UD 2
Uso de estructuras de control

RESULTADOS DE APRENDIZAJE Y CRITERIOS DE EVALUACIÓN

RA3. Escribe y depura código, analizando y utilizando las estructuras de control del lenguaje.

Criterios de evaluación

a) Se ha escrito y probado código que haga uso de estructuras de selección.
b) Se han utilizado estructuras de repetición.
c) Se han reconocido las posibilidades de las sentencias de salto.
d) Se ha escrito código utilizando control de excepciones.
e) Se han creado programas ejecutables utilizando diferentes estructuras de control.
f) Se han probado y depurado los programas.
g) Se ha comentado y documentado el código.

INDICE

1. Introducción

Hasta ahora las instrucciones que hemos visto con Java, son instrucciones que se ejecutan secuencialmente.

Las **instrucciones de control de flujo** permiten alterar esta forma de ejecución. A partir de ahora habrá líneas en el código que se ejecutarán o no dependiendo de una condición.

Esa condición se construye utilizando lo que se conoce como **expresión lógica**. Una expresión lógica es cualquier tipo de expresión Java que dé un resultado **booleano** (verdadero o falso).

Las expresiones lógicas se construyen por medio de variables booleanas o bien a través de los operadores relacionales (==, >, <,…) y lógicos (&&, | |, !).

2. Estructuras de selección

1.1 IF – Sentencia condicional simple

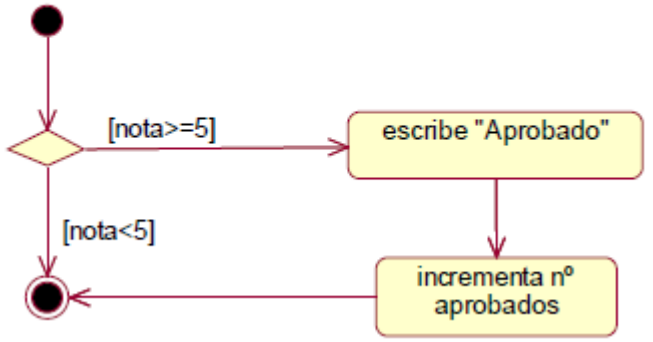
if – Sentencia condicional simple

- Se trata de una sentencia que, tras evaluar una expresión lógica, ejecuta una serie de instrucciones en caso de que la expresión lógica sea verdadera. Si la expresión tiene un resultado falso, no se ejecutará ninguna expresión. Su sintaxis es:

```
if(expresión lógica)
{
        instrucciones
        ….
}
```

- Las llaves se requieren sólo si va a haber varias instrucciones. En otro caso se puede crear el if sin llaves:
  - if(expresión lógica) sentencia;
- Ejemplo:

```
if(nota>=5)
{
    System.out.println("Aprobado");
    aprobados++;
}
```



1.2 IF – Sentencia condicional compuesta

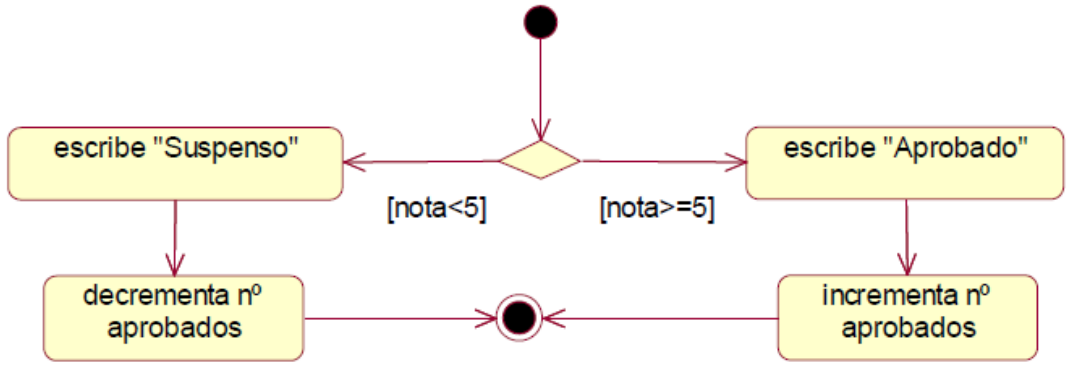
if – Sentencia condicional compuesta

- Es igual que la anterior, sólo que se añade un apartado else que contiene instrucciones que se ejecutarán si la expresión evaluada por el if es falsa. Sintaxis:

```
if(expresión lógica)                    instrucciones
{                                       …
        instrucciones                 }
        ….
}
else
{
```

- Como en el caso anterior, las llaves son necesarias sólo si se ejecuta más de una sentencia. Ejemplo de sentencia if-else:

```
if(nota>=5)
{
    System.out.println("Aprobado");
    aprobados++;
}
else
{
    System.out.println("Suspenso");
    suspensos++;
}
```



1.3 IF – Anidación

if – Anidación

```
if (x==1)
{
        Instrucciones
        …
}
else
{
        if(x==2)
        {
                instrucciones
                …
         }
         else
         {
                if(x==3)
                {
                        instrucciones
                        …
                 }
          }
}
```

- Dentro de una sentencia if se puede colocar otra sentencia if. A esto se le llama **anidación** y permite crear programas donde se valoren expresiones complejas. La nueva sentencia puede ir tanto en la parte if como en la parte else.
- Las anidaciones se utilizan muchísimo al programar. Sólo hay que tener en cuenta que siempre se debe cerrar primero el último if que se abrió. Es muy importante también tabular el código correctamente para que las anidaciones sean legibles.
- El código podría ser:
- Una forma más legible de escribir ese mismo código sería:

dando lugar a la llamada instrucción **if-else-if.**

```
if (x==1)
{
      instrucciones
      …
}
else if (x==2)
{
       instrucciones
       …
}
else if (x==3)
{
       instrucciones
       …
}
```

1.4 El operador ternario ?:

El operador ternario ?:

- El operador ternario ?: devuelve un valor que se selecciona entre dos posibles.
- La selección dependerá de la evaluación de una expresión relacional o lógica: es decir, que puede tomar dos valores, verdadero o falso.
- El operador tiene la siguiente sintaxis:

expresiónCondicional ? valor1 : valor2

La evaluación de la expresión decidirá cual de los dos posibles valores se devuelve: en el caso de que la **expresión** resulte **verdadera** se devuelve **valor1**, y cuando resulte **falsa** devuelve **valor2**.

Ejemplo

```
Int a, b;

a=3<5 ? 1 : -1; // 3<5 es verdadero: a toma el valor 1
b=a==7 ? 10 : 20; // a (que vale 1) == 7 es falso: b toma el valor 20
```

TAREA 01. Hoja de ejercicios 1.

Estructura condicional IF

Realiza la tarea correspondiente que encontrarás en la plataforma educativa.

Realiza la entrega de esta tarea de acuerdo con las instrucciones que se indiquen.

1.4 SWITCH

switch

- Se la llama estructura condicional compleja porque permite evaluar varios valores a la vez. En realidad, sirve como sustituta de algunas expresiones de tipo if-else-if.
- Sintaxis:

```
switch (expresiónEntera)
{
        case valor1:
                instrucciones del valor 1
        break;
        case valor2:
                instrucciones del valor 2
        break;
                …
        default:
        /*instrucciones que se ejecutan si la expresión no toma
        ninguno de los valores anteriores*/

}
```

- Esta instrucción evalúa una expresión (que debe ser **short, int, byte o char**), y **según el valor** de la misma ejecuta instrucciones. Cada case contiene un valor de la expresión; si efectivamente la expresión equivale a ese valor, **se ejecutan las instrucciones de ese case y de los siguientes** (incluso default).

switch

- La instrucción **break** se utiliza para salir del switch. De tal modo que si queremos que para un determinado valor se ejecuten las instrucciones de **un apartado case y sólo las de ese** apartado, entonces habrá que finalizar ese case con un break.
- El bloque **default** sirve para ejecutar instrucciones para los casos en los que la expresión **no se ajuste a ningún case.**

```
switch (diasemana)
{
    case 1:
        texto="Lunes";
    break;
    case 2:
        texto="Martes";
    break;
    case 3:
        texto="Miércoles";
    break;
    case 4:
        texto="Jueves";
    break;
    case 5:
        texto="Viernes";
    break;
    case 6:
      texto="Sábado";
    break;
    case 7:
        texto="Domingo";
    break;
    default:
        texto="?";
}
```

```
switch (diasemana)
{
    case 1:
    case 2:
    case 3:
    case 4:
    case 5:
        laborable=true; break;
    case 6:
    case 7:
        laborable=false;
}
```

TAREA 02. Hoja de ejercicios 2.

Estructura condicional Switch

Realiza la tarea correspondiente que encontrarás en la plataforma educativa.

Realiza la entrega de esta tarea de acuerdo con las instrucciones que se indiquen.

3. Estructuras de repetición

3.1 While

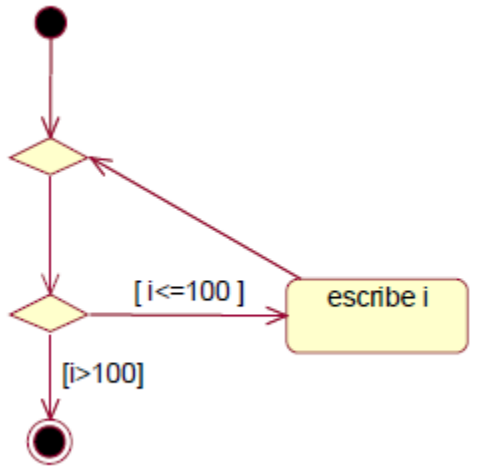
while

- La instrucción **while** permite crear **bucles**. Un bucle es un conjunto de sentencias que se repiten mientras se cumpla una determinada condición. Los bucles while agrupan instrucciones las cuales se ejecutan continuamente hasta que una condición que se evalúa sea falsa.
- La **condición se mira antes** de entrar dentro del while y cada vez que se termina de ejecutar las instrucciones del while
- Sintaxis:

```
while (expresión lógica)
{
sentencias que se ejecutan si la condición es true
}
```

- El programa se ejecuta siguiendo estos pasos:
  1) Se evalúa la expresión lógica
  2) Si la expresión es verdadera ejecuta las sentencias, sino el programa abandona la sentencia while
  3) Tras ejecutar las sentencias, volvemos al paso 1
- Ejemplo (escribir números del 1 al 100):

```
int i=1;
while (i<=100)
{
        System.out.println(i);
        i++;
}
```



while

Bucles con contador

- Se llaman así a los bucles que se repiten una serie determinada de veces. Están controlados por un **contador** (o incluso más de uno). El contador es una variable que va variando su valor (de uno en uno, de dos en dos,…) en cada vuelta del bucle. Cuando el contador alcanza un límite determinado, entonces el bucle termina.
- En todos los bucles de contador necesitamos saber:
  1. **Lo que vale la variable contadora** al principio. Antes de entrar en el bucle
  2. **Lo que varía** (lo que se incrementa o decrementa) el contador en cada vuelta
  3. Las **acciones** a realizar **en cada vuelta** del bucle
  4. El **valor final del contador**. En cuanto se rebase el **bucle termina**. Dicho valor se pone como condición del bucle, pero a la inversa; es decir, la condición mide el valor que tiene que tener el contador para que el bucle se repita y no para que termine.

```
i=10; //Valor inicial del contador, empieza valiendo 10 (por supuesto i debería estar declarada como entera, int)
while (i<=200)
{  //condición del bucle, mientras i sea menor de 200, el bucle se repetirá, cuando i rebase este valor, el bucle termina
      System.out.println(i); //acciones que ocurren en cada vuelta del bucle.
                             //En este caso simplemente escribe el valor del contador
      i+=10;                 //Variación del contador, en este caso cuenta de 10 en 10
}
```

while

Bucles con centinela

- Es el segundo tipo de bucle básico. Una **condición lógica** llamada **centinela**, que puede ser desde una simple variable booleana hasta una expresión lógica más compleja, sirve para decidir si el bucle se repite o no. De modo que cuando la condición lógica se incumpla, el bucle termina.
- Esa expresión lógica a cumplir es lo que se conoce como centinela y normalmente la suele realizar una variable booleana.
- Ejemplo:

```
boolean salir=false; /* En este caso el centinela es una variable booleana que inicialmente vale falso */
int n;
while(salir==false)        // Condición de repetición: que salir siga siendo falso. Ese es el centinela.
{                          //También se podía haber escrito simplemente: while(!salir)
        n=(int) (Math.random()*500 +1); // Lo que se repite en el bucle:
        System.out.println(n);                    // calcular un número aletorio de 1 a 500 y escribirlo
        salir=(n%7==0);                           //El centinela vale verdadero si el número es múltiplo de 7
}
```

- Un bucle podría ser incluso mixto: de centinela y de contador.
- Por ejemplo: un programa que escriba números de uno a 500 y se repita hasta que llegue un múltiplo de 7, pero que como mucho se repite cinco veces. **¡¡CODIFÍCALO!!**

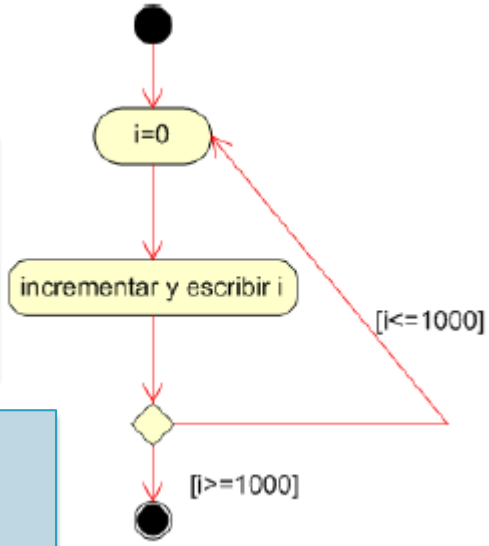
3.2 Do while

do while

- La única diferencia respecto a la anterior está en que la expresión lógica se evalúa después de haber ejecutado las sentencias. Es decir el bucle al menos se ejecuta una vez. Es decir los pasos son:
  1) Ejecutar sentencias
  2) Evaluar expresión lógica
  3) Si la expresión es verdadera volver al paso 1, sino continuar fuera del while

- Sintaxis:

```
do
{
        instrucciones
} while (expresión lógica); // es el único bucle que termina con ;
```



- Ejemplo (contar de uno a 1000):

```
int i=0;
do
{
     i++;
     System.out.println(i);
} while (i<1000);
```

- Se utiliza cuando al menos las sentencias del bucle se van a repetir una vez (en un bucle while puede que incluso no se ejecuten las sentencias que hay dentro del bucle si la condición fuera falsa, ya desde un inicio).
- De hecho cualquier sentencia do.while se puede convertir en while.
- INTENTA CODIFICAR EL EJEMPLO ANTERIOR UTILIZANDO UNA ESTRUCTURA WHILE

TAREA 03. Hoja de ejercicios 3.

Estructuras repetitivas While y Do While

Realiza la tarea correspondiente que encontrarás en la plataforma educativa.

Realiza la entrega de esta tarea de acuerdo con las instrucciones que se indiquen.

3.3 For

for

- Es un bucle más complejo especialmente pensado para rellenar arrays o para ejecutar instrucciones controladas por un contador. Una vez más se ejecutan una serie de instrucciones en el caso de que se cumpla una determinada condición.
- Sintaxis:

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

- Las sentencias se ejecutan mientras la condición sea verdadera. Además, antes de entrar en el bucle se ejecuta la instrucción de inicialización y en cada vuelta se ejecuta el incremento. Es decir, el funcionamiento es:
  1) Se ejecuta la instrucción de inicialización (sólo se hace una vez al principio)
  2) Se comprueba la condición
  3) Si la condición es cierta, entonces se ejecutan las sentencias. Si la condición es falsa, abandonamos el bloque for
  4) Tras ejecutar las sentencias, se ejecuta la instrucción de incremento y se vuelve al paso 2

for

- Ejemplo (contar números del 1 al 1000):

```
for(int i=1;i<=1000;i++)
{
    System.out.println(i);
}
```

- La ventaja que tiene es que el código se reduce. La desventaja es que el código es menos comprensible. El bucle anterior es equivalente al siguiente bucle while:

```
int i=1; /*sentencia de inicialización*/
while(i<=1000) /*condición*/
{
    System.out.println(i);
    i++; /*incremento*/
}
```

- Como se ha podido observar, es posible **declarar la variable contadora dentro del propio bucle** for. De hecho, es la forma habitual de declarar un contador. De esta manera se crea una variable **que muere en cuanto el bucle acaba**: es decir, el **ámbito** de dicha variable es el propio bucle.

TAREA 04. Hoja de ejercicios 4.

Estructura repetitiva For

Realiza la tarea correspondiente que encontrarás en la plataforma educativa.

Realiza la entrega de esta tarea de acuerdo con las instrucciones que se indiquen.

3.4 Bucles anidados

3.4 Bucles anidados

¿En que consisten?

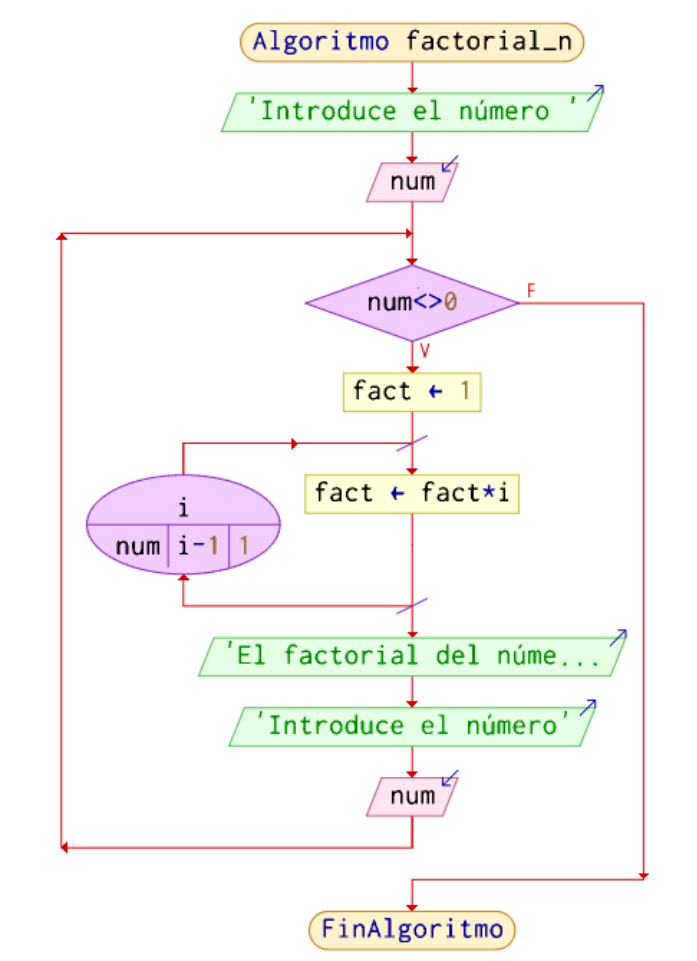
Se trata de usar una estructura de bucle dentro de otra ya existente.

Esta puede ser cualquiera de las estructuras repetitivas vistas hasta ahora.

- While
- Do
- For

Puedo anidar cualquier número de ellas unas dentro de otras, aunque lo más normal es no pasar de 3.

CALCULAR EL FACTORIAL DE N NÚMEROS INTRODUCIDOS POR TECLADO. La introducción finaliza al introducir el 0.




```
1  Algoritmo factorial_n
2      escribir "Introduce el número "
3      Leer  num
4      Mientras num ≠0 Hacer
5          fact = 1
6          Para i=num Hasta 1 Con Paso i-1 Hacer
7              fact = fact * i
8          Fin Para
9          Escribir "El factorial del número es: " fact
10         Escribir "Introduce el número"
11         Leer  num
12     Fin Mientras
13 FinAlgoritmo
```

En Java

```java
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package Bucles_anidados;
import java.util.Scanner;

/**
 *
 * @author
 *
 * Calcular el factorial de n números introducidos por teclado.
 * La introducción finaliza al introducir el 0.
 */
public class Bucle_1 {
    public static void main(String[] args) {
        Scanner entrada = new Scanner(System.in);
        int num, fact;

        //Intro datos
        System.out.print("Introduce el número: ");
        num=entrada.nextInt();

        //Calculo
        while (num !=0) {
            fact=1;
            for (int i = num; i > 0; i--) {
                fact*=i;
            }
            System.out.println("El factorial del número "+ num+" es: " +fact);

            //Vuelvo a pedir el número
            System.out.print("Introduce el número: ");
            num=entrada.nextInt();

        }
        System.out.println("************FIN************");
    }

}
```

DIBUJAR UN CUADRADO CON * CUYO NÚMERO DE * POR LADO SERÁ PEDIDO POR TECLADO

```
Output - EJEMPLOS (ru...
run:
Introduce el número de astericos por lado: 4
****
*  *
*  *
****
BUILD SUCCESSFUL (total time: 4 seconds)
```

```java
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package Bucles_anidados;

import java.util.Scanner;

/**
 *
 * @author
 *
 * Programa que dibuja un cuadrado de asteriscos
 *
 *
 */
public class Bucle_2 {
    public static void main(String[] args) {
        Scanner entrada = new Scanner(System.in);
        int numAsteriscosLado;



        System.out.print("Introduce el número de astericos por lado: ");
        numAsteriscosLado=entrada.nextInt();

        //Dibujamos la parte de arriba del cuadrado
        for(int cont=0;numAsteriscosLado>cont;cont++){
            System.out.print("*");
        }
        System.out.println("");

        //Usamos un bucle anidado para dibujar los asteriscos del medio
        //Calcula las filas intermedias poniendo un * al inicio y final de llas.
        for(int cont=1;(numAsteriscosLado-2)>=cont;cont++){
            System.out.print("*");
            //Este bucle dibuja los espacio entre el primer y ultimo asterisco
            //de cada una de las filas.
            for (int i=0;(numAsteriscosLado-2)>i;i++){
                System.out.print(" ");
            }
            System.out.print("*");
            System.out.println("");
        }

        //Dibujamos la parte de abajo del cuadrado
        for(int cont=0;numAsteriscosLado>cont;cont++){
            System.out.print("*");
        }
        System.out.println("");
    }

}
```

TAREA 05. Hoja de ejercicios 5.

Estructuras de control. Bucles anidados

Realiza la tarea correspondiente que encontrarás en la plataforma educativa.

Realiza la entrega de esta tarea de acuerdo con las instrucciones que se indiquen.

TAREA 06. Hoja de ejercicios 6.

Estructuras de control. Bucles anidados

Realiza la tarea correspondiente que encontrarás en la plataforma educativa.

Realiza la entrega de esta tarea de acuerdo con las instrucciones que se indiquen.

4. Estructuras de salto

En la gran mayoría de libros de programación y publicaciones de Internet, siempre se nos recomienda que prescindamos de sentencias de salto incondicional, es más, se desaconseja su uso por provocar una mala estructuración del código y un incremento en la dificultad para el mantenimiento de los mismos. Pero Java incorpora ciertas sentencias o estructuras de salto que es necesario conocer y que pueden sernos útiles en algunas partes de nuestros programas.

Estas estructuras de salto corresponden a las sentencias break, continue, las etiquetas de salto y la sentencia return. Pasamos ahora a analizar su sintaxis y funcionamiento.

4.1 Sentencia break

Al igual que la sentencia ***continue***, que luego veremos, la sentencia ***break*** permite modificar el comportamiento de otras estructuras o sentencias de control, simplemente por el hecho de estar incluidas en algún punto de su secuencia de instrucciones.

La sentencia ***break*** incidirá **sobre las estructuras de control** switch, while, for y do-while del siguiente modo:

- Si aparece una sentencia **break dentro** de la secuencia de instrucciones de cualquiera de las estructuras mencionadas anteriormente, dicha **estructura terminará inmediatamente**.
- Si aparece una sentencia break **dentro de un bucle anidado sólo finalizará la** sentencia de iteración **más interna**, el resto se ejecuta de forma normal.

Es decir, **break** sirve **para romper el flujo** de control **de un bucle**, **aunque no se haya cumplido la condición del bucle**. Es decir, cuando se alcance el break dentro del código de un bucle, automáticamente **se saldrá del bucle pasando a ejecutarse la siguiente instrucción inmediatamente después de él**.

Sintaxis

```
break;
```

Ejemplo

Uso de la sentencia break dentro de un bucle for.

```
6  public class sentencia_break {
7      public static void main(String[] args) {
8          // Declaración de variables
9          int contador;
10
11
12         //Procesamiento y salida de información
13
14         for (contador=1;contador<=10;contador++)
15         {
16             if (contador==7)
17                 break;
18             System.out.println ("Valor: " + contador);
19         }
20          System.out.println ("Fin del programa");
21         /* El bucle sólo se ejecutará en 6 ocasiones, ya que cuando
22          * la variable contador sea igual a 7 encontraremos un break que
23          * romperá el flujo del bucle, transfiriéndonos a la sentencia que
24          * imprime el mensaje de Fin del programa.
25          */
26     }
```

4.2 Sentencia continue

La sentencia **continue** se utiliza en los bucles for, while y do...while.

La sentencia continue incidirá sobre las sentencias o estructuras de control while, for y do-while del siguiente modo:

- Si aparece una sentencia **continue** dentro de la secuencia de instrucciones de cualquiera de las sentencias anteriormente indicadas, dicha sentencia **dará por terminada la iteración actual y se ejecuta una nueva iteración**, **evaluando de nuevo la** expresión **condicional** del bucle.
- Si aparece **en el interior de un bucle anidado solo afectará a la sentencia de iteración más interna**, el resto se ejecutaría de forma normal.
- **Con for** el **control salta al final del bucle**, con lo que **el contador se incrementa y comienza otra comprobación**. Con **while**, el **control pasa** inmediatamente **a la condición de control**.

Es decir, la sentencia **continue** forzará a que se ejecute la siguiente iteración del bucle, sin tener en cuenta las instrucciones que pudiera haber después del continue, y hasta el final del código del bucle.

Sintaxis
continue;
Ejemplo

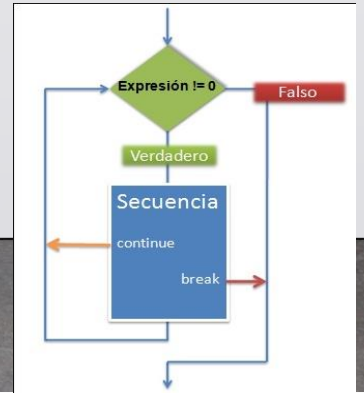
Uso de la sentencia continue dentro de un bucle for.

```
 * Uso de la sentencia continue
 */
public class sentencia_continue {
    public static void main(String[] args) {
        // Declaración de variables
        int contador;

        System.out.println ("Imprimiendo los números pares que hay del 1 al 10... ");
        //Procesamiento y salida de información

        for (contador=1;contador<=10;contador++)
        {
            if (contador % 2 != 0) continue;
            System.out.print(contador + " ");
        }
         System.out.println ("\nFin del programa");
        /* Las iteraciones del bucle que generarán la impresión de cada uno
         * de los números pares, serán aquellas en las que el resultado de
         * calcular el resto de la división entre 2 de cada valor de la variable
         * contador, sea igual a 0.
         */
    }

}
```



4.3 Etiquetas

Los **saltos incondicionales y en especial, saltos a una etiqueta son totalmente desaconsejables**. **No obstante, Java permite asociar etiquetas cuando se va a realizar un salto**. De este modo puede conseguirse algo más de legibilidad en el código.

Las estructuras de salto **break y continue, pueden tener asociadas etiquetas**. Es a lo que se llama un **break etiquetado** o un **continue etiquetado**. Pero **sólo se recomienda su uso cuando se hace necesario salir de bucles anidados hacia diferentes niveles**.

Para **crear un salto a una etiqueta**, crearemos la etiqueta mediante un identificador seguido de dos puntos (:) → nombre_etiqueta: . A continuación, se escriben las sentencias Java asociadas a dicha etiqueta encerradas entre llaves. La creación de una etiqueta **fija un punto de salto en el programa para poder saltar a él desde otro lugar de dicho programa**. Para saltar a este punto, en el lugar donde vayamos a colocar la sentencia break o continue, añadiremos detrás el identificador de la etiqueta.

Sintaxis

nombre_etiqueta: *sentencias*
break/continue <nombre_etiqueta>;

Ejemplo

Uso de la sentencia break dentro de un bucle for.

```
/**
 *
 * Uso de etiquetas en bucle
 */
public class etiquetas {
    public static void main(String[] args) {

        for (int i=1; i<3; i++) //Creamos cabecera del bucle
        {
            bloque_uno: {   //Creamos primera etiqueta
                bloque_dos:{    //Creamos segunda etiqueta
                    System.out.println("Iteración: "+i);
                    if (i==1) break bloque_uno;   //Llevamos a cabo el primer salto
                    if (i==2) break bloque_dos;   //Llevamos a cabo el segundo salto
                }
                System.out.println("después del bloque dos");
            }
            System.out.println("después del bloque uno");
        }
        System.out.println("Fin del bucle");
    }
}
```

https://www.w3schools.com/java/java_break.asp

4.4 Sentencia return

Es posible hacer que los métodos detengan su ejecución antes de que finalice el código asociado a ellos a través de la sentencia **return**.

La sentencia **return** puede utilizarse de dos formas:

- **Para terminar la ejecución del método donde esté escrita**, con lo que transferirá el control al punto desde el que se hizo la llamada al método, continuando el programa por la sentencia inmediatamente posterior.
- **Para devolver o retornar un valor**, siempre que junto a return se incluya una expresión de un tipo determinado. Por tanto, en el lugar donde se invocó al método se obtendrá el valor resultante de la evaluación de la expresión que acompañaba al método.

En general, **una sentencia return suele aparecer al final de un método**, de este modo el método tendrá una entrada y una salida. También **es posible utilizar una sentencia return en cualquier punto de un método**, con lo que **éste finalizará en el lugar** donde se encuentre dicho return. **No es recomendable incluir más de un return en un método** y por regla general, deberá ir al final del método, como hemos comentado.

El valor de retorno es opcional, **si lo hubiera debería de ser del mismo tipo o de un tipo compatible al tipo del valor de retorno definido en la cabecera del método**, pudiendo ser desde un entero a un objeto creado por nosotros. **Si no lo tuviera, el tipo de retorno sería void**, y return **serviría para salir del método** sin necesidad de llegar a ejecutar todas las instrucciones que se encuentran después del return.

Sintaxis

return <valor de retorno>;

Ejemplo

https://www.w3schools.com/java/ref_keyword_return.asp

5. Control de excepciones

Errores: suelen ser los que nos ha indicado el compilador: un punto y coma, un nombre de variable incorrecto,... Cuando lo detectamos, se corrige y obtenemos nuestra clase compilada correctamente.

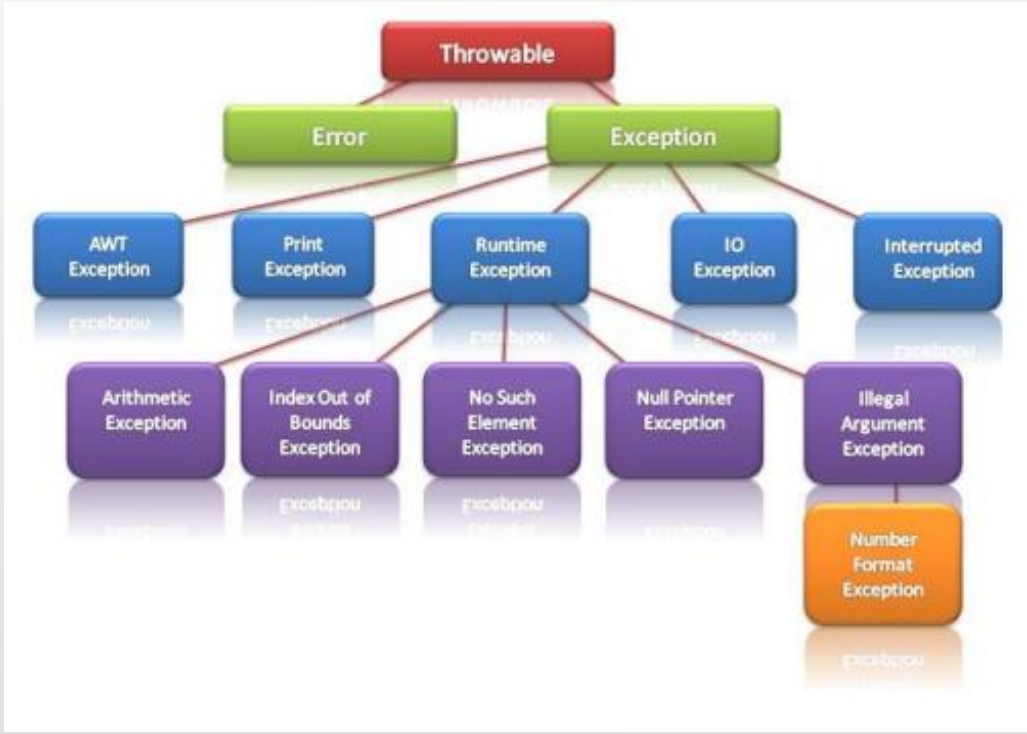
Excepciones: un programa perfectamente compilado en el que **no** existen **errores de sintaxis**, puede generar **otros tipos de errores** en tiempo de ejecución. A estos errores se les conoce como **excepciones**.

Por **ejemplo**:

- El programa pide un número y el usuario inserta letras.
- Se quiere abrir un fichero que no existe.
- Dividir entre cero.
- Hacer uso de un objeto, el cual no apunta a ningún espacio de memoria…

5. Control de excepciones …

En Java se puede preparar los fragmentos de código que pueden provocar errores de ejecución para que si se produce una **excepción**, **el flujo del programa sea lanzado** (**throw**) hacia ciertas zonas o rutinas que han sido creadas previamente por el programador y cuya **finalidad será el tratamiento efectivo de dichas excepciones**.



Si no se captura la excepción, el programa se detendrá con toda probabilidad. En Java, las excepciones están representadas por clases. El paquete **java.lang.Exception** y sus subpaquetes **contienen todos los tipos de excepciones**. **Todas las excepciones derivarán de la clase Throwable**, existiendo clases más específicas. **Por debajo** de la clase Throwable existen **las clases Error y Exception**.

5.1 Capturar una excepción

Para poder capturar excepciones, emplearemos la **estructura de captura de excepciones try-catch-finally**.

Esta estructura **trabaja en bloques**:

- En un bloque **try** (intentar) se introducen las **sentencias a proteger**. Si ocurre una excepción dentro de estos bloques, se lanza una excepción.
- Estas excepciones lanzadas se pueden capturar por medio de bloques **catch**. En un bloque catch se hará el manejo de las excepciones, dependiendo del tipo de excepción se controlará de un modo u otro.

Formato de la estructura:

```
try {
código que puede generar excepciones;
}
catch (Tipo_excepcion_1 objeto_excepcion) {
Manejo de excepción de Tipo_excepcion_1;
}
catch (Tipo_excepcion_2 objeto_excepcion) {
Manejo de excepción de Tipo_excepcion_2;
}
...
finally {
instrucciones que se ejecutan siempre
}
```

Si no hay ningún error se ejecutan las **instrucciones del try** completamente.

Si se produce un error dejan de ejecutarse las instrucciones que hay en el try para **ejecutarse las instrucciones que hay en el catch**.

La parte **catch puede repetirse** tantas veces como excepciones diferentes se desee capturar. En el momento que se produce un error se revisan todos los catch y se ejecuta el bloque de instrucciones del catch que posea el controlador de excepción adecuado. Por tanto, en esta estructura, **cada catch** maneja **un tipo de excepción**. Cuando pase por el primer catch donde el tipo de error coincida ya no pasará por los siguientes. Por ello **el orden de los catch es muy importante**.

La parte **finally** es **opcional** y, si aparece, solo podrá hacerlo **una sola vez**.

5.1 Capturar una excepción …

EJEMPLOcatch1:

```
import java.util.Scanner;
public class ExcepcionTipo1 {
 public static void main(String[] args) {
 Scanner teclado= new Scanner(System.in);
 int numero;
 System.out.println("introduce un numero
entero ");
 numero=teclado.nextInt();
 }
}
```

Si ejecutamos y el usuario introduce un carácter se aborta la ejecución del programa y se lanza un error.

```
run:
introduce un numero a
U

Exception in thread "main" java.util.InputMismatchException
Exception in thread "main" java.util.InputMismatchException
at java.util.Scanner.throwFor(Scanner.java:909)
at java.util.Scanner.next(Scanner.java:1530)
at java.util.Scanner.nextInt(Scanner.java:2160)
at java.util.Scanner.nextInt(Scanner.java:2119)
at ej_catch.catch1.main(catch1.java:11)
Java Result: 1
BUILD SUCCESSFUL (total time: 7 seconds)
```

Capturamos el error **InputMismatchException**

5.1 Capturar una excepción …

EJEMPLOcatch2:

```
import java.util.*;
public class ExcepcionTipo2 {
 public static void main(String[] args) {
 Scanner teclado= new Scanner(System.in);
 int numero;
 try {
 System.out.println("introduce un numero
entero ");
 a=teclado.nextInt();
 }
 catch(InputMismatchException e){
 System.out.println("Error, introduce un numero
entero");
 }
 }
}
```

Como sabemos la clase Exception incluye todos los tipos de errores controlables; así si ponemos:

```
catch (Exception e )
{
}
```

Cuando se produzca un error, pasará por ese catch y ya no pasará por ninguno de los otros catch que haya por debajo

5.1 Capturar una excepción …

EJEMPLOcatch3. Captura excepciones que se producen al introducir letras y al dividir por cero en el siguiente programa:

```
public class ExcepcionDivision {
 public static void main(String[] args) {
 Scanner teclado= new Scanner(System.in);
 int a,b;
 double resultado;

 System.out.println("introduce dos numeros a y b enteros");
 a=teclado.nextInt();
 b=teclado.nextInt();
 resultado=a/b;
 System.out.println("resultado "+resultado);
 }
}
```

```
public class ExcepcionDivision {
 public static void main(String[] args) {
 Scanner teclado= new Scanner(System.in);
 int a,b;
 double resultado;
 try{
 System.out.println("introduce dos numeros a y b enteros");
 a=teclado.nextInt();
 b=teclado.nextInt();
 resultado=a/b;
 System.out.println("resultado "+resultado);
 }
 catch(ArithmeticException e){
 System.out.println("error division por cero");
 }
 catch(InputMismatchException e){
 System.out.println("error introduce dos enteros");
 }
catch (Exception e){
 }
 }
}
```

Si ponemos primero el catch (Exception e) te da un error de codificación en el segundo y tercer catch indicando que ese error ya ha sido recuperado. Prueba a hacerlo.

Por lo tanto: primero se ponen los catch que capturan errores concretos y luego los catch que agrupan errores. Para ello hay que tener muy en cuenta el API de Java (conjunto de bibliotecas que nos proporciona paquetes de clases útiles para nuestros programas) y ver cuál es la herencia que hay entre las clases de excepciones.

Además del try y los catch está el bloque finally, en este bloque están las instrucciones que queremos que se ejecuten siempre, haya o no error

5.1 Capturar una excepción …

EJEMPLOcatch4. Como evitar situaciones de bucle infinito con la petición de valores de la clase Scanner:

Cuando realizamos una petición de valores de la clase Scanner, si el valor introducido por teclado no es correcto y genera una excepción que vamos a gestionar, queda en el buffer del escaner pendiente el salto a la siguiente petición de lectura.

Eso provoca que si estamos en un bucle y capturamos la excepción, no se realiza el salto a la siguiente petición del escaner sino que de nuevo vuelve al catch para gestionar la excepción.

El resultado de esto es un bucle infinito que podemos resolver haciendo una nueva petición al objeto de la clase Scanner dentro del catch; esta nueva petición la logramos hacer con el método next(), con la cual liberamos la memoria del buffer.

En el siguiente ejemplo se puede ver como resolver la situación. Prueba a comentar la línea entrada.next(); para ver cual sería el problema.

```
public static void main(String[] args) {
        Scanner entrada = new Scanner(System.in);
        int numero = 0;
        boolean nopase;
        //Pedir el número hasta que sea válido
do {
                nopase = false;
                System.out.println("Introduce el número: ");
          //hacemos el try hasta que el valor introducido sea correcto
                try {
                    numero = entrada.nextInt();
          /*si el valor introducido por teclado no es un entero salta la excepción
          quedando en el buffer del escaner pendiente el salto a la siguiente petición
de lectura.*/
                } catch (Exception e) {
                    System.out.println("No ha introducido un numero positivo ");
           /*next() Lee una cadena de caracteres de la línea de comandos,
             pero al encontrar un espacio del buffer de lectura termina la cadena.
             Hace que el escaner pase a la siguiente lectura.*/
                    entrada.next();
                    //cambiamos
                    nopase = true;
                } finally {
                    //Este código se ejecuta se produzca o no una excepción
                }
           } while (nopase);

// a partir de aquí continua el programa con el resto del código
}
```

TAREA 07. Hoja de ejercicios 7.

Control de excepciones

Realiza la tarea correspondiente que encontrarás en la plataforma educativa.

Realiza la entrega de esta tarea de acuerdo con las instrucciones que se indiquen.

TAREA 08. Hoja de ejercicios 8.

Control de excepciones

Realiza la tarea correspondiente que encontrarás en la plataforma educativa.

Realiza la entrega de esta tarea de acuerdo con las instrucciones que se indiquen.

TAREA 09. Hoja de ejercicios 9.

Estructuras de control

Realiza la tarea correspondiente que encontrarás en la plataforma educativa.

Realiza la entrega de esta tarea de acuerdo con las instrucciones que se indiquen.

TAREA 10. Hoja de ejercicios 10.

Estructuras de control

Realiza la tarea correspondiente que encontrarás en la plataforma educativa.

Realiza la entrega de esta tarea de acuerdo con las instrucciones que se indiquen.