



ESCUELA DE
INGENIERÍA EN CIENCIAS Y SISTEMAS
FACULTAD DE INGENIERÍA
UNIVERSIDAD DE SAN CARLOS DE GUATEMALA



Día, Fecha:

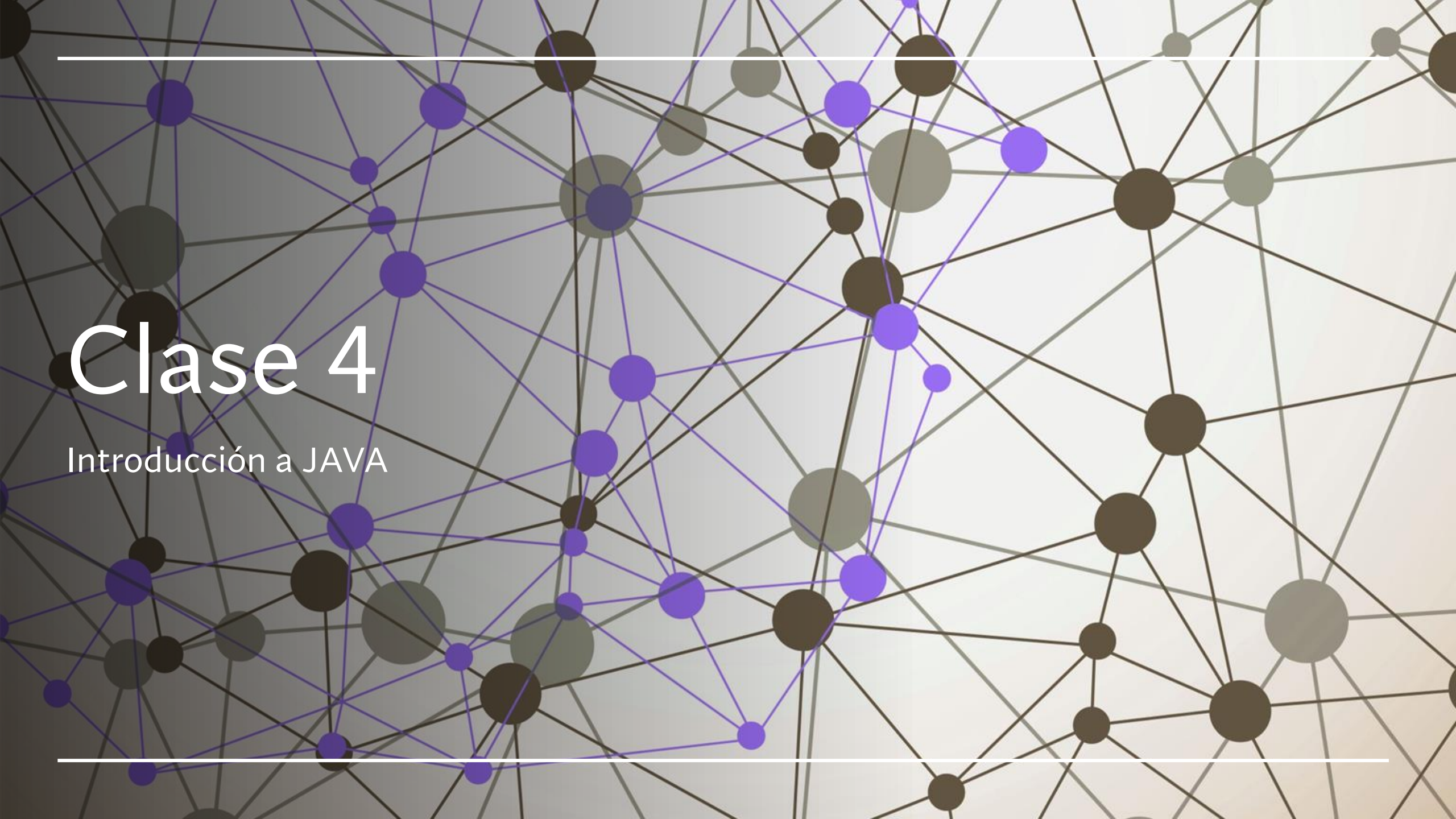
Miércoles, 12 de
febrero

Hora de inicio:

4:30 – 6:10 pm

Introducción a la Programación y Computación 1 Sección G

Max Rodrigo Durán Canteo



Clase 4

Introducción a JAVA

Agenda

01

Expresiones
Aritméticas

02

Input/Output

03

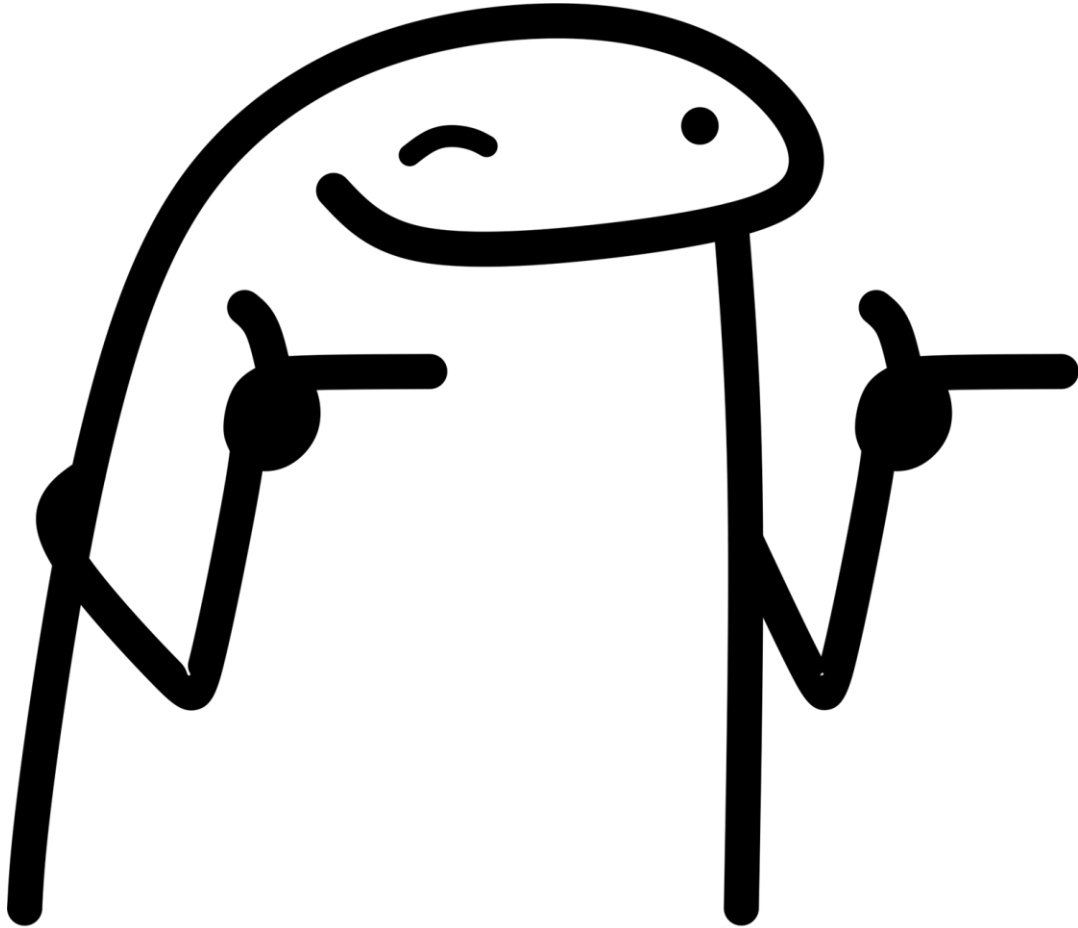
Estructuras
de Control y
Ciclos

04

Excepciones

05

Debuggin



Expresiones Aritméticas

Tipos

Expresiones Aritméticas

Operador	Significado	Ejemplo
+	Suma	2 + 2
-	Resta	5 - 3
*	Multiplicación	6 * 9
/	División	15 / 3
%	Modulo (Residuo)	4 % 2

Operador	Significado	Forma Utilizar	Equivalente
+=	Suma	Op1 += Op2	Op1 = Op1 + Op2
-=	Resta	Op1 -= Op2	Op1 = Op1 - Op2
*=	Multiplicación	Op1 *= Op2	Op1 = Op1 * Op2
/=	División	Op1 /= Op2	Op1 = Op1 / Op2
%=	Modulo (Residuo)	Op1 %= Op2	Op1 = Op1 % Op2

Expresiones Aritméticas

Valor Inicial X	Expresión	Valor Y	Valor X
5	$Y = X++$	5	6
5	$Y = ++X$	6	6
5	$Y = X--$	5	4
5	$Y = --X$	4	4

Expresiones de
Incremento/Decremento

Operador	Significado	Forma Utilizar
>	Mayor que	Op1 > Op2
<	Menor que	Op1 < Op2
>=	Mayor o igual que	Op1 >= Op2
<=	Menor o igual que	Op1 <= Op2
==	Igual a	Op1 == Op2
!=	Diferente a	Op1 != Op2

Expresiones Relacionales

Expresiones Lógicas

Operador	Significado	Forma Utilizar
&&	Y (AND)	Op1 && Op2
	O (OR)	Op1 Op2
!	NOT	!Op1

Input (Entrada)

Para leer la entrada de usuario es necesario usar la clase Scanner que se encuentra en el paquete java.util:

```
import java.util.Scanner();  
Scanner valor = new Scanner(System.in);
```

Método	Valor Devuelto
nextInt()	Entero
nextDouble()	Decimal
next()	Caracter
nextLine()	Cadena de caracteres



Output (Impresión en consola)

Para imprimir los valores de las variables en la consola, se utiliza lo siguiente:

Impresión con salto de línea

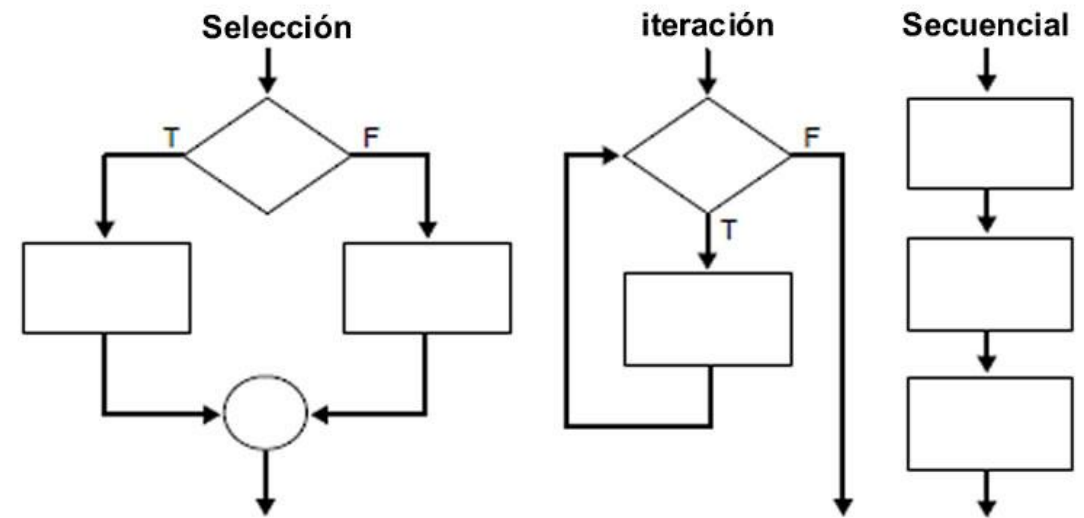
- `System.out.println();`

Impresión sin salto de línea

- `System.out.print();`
-

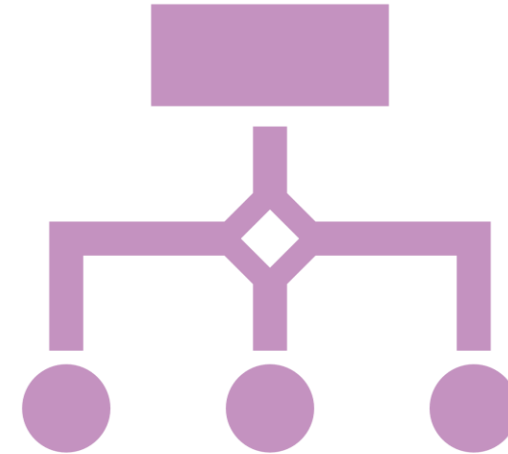
Estructuras de Control y Ciclos

Las estructuras de control y los ciclos son herramientas esenciales para controlar el flujo de ejecución en un programa



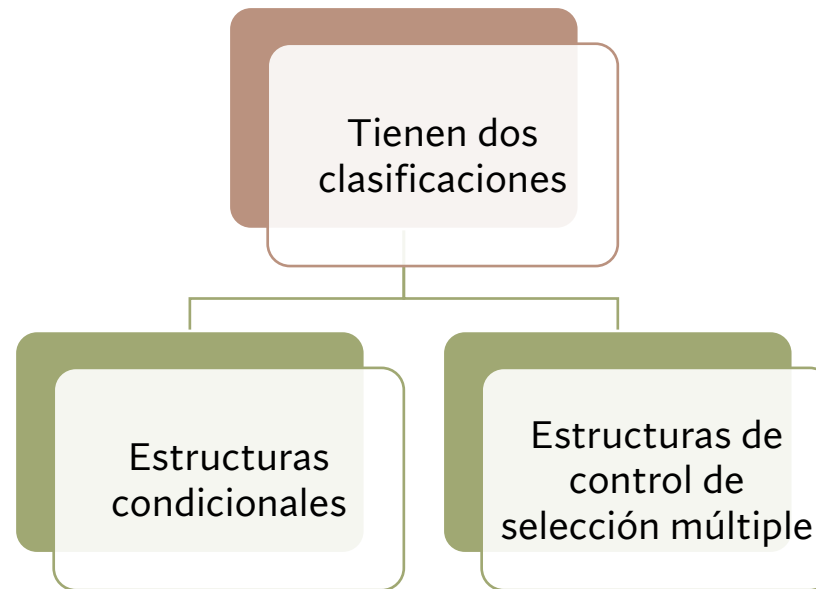
Determinan el flujo de ejecución de un programa y permiten la toma de decisiones basadas en condiciones, ejecutando bloques de código de manera selectiva.

En términos generales, las estructuras de control se dividen en estructuras condicionales y estructuras de selección múltiple. Dichas estructuras permiten la ejecución de diferentes bloques de código dependiendo de si es verdadera o no una condición, haciendo del programa más flexible y dinámico.



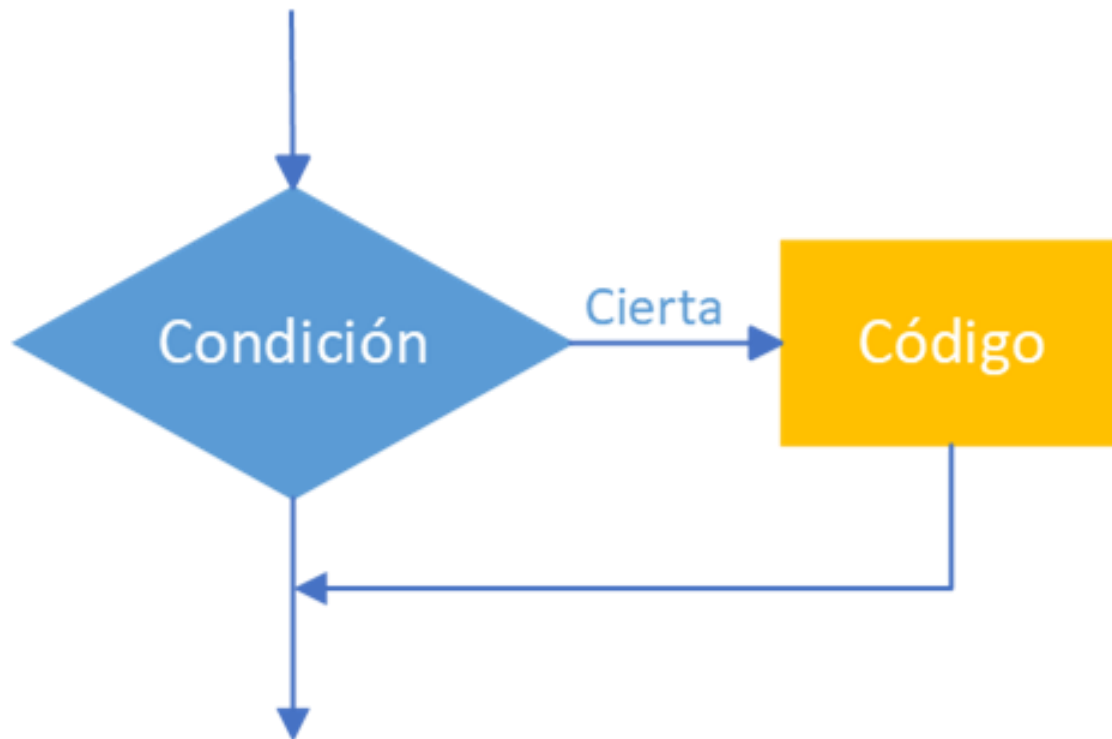
Estructuras de Control

Estructuras de Control



Estructuras de Control Condicional

Estructura Condicional simple



Las estructuras condicionales permiten evaluar expresiones lógicas y tomar decisiones en función de si dichas expresiones se evalúan como true o false.

IF

La estructura if es la más básica y evalúa una única condición. Si la condición es verdadera (true), se ejecuta el bloque de código dentro del if. Si la condición es falsa (false), el código dentro del if se omite.

```
if (<condicion>) {  
    // Ejecuta el código si se cumple la condición  
}  
  
int numeroGatos = 15;  
if (numeroGatos == 15) {  
    System.out.println("La cantidad de gatos es 15");  
}
```

IF-ELSE

Condición doble

Cuando se requiere evaluar una condición y ejecutar un bloque de código si es verdadera, pero otro bloque si es falsa, se utiliza if-else.

```
if (<condicion>) {  
    // Ejecuta el código si se cumple la condición  
} else {  
    /*  
        Ejecuta código de este bloque en caso  
        de no cumplir la condición inicial  
    */  
}  
  
int numeroGatos = 15;  
if (numeroGatos == 10) {  
    System.out.println("La cantidad de gatos es 10");  
} else {  
    System.out.println("La cantidad de gatos no es de 10")  
}
```

ELSE IF

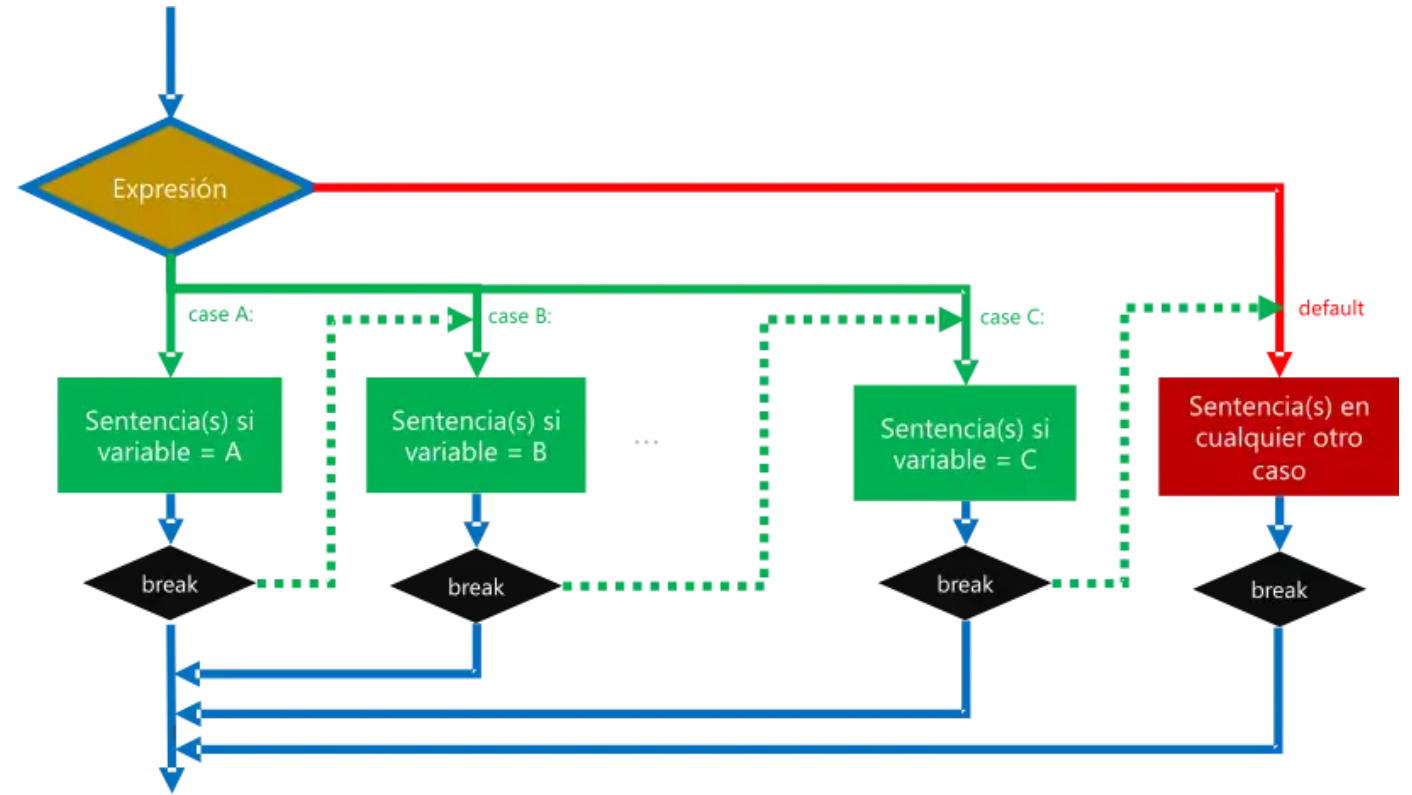
Múltiples condiciones

Esta estructura permite evaluar varias condiciones en cadena y ejecutar diferentes bloques de código dependiendo de cuál condición se cumpla primero.

```
if (<condicion>) {  
    /*  
     Ejecuta codigo en caso de cumplir <condicion>  
    */  
} else if (<condicion_2>){  
    /*  
     En caso de no cumplir <condicion>  
     Si <condicion_2> se cumple se ejecutará  
     codigo de este bloque  
    */  
} else {  
    /*  
     En caso de no cumplir ninguna condicion  
     Ejecuta código de bloque else  
    */  
}  
  
int nota = 62;  
if (nota > 60) {  
    System.out.println("El estudiante gana el curso");  
} else if (nota == 61) {  
    System.out.println("El estudiante gana raspado el curso");  
} else {  
    System.out.println("El estudiante reprobó el curso");  
}
```

Estructuras de Control de Selección Múltiple

Las estructuras de selección múltiple permiten evaluar múltiples valores posibles de una variable y ejecutar diferentes bloques de código según el valor que tome.



```
switch (<variable>) {  
    case valor1:  
        // Ejecuta código si <variable> == valor1  
        break;  
    case valor2:  
        // Ejecuta código si <variable> == valor2  
        break;  
    default:  
        /*  
        Ejecuta el código en caso que no coincida <variable>  
        con algún valor  
        */  
}
```

```
char clasificacionPersonaje = 'A';  
switch (clasificacionPersonaje) {  
    case 'S':  
        System.out.println("Personaje con rareza máxima");  
        break;  
    case 'A':  
        System.out.println("Personaje rareza media");  
        break;  
    case 'B':  
        System.out.println("Personaje común");  
        break;  
    default:  
        System.out.println("Sin personaje (toco arma pato)");  
}
```

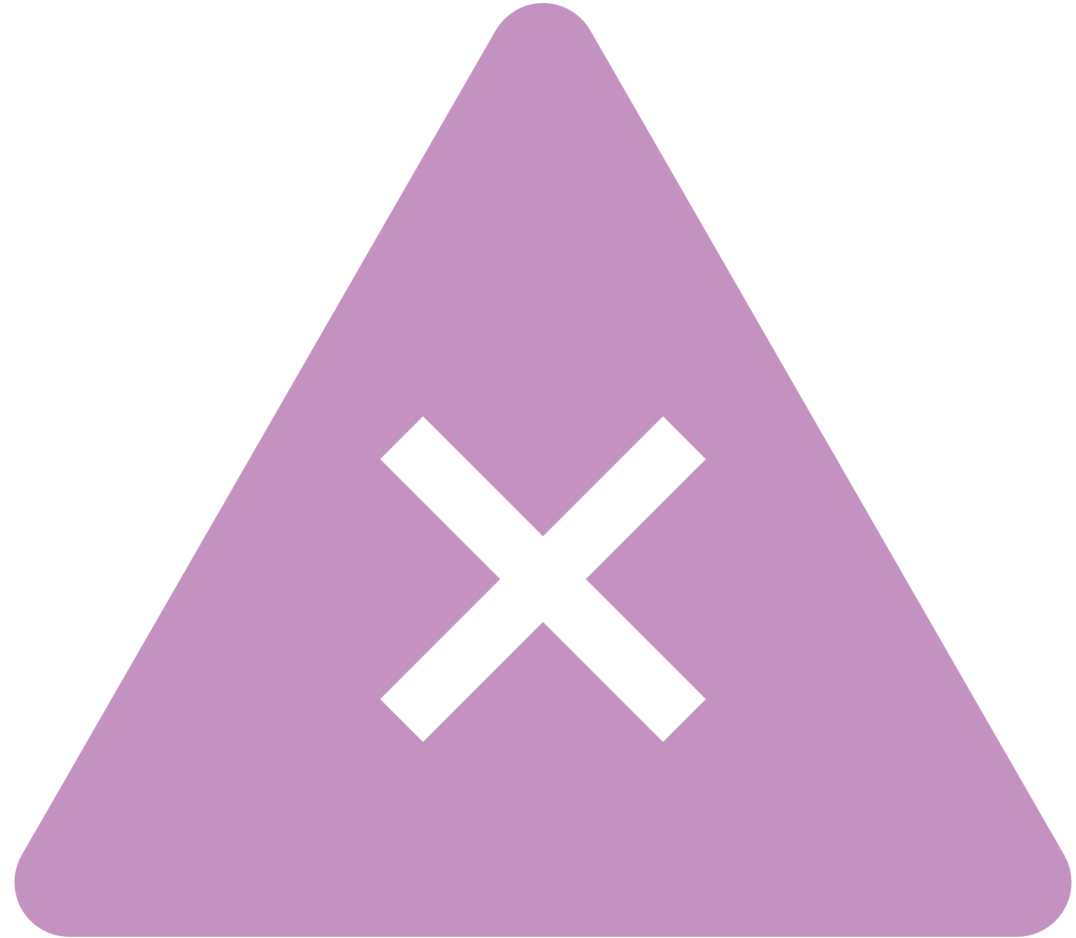
SWITCH

El switch es una alternativa más eficiente al if-else if cuando se trata de evaluar una sola variable contra múltiples valores específicos.

Importancia del break

Switch

Cada *case* dentro de un switch debe terminar con *break* para evitar la ejecución en cascada de los casos siguientes. Si se omite *break*, el programa ejecutará todos los bloques siguientes hasta encontrar un *break* o llegar al final.

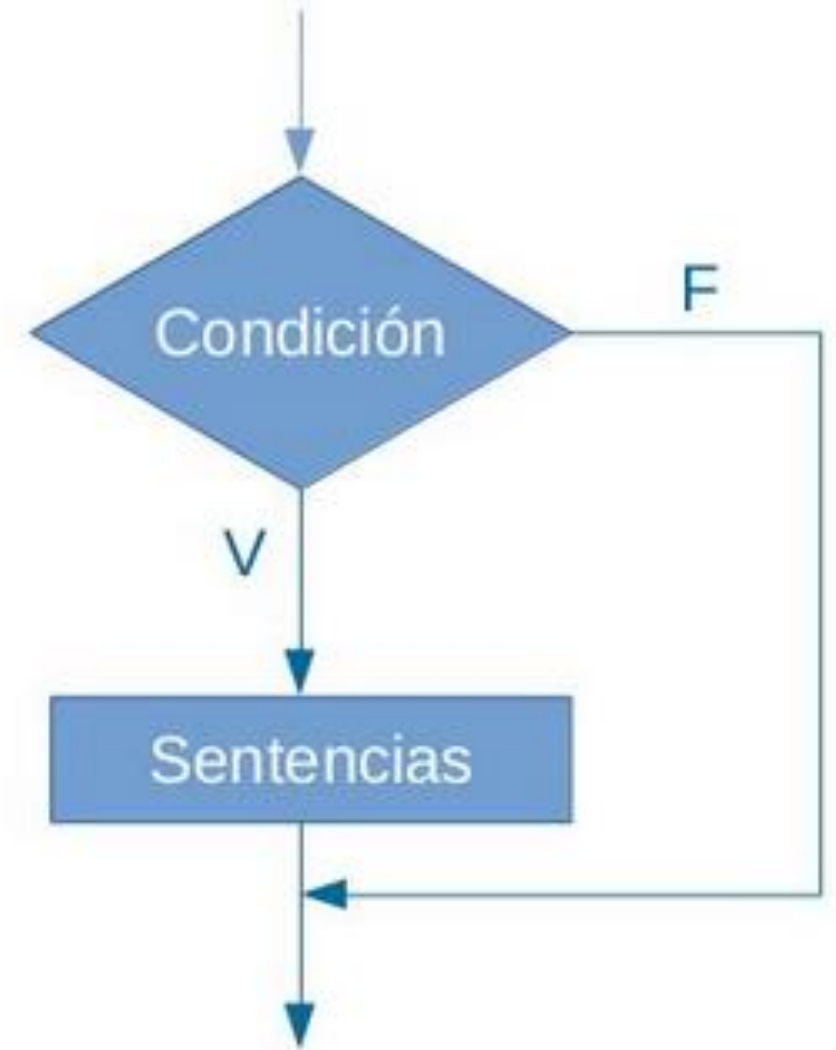


Característica	IF/ELSE IF/ELSE	Switch
Condiciones	Evalúa expresiones booleanas complejas	Evalúa igualdad con valores específicos
Uso Principal	Comparaciones y condiciones múltiples	Comparación directa de valores concretos
Tipo de Datos	Funciona con cualquier tipo de comparación (<, >, ==, !=, etc.)	Solo compara igualdad (==) en enteros, caracteres y String
Legibilidad	Menos legible si hay muchas condiciones	Más limpio y organizado para múltiples opciones

Diferencias IF/ELSE IF & SWITCH

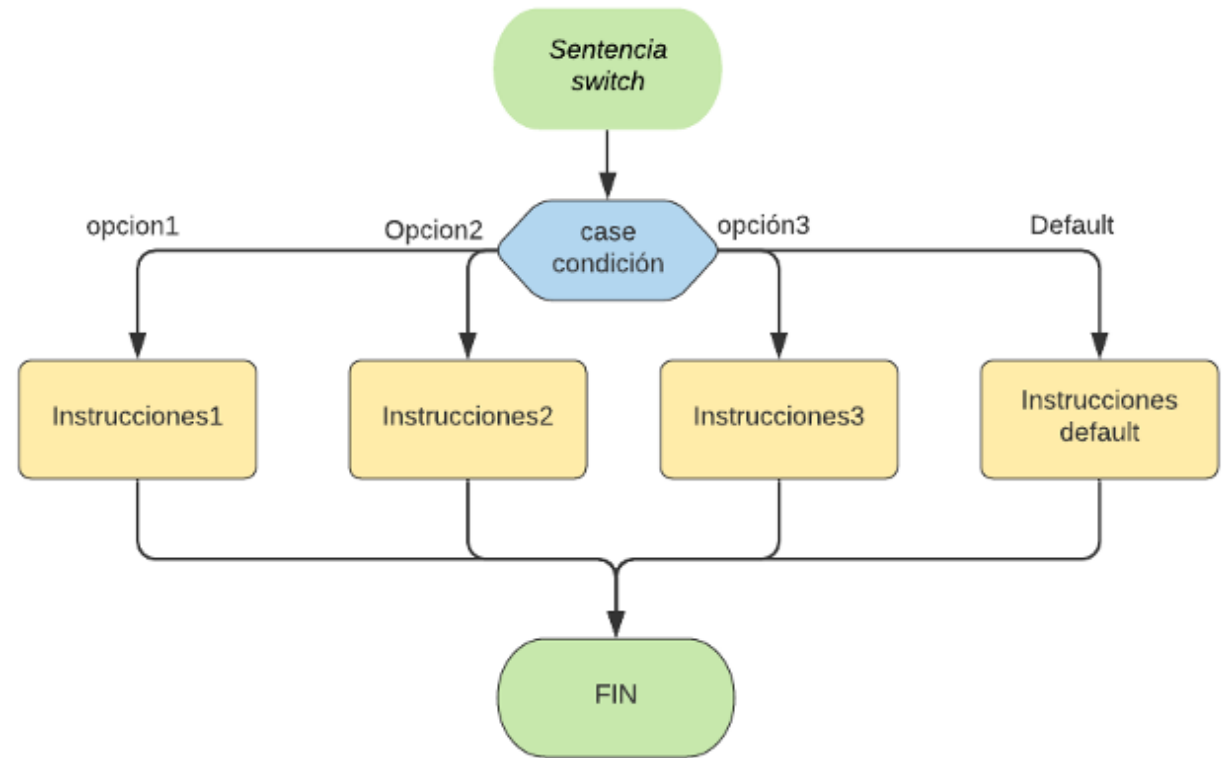
Cuando usar IF/ELSE IF/ELSE

- Cuando las condiciones son rangos (<, >, >=, <=).
- Cuando hay expresiones booleanas complejas.
- Cuando hay múltiples variables involucradas.



Cuando usar SWITCH

- Cuando se evalúa una variable contra valores específicos (int, char, String).
- Cuando hay muchas opciones y se quiere mejor legibilidad.
- Cuando se busca eficiencia en la ejecución.



Ciclos

Estructuras Iterativas

Los ciclos permiten repetir un bloque de código mientras se cumpla una condición o hasta que un conjunto de datos haya sido procesado.



WHILE

El ciclo while ejecuta un bloque de código mientras una condición dada sea verdadera (true). La condición se evalúa antes de cada iteración, por lo que si la condición es falsa desde el principio, el bloque no se ejecutará ni una sola vez.

```
while (<condicion>) {  
    // Ejecuta el código mientras cumpla <condición>  
}  
  
int valor = 0;  
while (valor < 5) {  
    System.out.println("Valor: " + valor);  
    valor++;  
}
```

Cuando usar While



Cuando no sabes cuántas veces se ejecutará el ciclo.



Cuando la condición de salida depende de algún cálculo o entrada dinámica.

DO-WHILE

El ciclo do-while es similar al ciclo while, pero con una diferencia clave: el bloque de código se ejecuta al menos una vez, ya que la condición se evalúa después de la primera iteración.

```
do {  
    // Código que se va a ejecutar hasta cumplir <condicion>  
} while (<condicion>);  
  
int valor = 0;  
do {  
    System.out.println("Valor: " + valor);  
    valor++;  
} while (valor < 5);
```

Cuando usar Do While

Cuando necesitas que el bloque de código se ejecute al menos una vez, sin importar si la condición inicial es verdadera o falsa.

```
int contador = 5;  
do {  
    System.out.println("Iteración: " + contador);  
    contador++;  
} while (contador < 5);
```


FOR

El ciclo for es una estructura más compacta que combina la inicialización, la condición y el incremento/decremento en una sola línea. Es ideal para cuando conoces de antemano cuántas iteraciones se realizarán.

```
for(<inicio>; <final>; <inc/dec>) {  
    // Ejecuta el código <final> - <inicio> veces  
}  
  
for (int i = 0; i < 5; i++) {  
    System.out.println("Valor: [ " + i + " ]");  
}
```

Cuando usar For



Cuando sabes exactamente cuántas veces se debe ejecutar el ciclo.



Para recorrer rangos numéricos o índices de arreglos.

FOR-EACH

El ciclo for-each es una versión simplificada del ciclo for que se utiliza para recorrer colecciones (como listas o arrays) sin necesidad de manejar índices explícitamente.

```
for (Tipo variable : coleccion) {  
    // Código ejecutar en la iteración  
}  
  
int[] numeros = {1, 2, 3, 4, 5};  
for (int numero : numeros) {  
    System.out.println(numero);  
}
```

Cuando usar For-Each



Cuando necesitas recorrer todos los elementos de una colección o array sin preocuparte por los índices.



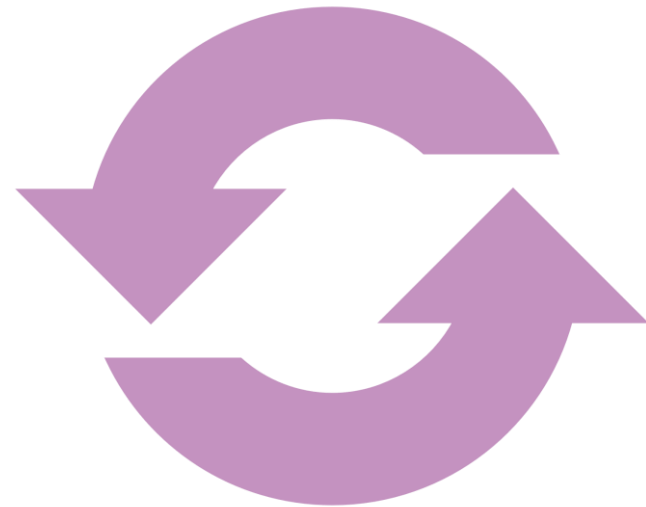
Para mejorar la legibilidad del código.

Tabla de Uso

Característica	While	Do-While	For	For-Each
Condición a Evaluar	Antes de cada iteración	Después de cada iteración	Antes de cada iteración	No aplica (Recorre todos los elementos)
Mínimo de Iteraciones	0	1	0	Igual al número de elementos
Uso común	Condiciones dinámicas	Tener como mínimo una iteración	Iteraciones controladas por índices	Recorrido de colecciones o arrays

Control de Ciclos

- **break:** Termina el ciclo de forma inmediata.
- **continue:** Salta a la siguiente iteración del ciclo.
- **return:** Termina el ciclo y el método donde se encuentra.



Excepciones

¿Qué es una Excepción? ¿Por qué Surgen? ¿Cómo manejarlas?

Excepción

Ante una incoherencia lógica que ocurre durante la ejecución de un programa informático que no puede resolverse, se genera una excepción que finaliza la ejecución de una sentencia de control, una subrutina o el mismo programa.

Manejo de Excepciones

El manejo de errores y excepciones es un concepto que surge de la necesidad de poder indicar cuando ocurre un error en la ejecución de un programa. Estos errores son en su totalidad, debidos a la semántica del programa. Esta técnica permite crear aplicaciones tolerantes a fallas. En Java, suelen ser clases que extienden el paquete Throwable.

Prevención de Errores

- Conocer el lenguaje
- Buenas prácticas de programación
- Flujo de aplicación definido
- Revisión, mantenimiento, pruebas

TRY

La sentencia de control “try” se utiliza en Java para indicar que podría ocurrir una excepción en la ejecución de sus subsistencias. Cuando efectivamente, ocurre un problema irrecuperable en la ejecución de las instrucciones dentro de el segmento “try”, Java desplaza la ejecución hacia la función “catch” correspondiente a la excepción que ocurrió. Java logra esto a través de un método conocido como “limpieza de pila”.

CATCH

Toda sentencia “try” en Java debe incluir al menos una sentencia “catch”. Este tipo de sentencia recibe como parámetro una única excepción, la cual funciona como variable local en el ámbito de sus sentencias. Esta excepción puede ser específica o generalizada (Exception), y adquirir cualquier identificador.

FINALLY

Esta clausula es opcional, y se ejecuta al finalizar la sentencia “try” o “catch”, es decir: Si las sentencias “try” no generan excepción, el flujo de la aplicación continúa hacia las sentencias “finally”; y si se transfiere el flujo a una sentencia “catch”, el flujo de la aplicación continúa hacia las sentencias “finally” .



Debuggin

Bug

Los fallos y anomalías de funcionamiento en un sistema o programa informático que provocan resultados indeseados son comúnmente denominados como 'bugs', un término en inglés cuya traducción al español sería 'bicho'.



El primer bug en la historia de la computación


El incidente de 1947:

- La historia más famosa sobre el origen del término en informática ocurrió el 9 de septiembre de 1947, cuando los ingenieros que trabajaban en la computadora Mark II Aiken Relay Calculator (un sistema basado en relés) encontraron un problema técnico.
- Al inspeccionar el hardware, descubrieron que una polilla (bug) había quedado atrapada en uno de los relés, causando un mal funcionamiento.
- El equipo, liderado por Grace Hopper (una pionera de la computación, madre de los compiladores y creadora del primer manual de programación), retiró la polilla y documentó el incidente en el diario de mantenimiento, pegando el insecto en la página junto con la anotación: "First actual case of bug being found."

9/9

0800 Antan started
1000 " stopped - antan ✓ { 1.2700 9.0
1300 (032) HP - MC ~~1.582647000~~ 9.0
(033) PRO 2 2.130476415
convd 2.130676415
Relays 6-2 in 033 failed special spec
in Relay " 11.00 test

1100 Relays changed
Started Cosine Tape (Sine check)
1525 Started Multi-Adder Test.

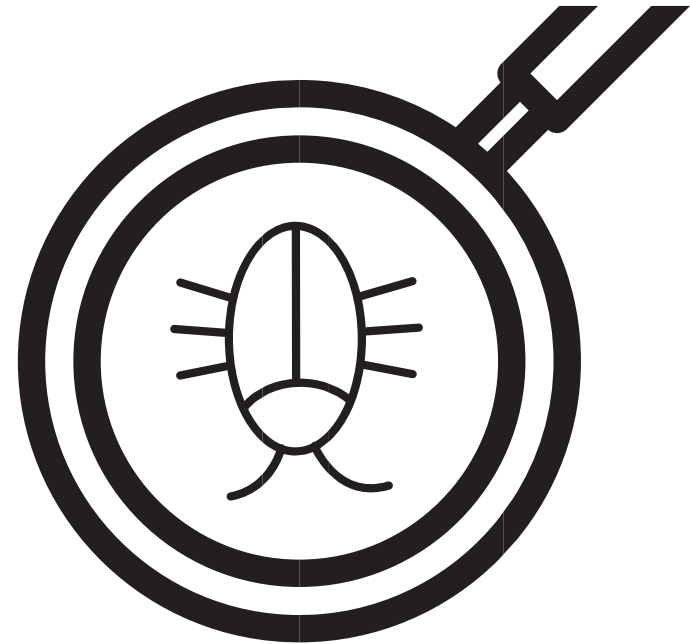
1545  Relay #70 Pan
(moth) in relay.

1630 Antan started.
1700 closed down.

First actual case of bug being found.

Importancia del suceso

Aunque el término ya se usaba en ingeniería, este incidente popularizó la palabra "bug" en el ámbito de la informática. Desde entonces, "debugging" (depuración) se convirtió en el término para describir el proceso de identificar y corregir errores en software o hardware.



Debuggin

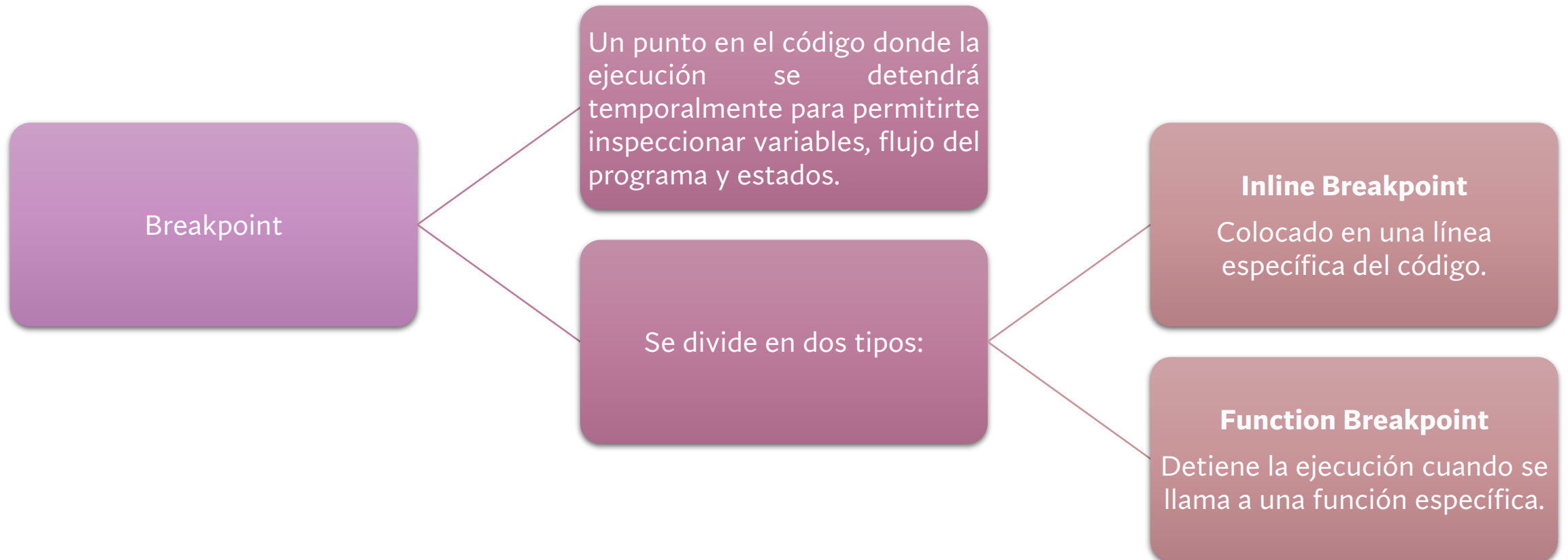
Su objetivo es encontrar errores que pueden impedir que los códigos funcionen de forma adecuada. Con este, es posible determinar lo que está ocurriendo dentro del código fuente y obtener sugerencias de acciones para mejoras.

Cada instrumento de desarrollo tiene su propia herramienta para realizar debug al código. A través de ellas, se puede determinar los puntos de parada, conocidos como break points, para verificar el estado actual de la aplicación.

También puedes acompañar el contenido de una determinada variable. Todo esto sirve para facilitar el agotador trabajo de encontrar un bug en un sistema.



Conceptos



Controles de Ejecución

Start/Resume:

- Inicia o reanuda la ejecución del programa hasta el siguiente breakpoint.

Pause:

- Detiene temporalmente la ejecución del programa en cualquier punto.

Step Over:

- Ejecuta la siguiente línea de código sin entrar en métodos llamados.

Step Into:

- Entra en el método llamado en la línea actual.

Step Out:

- Sale del método actual y regresa al punto donde fue llamado.

Stop:

- Termina la sesión de debugging.

Dudas o Comentarios

