se cumpla, el bloque de instrucciones asociado al bucle *while* continuará ejecutándose.
- Su formato es:

```
while (condición) {  
    // si condición es true  
    bloque de acciones;  
}
```

- *condición* será cualquier expresión evaluable que devuelva un valor booleano *true* o *false*

Estructuras iterativas (III)

- El siguiente ejemplo usa un bucle *while* para crear un contador. En cada **iteración** del bucle, se irán ejecutando cada una de las instrucciones del bloque *while*. Tras ejecutar la última, se volverá a **comprobar** la condición. **Si se cumple**, se realizará una nueva iteración. En caso contrario, se continuará con la instrucción siguiente al bloque *while*

```
int temp = 10;

System.out.println("Iniciando cuenta atrás...");

while(temp>0) {
    System.out.println(temp);
    temp--;
}

System.out.println("Despegue!");
```

Al **principio** de cada **iteración**, comprobamos el valor de **temp**

Si se cumplió la condición (**temp>0**), ejecutamos las instrucciones del bucle

El proceso sigue **repitiéndose** hasta que deje de cumplirse la condición

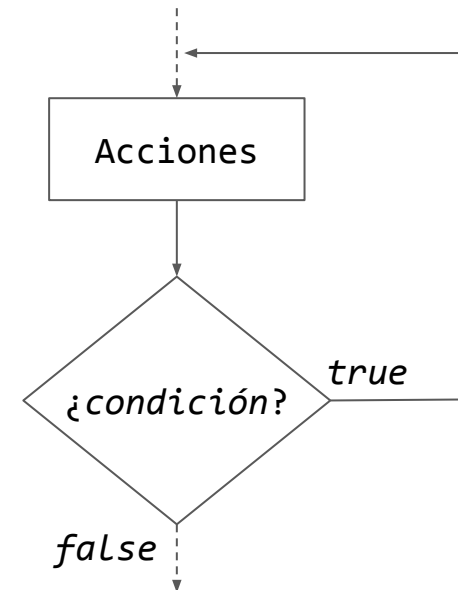
Esta instrucción **sólo** se ejecutará una vez que se salga del bucle

Estructuras iterativas (IV)

El bucle *do-while*

- El bucle *do-while* es similar al bucle *while* pero con la **diferencia** de que la condición se examina a la salida del bucle, no a la entrada.
Por tanto, **siempre** ejecutará, al menos, **una iteración** del mismo
- Su formato es:

```
do {  
    bloque de acciones;  
} while (condición);
```



Estructuras iterativas (V)

- Vamos a reescribir el ejemplo anterior usando ahora un bucle *do-while*

```
int temp = 10;

System.out.println("Iniciando cuenta atrás...");

do {
    System.out.println(temp);
    temp--;
} while(temp > 0);

System.out.println("Despegue!");
```

Las instrucciones del bucle se ejecutan, **al menos**, una vez

El proceso sigue **repitiéndose** hasta que deje de cumplirse la condición (**$temp > 0$**)

Al **final** de cada **iteración**, comprobamos el valor de **temp**

Esta instrucción **sólo** se ejecutará una vez que se salga del bucle

- En general, las construcciones *while* y *do-while* son, con pequeños ajustes, intercambiables. Sin embargo, habrá situaciones donde el empleo de una, u otra, puede mejorar la legibilidad del programa

Estructuras iterativas (VI)

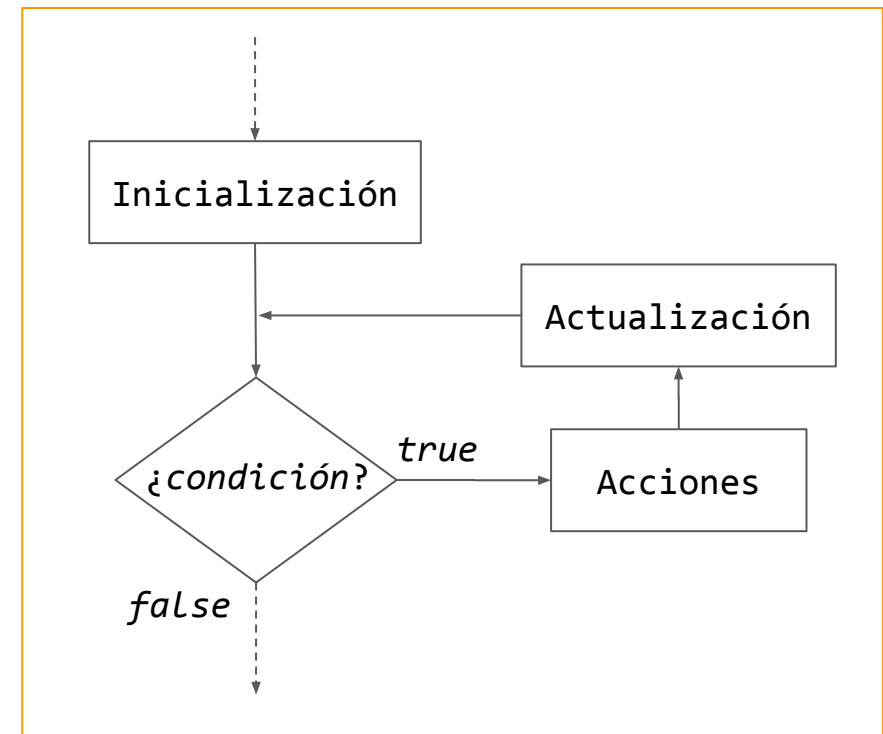
El bucle *for*

- Java nos proporciona la sentencia *for* para crear bucles controlados por un **contador**. Aunque este tipo de bucles son fácilmente implementables con *while*, la sentencia *for* nos permite definir en una única línea la inicialización, incremento y *test* de la **variable de control** del bucle
- Su formato general es:

```
for (inicialización; condición; actualización) {  
    // si condición es true  
    bloque de acciones;  
}
```

Estructuras iterativas (VII)

- La especificación del bucle *for* consta de tres expresiones **opcionales**:
 - **inicialización**, suele consistir en una operación de asignación en la que se establece el valor inicial de la *variable de control* del bucle. Se ejecutará una única vez al inicio de la sentencia *for*
 - **condición**, expresión booleana y que determina si se ejecutará, o no, una nueva iteración del bucle. Se evalúa antes de cada nueva iteración
 - **actualización**, generalmente será una expresión mediante la cual se actualiza el valor actual de la *variable de control* o contador. Se ejecutará al final de cada iteración del bucle



Estructuras iterativas (VIII)

❖ Ejemplo: calcula las raíces de los primeros 100 números...

```
//: Raices.java
class Raices {
    public static void main(String[] args) {
        int num;
        double raiz, err;

        for(num=1; num<101; num++) {
            raiz = Math.sqrt(num);
            err = num - (raiz*raiz);
            System.out.println("\nLa raíz de " + num + " es: " + raiz);
            System.out.println("El error de redondeo es: " + err);
        }
    }
}

/* Output:
. . .
La raíz de 97 es: 9.848857801796104
El error de redondeo es: 1.4210854715202004E-14

La raíz de 98 es: 9.899494936611665
El error de redondeo es: 0.0
. . .
*///:~
```

inicialización -> **num = 1**

condición -> **num < 101**

actualización -> **num++**

Estructuras iterativas (IX)

❏ Más ejemplos...

- La *variable de control* del bucle se puede tanto incrementar como decrementar, y por cualquier cantidad. El siguiente ejemplo imprime los números de 100 a -100 de manera decreciente...

```
int x;  
for(x=100; x>-101; x-=5) { // decremento de 5 en 5  
    System.out.println(x);  
}
```

- Como la *condición* se testea por primera vez al entrar en el *for*, puede suceder que el bucle no llegue a ejecutarse nunca...

```
int x;  
for(x=10; x<5; x++) {  
    System.out.println(x);    // no se ejecuta nunca  
}
```

Estructuras iterativas (X)

- En las secciones de *inicialización* y *actualización*, podemos inicializar y actualizar más de una variable, separadas por **comas**...

```
int i, j;  
for(i=0, j=10; i<j; i++, j--) {  
    System.out.println("i: " + i + "\tj: " + j);  
}
```

- En la sección *condición* podrá haber cualquier expresión que devuelva un valor *booleano*

```
int i;  
Scanner scn = new Scanner(System.in);  
System.out.println("Pulsa [Q] para terminar");  
  
for(i=0; scn.nextLine().charAt(0) != 'Q'; i++) {  
    System.out.println("Iteración #" + i);  
}
```

Estructuras iterativas (XI)

- Las distintas secciones de la sentencia *for* son **opcionales** (aunque deberemos mantener los **;**)...

```
int x;

x = 0;    // realizamos la inicialización fuera del bucle
for(; x<10; ) {
    System.out.println(x);
    x++;  // actualización de la vble de control
}
```

- Hasta podemos omitir el *bloque de acciones*...

```
int i, sum;

// suma los números del 1 al 10
for(i=1, sum=0; i<=10; sum += i++); // termina en ;
System.out.println("La suma es: " + sum);
```

Estructuras iterativas (XII)

Las sentencias *break* y *continue*

- En ocasiones, precisaremos abandonar prematuramente o alterar la ejecución normal del bucle.

❖ *break*

- La sentencia *break* fuerza la salida del bucle en el que nos encontremos:

```
int num = 100;

for(int i=0; i<num; i++) {
    if(i*i >= num) {
        break; // salimos del bucle si i*i >= 100
    }
    System.out.print(i + " ");
}
System.out.print("Fin bucle");

/* Output: 0 1 2 3 4 5 6 7 8 9 Fin bucle */
```

Estructuras iterativas (XIII)

- Cuando se usa dentro de bucles anidados, finalizará sólo la ejecución del bucle interno en el que se encuentre:

```
for(int i=0; i<3; i++) {  
    System.out.println("Bucle externo: " + i);  
    System.out.print("\tBucle interno: ");  
    int t = 0;  
    while(t < 100) {  
        if(t == 10) break; // salimos del bucle interno si t = 10  
        System.out.print(t + " ");  
        t++;  
    }  
    System.out.println();  
}
```

```
Bucle externo: 0  
    Bucle interno: 0 1 2 3 4 5 6 7 8 9  
Bucle externo: 1  
    Bucle interno: 0 1 2 3 4 5 6 7 8 9  
Bucle externo: 2  
    Bucle interno: 0 1 2 3 4 5 6 7 8 9
```


Estructuras iterativas (XIV)

- Java nos permite asignar **etiquetas** a los bucles, de forma que puedan ser empleadas con *break* para indicar de cuál queremos forzar la salida:

```
ext_loop:
for(int i=0; i<10; i++) {
    int_loop:
    for(int j=0; j<10; j++) {
        if(j>i)
            break int_loop; // salimos del bucle interno
        if(i==5)
            break ext_loop; // salimos del bucle externo
        System.out.print(j + " ");
    }
    System.out.println();
}
```

```
0
01
012
0123
01234
```

Estructuras iterativas (XV)

❖ *continue*

- La sentencia *continue* dentro de un bucle, provoca que **finalice** la **iteración actual** omitiendo la ejecución de las instrucciones restantes. Es decir, *continue* “fuerza” que se salte a la siguiente iteración, provocando un nuevo ciclo de *actualización* (*for*), evaluación de la *condición* (*for/while*) y ejecución del bloque de sentencias del bucle si procede

```
// Imprime sólo números pares
System.out.print("Números pares menores que 20: ");
for(int i=0; i<20; i++) {
    if( i%2 != 0 )
        continue;
    System.out.print(i + " ");
}
```

Si *i* es **par**, salta al principio del bucle (*continue*) y la sentencia *print* no se ejecuta. A continuación, se incrementa *i* (*i++*) y, si cumple la condición (*i<20*), se ejecuta una nueva iteración del bucle

Números pares menores que 20: 0 2 4 6 8 10 12 14 16 18

Estructuras iterativas (y XVI)

- Como en el caso de *break*, podemos emplear **etiquetas** con *continue* para indicar sobre qué bucle queremos forzar la siguiente iteración

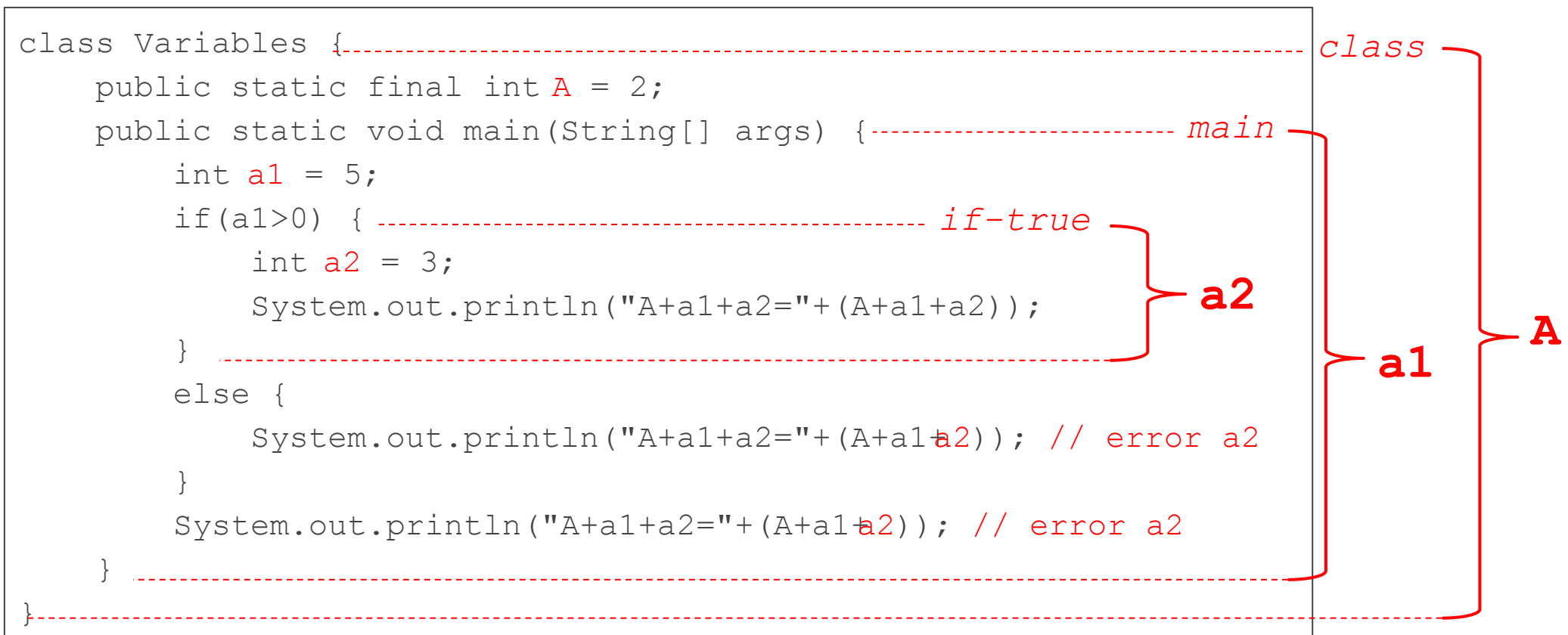
```
ext_loop:
for(int i=0; i<10; i++) {
    System.out.println();
    for(int j=0; j<10; j++) {
        if(j>i)
            continue ext_loop;    // avanzamos el bucle externo
        if(i==5)
            break ext_loop;       // salimos del bucle externo
        System.out.print(j + "");
    }
}
```

```
0
01
012
0123
01234
```

Estructuras y ámbito de las variables (I)

- El *alcance* o *visibilidad* de una variable queda determinado por el **bloque** (estructura de control, método,...) donde haya sido declarada. Una vez finalizado dicho bloque, dicha variable dejará de existir y cualquier intento de acceso a ella generará un error de compilación

```
class Variables { ..... class
    public static final int A = 2;
    public static void main(String[] args) { ..... main
        int a1 = 5;
        if(a1>0) { ..... if-true
            int a2 = 3;
            System.out.println("A+a1+a2="+ (A+a1+a2));
        } .....
        else {
            System.out.println("A+a1+a2="+ (A+a1+a2)); // error a2
        }
        System.out.println("A+a1+a2="+ (A+a1+a2)); // error a2
    } .....
}
```



Estructuras y ámbito de las variables (y II)

❖ Declaración de la *variable de control* del bucle

- Es frecuente que la *variable de control* del bucle sólo se necesite para ese propósito. En ese caso, es posible declararla en la sección de *inicialización*, teniendo en cuenta que su **alcance** quedará reducido a la propia sentencia *for*. Al abandonar el bucle, dicha variable dejará de existir

```
int sum = 0;

// suma los números del 1 al 10
for(int i=1; i<=10; i++) {
    suma += i;
}

// la variable i no existe aquí

System.out.println("La suma es: " + sum);
```