# TEMA 2: USO DE ESTRUCTURAS DE CONTROL

1. **Estructuras de selección.**

    **1.1 Estructura If.**
    **1.2 Switch.**

2. **Estructuras de repetición.**

    **2.1 Bucle While.**
    **2.2 Bucle Do-While.**
    **2.3 Bucle For.**

3. **Control de excepciones.**

# 1.-Estructuras de selección.

## 1.1 Estructura If.

Permite ejecutar una instrucción (o secuencia de instrucciones) si se da una condición.
La sentencia **if** es la sentencia básica de selección. Existen tres variantes, selección simple, selección doble y selección múltiple, pero las 3 se basan en la misma idea.

a) Selección **Simple**. Su sintaxis es la siguiente:

```
if (condición){
   sentencias;
}
```

Donde:

- **condición** es una expresión booleana
- **sentencias** representa un bloque de sentencias o una sentencia única
- Si es una sentencia única, se pueden quitar las llaves

Ej:

```
if (numero%2!=0){
   System.out.println("El numero es impar ");
   cont++;
}
```

b) **Selección Doble**. Se añade una parte else a la sentencia if la cual se ejecutara si la condición es falsa. Su sintaxis es:

```
if (condición){
   sentencias1;
}
else{
   sentencias2;
}
```

Ej:

```
if (a>b)
   System.out.println ("El número mayor es  " + a);
else
   System.out.println("El número mayor es  " + b+ "  o son iguales");
```

c) **Selección Múltiple**. Es habitual cuando hay más de una condición. Su sintaxis es la siguiente:

```
if (condición1){
   sentencias1;
}
else if (condición2){
   sentencias2;
}
else if (condición3){
   sentencias3;
else{
  sentencias;
}
```

Ej:

```
if (mes == 12 || mes == 1 || mes == 2)
   System.out.println ( "Invierno");
else if (mes == 3 || mes == 4 || mes == 5)
   System.out.println( "Primavera");
else if (mes == 6 || mes == 7 || mes == 8)
    System.out.println ("Verano");
else
    System.out.println ("Otoño");
```

## 1.2 Switch.

Es una estructura de selección múltiple más cómoda de leer y utilizar que el else-if. Selecciona un bloque de sentencias dependiendo del valor de una expresión. Su sintaxis es:

```
switch (expresion) {
   case valor1:sentencias;
               break;
   case valor2:sentencias;
               break;
   case valor3:sentencias;
               break;
        ...
   default: sentencias;
            break;
}
```

Donde:

- Expresión tiene que tomar un valor entero o un carácter.
- Break indica que ha acabado la ejecución de ese caso y seguiría ejecutando la sentencia siguiente al switch.
- Default es opcional y se ejecutara cuando la expresión tome un valor que no esté recogido en los distintos casos especificados.

Ej:

```
switch (mes) {
        case 4:
        case 6:
        case 9:
        case 11: dias = 30;
                  break;
        case 2: dias = 28;
                 break;
        default: dias = 31;
                  break;
}
```

Es posible juntar distintos casos, dejándolos en blanco y especificando las instrucciones en el último de los casos de grupo.
Por ejemplo, si la variable mes toma los valores 4, 6, 9 y 11 se realiza la misma sentencia, se le asigna a la variable dia el valor 30.

## 2.- Estructuras de repetición.

Estas estructuras se utilizan para repetir un bloque de sentencias mientras se cumpla una condición. Son también llamadas sentencias de iteración o bucles.
Los tipos de bucles son: while, do-while y for .

### 2.1 Bucle While: La sintaxis de la sentencia es:

```
while (condición){
sentencias;
}
```

Donde:

- **condición** es una expresión booleana que se evalúa al principio del bucle y antes de cada iteración de las sentencias.
- Si la condición es verdadera, se ejecuta el bloque de sentencias, y se vuelve al principio del bucle.
- Si la condición es falsa, no se ejecuta el bloque de sentencias, y se continúa con la siguiente sentencia del programa.
- Si la condición es falsa desde un principio, entonces el bucle nunca se ejecuta.
- Si la condición nunca llega a ser falsa, se tiene un bucle infinito.

Ej:

```
int i = 0;
while (i < 10) {
   System.out.println (i);
    i++;
}
```

**2.2 Bucle Do-While:** La sintaxis de la sentencia es:

```
do {
   sentencias;
} while (condición);
```

La sentencia es muy parecida a while. El bloque de sentencias se repite mientras se cumpla la condición.
La condición se comprueba después de ejecutar el bloque de sentencias.
El bloque se ejecuta siempre al menos una vez.

Ej:

```
int i=0;
do{
   System.out.println (i);
   i++;
}while (i<10);
```

**2.3 Bucle For:** La sintaxis de la sentencia es:

```
for (inicialización;condición; incremento){
  sentencias;
}
```

Donde

- La inicialización se realiza sólo una vez, antes de la primera iteración.
- La condición se comprueba cada vez antes de entrar al bucle. Si es cierta, se entra. Si no, se termina.
- El incremento se realiza siempre al terminar de ejecutar la iteración, antes de volver a comprobar la condición.
- El incremento puede ser positivo o negativo.

Ej:

```
int i;
for (i=0; i<10; i++)
    System.out.println (i);
```

## 3.- Control de excepciones.

Una excepción es un evento que ocurre durante la ejecución de un programa y detiene el flujo normal de la secuencia de instrucciones del programa.
El control de dichas excepciones se utiliza para la detección y corrección de errores. Si hay un error, la aplicación no debería "morirse".

Para manejar las excepciones en Java, se actúa de la siguiente manera:

- Se intenta (try) ejecutar la sentencia o bloque de sentencias que pueden producir algún error.
- Se captura (catch) las posibles excepciones que se hayan podido producir, ejecutando una serie de sentencias que informen o intenten resolver el error.
- Finalmente (finally) se puede ejecutar una serie de sentencias tanto si se ha producido un error como si todo ha ido bien.

El formato es:

```
try {
    Sentencias que pueden producir error
}catch(ClaseExcepción variableRecogeExcepción){
    Sentencias que informan o procuran solucionar
    el error.
    La variable variableRecogeExcepción no se
    tiene declarar antes.
}catch(ClaseExcepción2 variableRecogeExcepción2){
    Puede haber varios catch. Uno para cada tipo
    de Exception.
}finally {
    Sentencias que deben ejecutarse en cualquier
    caso (opcional)
}
```

El elemento **ClaseExcepción** que aparece junto a catch, debe ser una de las clases de excepción. Al generarse el error durante la ejecución se puede comprobar qué clase de excepción se ha producido. De forma general, la clase **Exception** recoge todos los tipos de excepciones. Si se desea un control más exhaustivo del tipo de error que se produce, se debe concretar la clase de excepción correspondiente.

Por ejemplo, cuando se intenta convertir al tipo de dato numérico entero un dato introducido por el usuario en un campo de texto se utiliza una sentencia como:

```
int num = Integer.valueOf(campoNúmero.getText());
```

Si el valor introducido no es numérico, sino una cadena de caracteres, la llamada a Integer.valueOf produce una excepción, como se puede apreciar en la salida estándar:


Se puede apreciar que se produce una excepción del tipo NumberFormatException, por tanto se debería capturar esa excepción para controlar el error.

```
try {
  int num = Integer.valueOf(campoNúmero.getText());
} catch(NumberFormatException e) {
  System.out.println("Error: El valor indicado no es
un número");
}
```

Si no se tiene claro el tipo de excepción que se quiere controlar, o se quiere controlar cualquier tipo de excepción que se pueda producir, se indica en el catch la clase Exception que es genérica para todas las excepciones.

```
try {
  int num = Integer.valueOf(campoNúmero.getText());
} catch(Exception e) {
  System.out.println("Se ha detectado un error");
}
```

La variable que se indique en catch(), recoge información sobre la excepción que se ha producido. Es muy frecuente que a dicha variable se le asigne el nombre e. Si se desea mostrar qué tipo de error se ha producido se puede utilizar el método getMessage o el método toString (similar a getMessage pero además incluye el nombre de la clase de excepción) sobre la variable que se ha utilizado en el catch:

```
System.out.println(e.getMessage());
System.out.println(e.toString());
```

Ej:

```
int dividendo=10, divisor=0, cociente;
try {
    cociente = dividendo/divisor;
} catch (ArithmeticException e) {
        System.out.println("Error: Division por 0");
        cociente = 0;
}

// en cualquier caso el programa continúa por aquí
```