Ciclo de Desarrollo de Aplicaciones Multiplataforma

Grado Superior

U.T.2.Aplicación de las Estructuras de Control

Módulo de Programación

Departamento de Informática

I.E.S. Monte Naranco

1. API O BIBLIOTECA DE CLASES EN JAVA

Clases y paquetes en java

Una clase en java es la unidad fundamental de programación. Podemos definirla como un fragmento de código que permite crear objetos.

Las clases pueden ser creadas por nosotros mismos o venir predefinidas en el propio lenguaje, para ayudar al programador a realizar múltiples tareas dentro de un programa.

Las clases predefinidas se agrupan en lo que se conoce como Interfaz de Programación de Aplicaciones (API) o Biblioteca de Clases de Java.

La API está organizada en paquetes lógicos ordenados de forma alfabética, donde cada paquete contiene un conjunto de clases relacionadas semánticamente. Por tanto podemos decir, que las clases de java (tanto las predefinidas como las creadas por nosotros mismos) se organizan en paquetes (lo que habitualmente llamamos carpetas).

Las clases se agrupan en paquetes por varias razones:

1. Una cuestión organizativa. Un paquete contiene un conjunto de clases relacionadas entre sí.

2. Para evitar conflicto de nombres. Dos clases de java pueden tener el mismo nombre, siempre que estén en paquetes diferentes, es decir, no puede existir dos clases con el mismo nombre en el mismo paquete.

Es una razón lógica, puesto que si realizamos un proyecto java en el cual participan varios programadores, podemos crear clases propias con el mismo nombre que otro programador, lo cual podría crear un conflicto. Conflicto resuelto si las clases las metemos en diferentes paquetes. Por eso cuando iniciamos un proyecto al crear una clase, eclipse nos pide definir un paquete. Si no introducimos el nombre del paquete, java crea uno por defecto para organizar las clases. El uso de paquetes por defecto está desaconsejado. Los paquetes quedan definidos mediante la palabra package seguida del nombre del paquete.

Ejm:

```
package terminal;
public class Telefono {....}
public class Ordenador {.....}
```

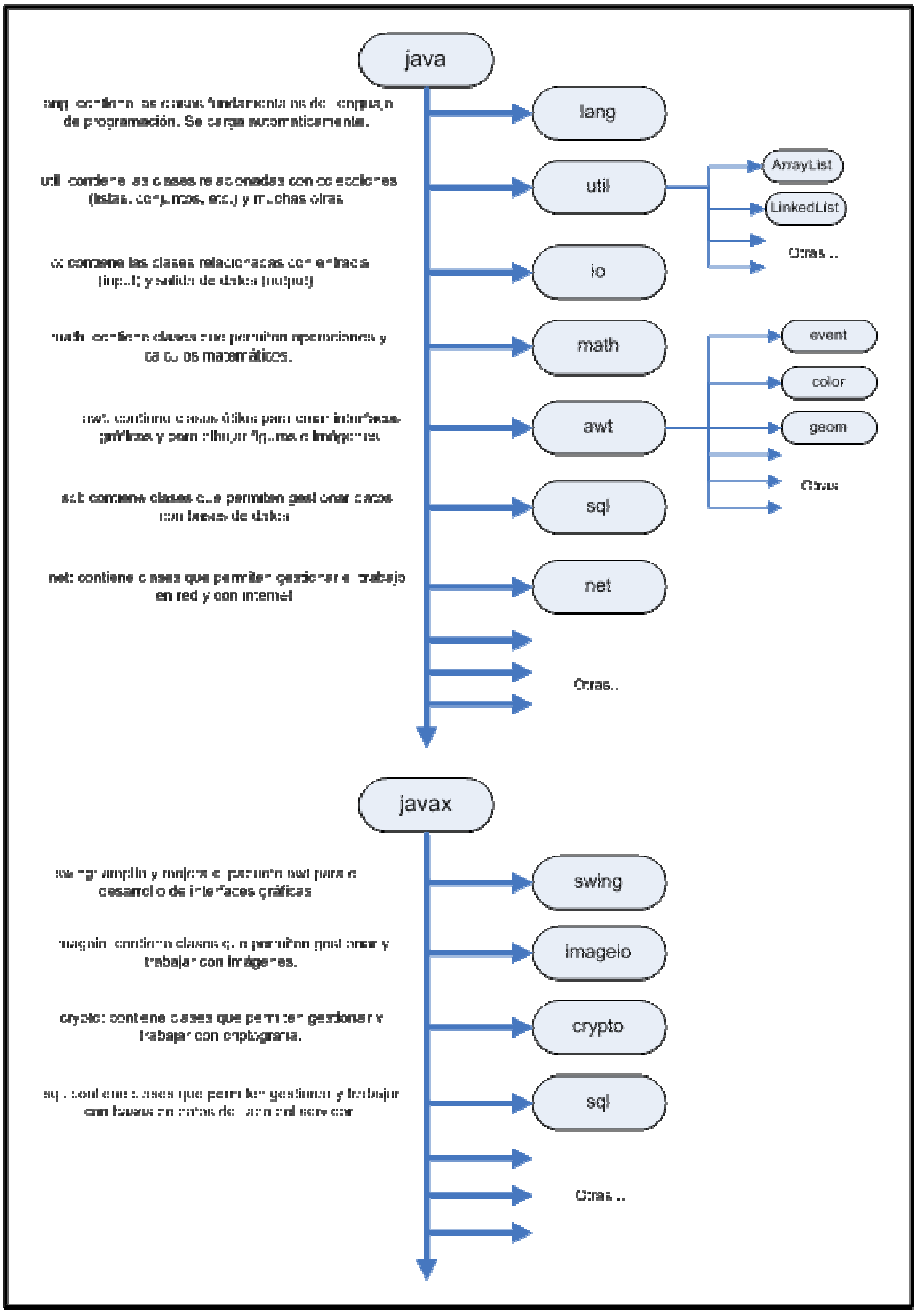
El nombre de los ficheros que contienen las clases van a ser: Telefono.java, y Ordenador.java, que van a estar almacenados en un directorio con el nombre del paquete

3. Para controlar la visibilidad de las clases.

Si echamos un vistazo a la API de java, vemos que consta de varios paneles: panel de paquetes , panel de clases y panel de especificación.

Panel de paquetes

La jerarquía de paquetes se divide en dos ramas: java y javax. Cada rama contiene una serie de paquetes que representan el orden jerárquico de las clases de java, y a su vez, cada paquete contiene una serie de clases predefinidas.



Los paquetes se nombran poniendo la rama de la cual descienden, un punto y el nombre del paquete.

Ejm: java.util (el paquete es util, desciende de la rama java).

Panel de clases

Nos muestra las clases e interfaces que tiene un paquete determinado en orden alfabético. Para ello tenemos que hacer click sobre el paquete cuyas clases queramos visualizar. Así dentro del paquete java.lang, está la clase Math y la clase String, utilizadas frecuentemente.

Panel de especificación

Se nos da información sobre los paquetes, clases, etc.

Importación de paquetes en java

Para poder interactuar entre diferentes paquetes, es decir, para poder utilizar una clase de un paquete en otra clase de otro paquete diferente, necesitamos importarla. Esto no sería necesario si las clases se encuentran en el mismo paquete.

Para importar paquetes debemos hacer uso de la cláusula **import,** la cual va siempre antes de la declaración de clases y después de la cláusula package.

Formas de importar una clase

Hay dos formas de importar una clase siempre que esté declarada como pública:

1. **Importar directamente la clase**

   En la palabra clave import, especificando el nombre del paquete como ruta, y el nombre de la clase.

   Ejm: import java.util.Scanner;

2. **Usar el comodín asterisco (*) para importar todas las clases del paquete**

   Ejm: import java.util.* //Indica que estamos importando a nuestro programa todas las clases del paquete java.util.

Esta opción incrementa el tiempo de compilación, sin embargo no afecta al rendimiento durante la ejecución.

Paquete java.lang

Debido a que dentro de este paquete están gran parte de las clases más utilizadas en java, se importa automáticamente formando parte de todos los programas. Esto quiere decir que no es necesario importarlo.

2. ENTRADA/SALIDA DE DATOS

La salida de datos consiste en sacar o visualizar información desde nuestro programa al exterior. Hasta ahora usamos la consola del sistema a través de la instrucción System.out, junto con el método print. Pero los programas necesitan generalmente que se les introduzca información, lo ideal es construir una IGU (interfaz gráfica de usuario), aunque por ahora nosotros usaremos la clase Scanner, que nos va a permitir introducir información usando la consola del sistema.

Clase Scanner

Si vamos a la API de java y abrimos la clase Scanner, lo primero que observamos es que pertenece al paquete java.util (por tanto hay que importarlo).

Para permitirnos la entrada de datos la clase Scanner utiliza varios métodos: nextByte(), nextShort(),nextInt(),nextLong(), nextFloat(), nextDouble(), nextBoolean(), nextLine(), next(). Todos estos métodos no son estáticos, lo que significa que para poder usarlos necesito crear un objeto de la clase Scanner, con la opción new.

Ejm:

```
Package entrada;
Import java.util.*;
public class Practica{
    public static void main(String[] args){
        Scanner teclado=new Scanner (System.in);
        int edad=0;
        String nombre="";
        System.out.printn ("Meter edad");
        edad=teclado.nextInt();
        System.out.printn ("Meter nombre");
        nombre=teclado.nextLine();
        System.out.println("la edad es "+edad+" y el nombre"+ nombre);
        teclado.close();
        } }
```

Problema que surge en el programa anterior: Al ir a introducir por el teclado el nombre de una persona, el programa no espera a que lo podamos insertar por el teclado. No me deja introducirlo. Esto es debido a que el buffer de entrada no está vacío.

Cómo limpiar el buffer de entrada en java

Cuando en un programa se leen por teclado datos numéricos y datos de tipo carácter o String, debemos tener en cuenta que al introducir los datos y pulsar intro, estamos introduciendo en el buffer de entrada el intro.

Así, cuando encontramos en el programa anterior la instrucción: **edad=teclado.nextInt();** el programa espera a que tecleemos un número. Si meto por el teclado por ejemplo el 5 y confirmamos con intro, en el buffer de entrada se introduce un 5\n. Posteriormente se asigna a la variable edad el valor 5, pero el \n permanece en el buffer.

Si a continuación se pide que se introduzca por teclado una cadena de caracteres: **nombre= teclado.nextLine();** el método nextLine(), extrae del buffer de entrada todos los caracteres hasta llegar a un \n, y elimina el intro del buffer. En este caso asigna una cadena vacía a la variable nombre y limpia el intro. Esto provoca que el programa no funcione correctamente, ya que no se detiene para que se introduzca el nombre.

Solución: se debe limpiar el buffer de entrada si se van a leer datos de tipo carácter, a continuación de la lectura de datos numéricos.

La forma más sencilla de limpiar el buffer de entrada en java, es ejecutar la instrucción: **teclado.nextLine();** después de la lectura del int y antes de leer el String.

Ejm:

```
Package entrada;
Import java.util.*;
public class Practica{
    public static void main(String[] args){
        Scanner teclado=new Scanner (System.in);
        int edad=0;
        String nombre="";
        System.out.printn ("Meter edad");
        edad=teclado.nextInt();
        teclado.nextLine();
        System.out.printn ("Meter nombre");
        nombre=teclado.nextLine();
        System.out.println("la edad es "+edad+" y el nombre"+ nombre);
        teclado.close();
        } }
```

Otra forma de solucionar el problema, sería leer primero el String, y después el int.

3.- ESTRUCTURAS DE CONTROL DE FLUJO

Los tipos de **estructuras** que se emplean **para el control del flujo** de los datos son las siguientes:

- **Secuencial:** compuestas por 1 o N sentencias, que se ejecutan en el orden en que han sido escritas. Es la estructura más sencilla y sobre la que se construirán el resto de estructuras.
- **Alternativa o selección.**
- **Repetitiva o iteración.**

Estructuras alternativas o de selección

Las estructuras alternativas se dividen en:

1. Estructuras alternativas simples o estructura if simple.
2. Estructuras alternativas compuestas o estructura if-else.
3. Estructuras alternativas múltiples o estructura switch.

A continuación detallaremos las características y funcionamiento de cada una de ellas. Es importante que a través de los ejemplos que vamos a ver, puedas determinar en qué circunstancias utilizar cada una de estas estructuras.

Estructura if simple

Sintaxis:

```
if (condición o expresión lógica)
{
  sentencia/s;
}
```

Ejemplo:

```
int valor1=2, valor2=2, valor3=2;

if ((valor1==valor2)&&(valor3<4))
   System.out.println("verdad");
```

Si el cuerpo del if consta de una sola instrucción o sentencia, no es necesario encerrarla entre { }, aunque puede hacerse sin que sea un error.

Si el cuerpo del if consta de más de una instrucción, es obligatorio encerrarlas entre llaves, para señalar el principio y final del bloque de instrucciones.

Funcionamiento:

Debemos evaluar la condición o expresión lógica. Si el resultado es verdadero, se ejecutan las instruccione/s escritas debajo del if, y a continuación se continúa ejecutando el programa normalmente. En caso contrario (resultado falso), no se ejecutan las instruccione/s correspondientes al if, y se continúa ejecutando el programa.

Estructura if - else

Es una ampliación de la estructura if simple. Un if puede no tener un else, pero no puede existir un else, sin su if correspondiente.

Sintaxis:

```
if (condición o expresión lógica )
{
  sentencia/s;
}
else
{
   sentencia/s;
}
```

Ejemplo:

```
int num = 6;
if (num < 6)
{
  System.out.println ("menor");
  num = 8;
}
else
  System.out.println ("mayor");
System.out.println ("Fin");
```

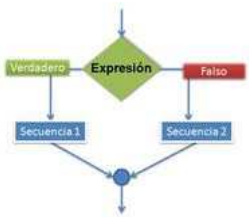
Si el cuerpo del if o del else constan de una sola instrucción o sentencia, no es necesario encerrarla entre { }, aunque puede hacerse sin que sea un error.

Si el cuerpo del if o del else constan de más de una instrucción, es obligatorio encerrarlas entre llaves, para señalar el principio y final del bloque de instrucciones.

Funcionamiento:

Debemos evaluar la condición o expresión lógica. Si el resultado es verdadero, se ejecutan las instrucciones escritas debajo del if, se saltan las correspondientes al else, y se continúa ejecutando el programa normalmente. En caso contrario (resultado falso), se ejecutan las instrucciones correspondientes al else (saltándose las correspondientes al if), y se continúa ejecutando el programa.

Representación mediante diagrama de flujo:



Estructura if anidada

Las estructuras if e if-else, pueden anidarse, de tal forma que dentro de un bloque de sentencias puede incluirse otro if o if-else. El nivel de anidamiento queda a criterio del programador, pero si éste es demasiado profundo, podría provocar problemas de eficiencia y legibilidad en el código. En otras ocasiones, un nivel de anidamiento excesivo puede denotar la necesidad de utilización de otras estructuras de selección más adecuadas.

Sintaxis:

```
if (condición o expresión lógica )
{
  sentencia/s;
}
else
   if (condición o expresión lógica )
  {
    sentencia/s;
  }
 else
 {
   sentencia/s;
 }
```

Ejemplo:

```
int sueldo =900;
if (sueldo > 1500)
  System.out.println (sueldo);
else
  if (sueldo < 1000)
  {
     System.out.println ("El sueldo es :");
     System.out.println (sueldo+" poco");
  }
 else
         System.out.println ("mayor ");

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

Cuando se utiliza anidamiento de este tipo de estructuras, es necesario poner especial atención en saber a qué if está asociada una cláusula else.

Cada else se corresponde con el if inmediatamente superior o más próximo a él que esté sin asignar todavía, empezando por el primer else escrito.

Estructura switch.

Es la alternativa a la estructura if anidada, cuando la expresión puede tomar distintos valores. Nos permite por tanto, realizar una selección múltiple.

Sintaxis:

```
switch (expresión)
{
  case valor1:
       sentencia/s;
       break;
   case valor2:
        sentencia/s;
        break;
.....
  [default]:
   sentencia/s;
 }
```

Ejemplo:

```
int valor =2;
switch (valor)
{
   case 1:
      System.out.println ( "uno");
       break;
    case 2:
       System.out.println ( "dos");
       break;
 }
 System.out.println ("Fin");
```

Condiciones:

- Expresión debe ser de tipo byte, short, int o char.
- Hay que asociar una cláusula case, a cada uno de los valores a tener en cuenta.
- La cláusula case puede llevar asociadas una o varias sentencias, sin necesidad de delimitar dichos bloques por medio de llaves.
- Si se omite un break en alguno de los case puestos, se ejecutarán el resto de los case, hasta encontrar un break.

Ejm:

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

int estado =sc.nextInt();

switch (estado){

case 1:/*Tanto si estado vale 1 como si vale 2 se visualiza por pantalla:
        se omitió break*/

case 2:

  System.out.println("se omitió break");

  break;

case 3:

  System.out.print("El valor es 3 ");

  /*Si estado vale 3, se imprime: El valor es 3 El valor es 4. Hasta que
  encuentra un break*/

case 4:

  System.out.println("El valor es 4");

  break;

default:

  System.out.println("No es ningún valor");

}
```

- En un case no podemos definir variables locales, pero en una estructura switch sí.

  Ejm:

```
Scanner sc=new Scanner (System.in);
String cadena=sc.next();//No admite espacios en blanco
char caracter=cadena.charAt(0);
switch (caracter){
int x=2; //Declaración correcta
case 'a':
  int y=9; //Declaración no permitida
  System.out.println("El carácter era a");
  break;
case 'b':
  System.out.print("El carácter es b ");
default:
  System.out.println("No es ningún valor");
}
```

- La cláusula default es opcional. No es necesario que contenga la sentencia break.

Funcionamiento:

Se evalúa la expresión. Si el resultado coincide con alguno de los valores asociados a las cláusulas case, se ejecutarán todas las instrucciones relativas a ese valor hasta encontrar la sentencia break, que hace que el programa salte a la sentencia siguiente al switch. Si ningún valor de la expresión se corresponde con ninguna opción, se ejecutarían las sentencias del bloque default si existe, y se continúa ejecutando el programa normalmente.

Una aplicación típica de la estructura switch, es el uso de menús.

Estructura switch anidada

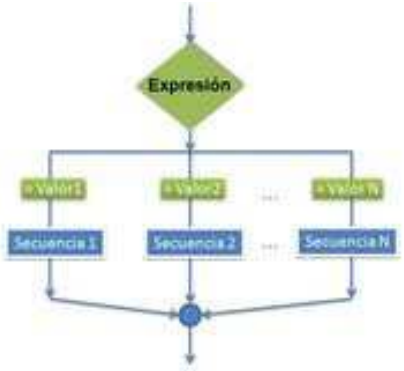
Al igual que el if, la estructura switch admite anidamiento.

Ejm:

```
Scanner sc=new Scanner (System.in);
int estado =sc.nextInt();
String cadena=sc.next();
char caracter=cadena.charAt(0);
int dato= sc.nextInt();
switch (caracter)
{
    case 'a':
        switch (dato)
            {
                case 1:
                    System.out.println ("El valor de dato es 1");
                    break;
                case 2:
                    System.out.println ("El valor de dato es 1");
                    break;
                default:
                    System.out.println ("No es ningún valor");

            }
        System.out.println ("Salimos del anidamiento");
        break;
    case 'b':
        System.out.println("El valor de estado es dos");
        break;
}
```

Representación mediante diagrama de flujo:



Estructuras repetitivas o de iteracción

Nos permiten repetir una secuencia de instrucciones un número determinado de veces, dependiendo del resultado de una condición.

Las instrucciones que se repiten un número determinado de veces, constituyen lo que denominamos bucle.

Existen cuatro tipos de estructuras repetitivas:

- Bucle while (repite mientras)
- Bucle do while (repite hasta)
- Bucle for (repite para)
- Bucle for/in (repite para cada)

Estructura while.

Se usa cuando tenemos que repetir una o más instrucciones un número determinado de veces, pero no conocemos cuántas.

Sintaxis:

```
while (condición)
{
  sentencia/s;
}
```

Si el cuerpo del while consta de una sola instrucción o sentencia, no es necesario encerrarla entre { }, aunque puede hacerse sin que sea un error.

Si el cuerpo del while consta de más de una instrucción, es obligatorio encerrarlas entre llaves para señalar el principio y final del bloque de instrucciones.

Funcionamiento:

La condición se evaluará siempre al principio, pudiendo darse el caso de que las instrucciones contenidas en él no lleguen a ejecutarse nunca, si no se satisface la condición de partida.

Mientras la condición sea cierta, las instrucciones pertenecientes al while se ejecutarán. En el momento en el que la condición no se cumpla, el control del flujo del programa pasará a la siguiente instrucción que exista justo detrás del bucle while.

Es imprescindible que en el interior del bucle while, se realice alguna acción que modifique la condición que controla la ejecución del mismo, en caso contrario estaríamos ante un bucle infinito.

Ejemplo:

```
int num=0;
while (num <3)
{
    suma=suma+num; //Acumulador
    num=num+1; //Contador
}
```

En este ejemplo aparece dos figuras que usaremos con bastante asiduidad:

Contadores

Un contador es una variable cuyo valor se incrementa o decrementa en una cantidad constante (1,2,3...), cada vez que se produce una acción. Debe inicializarse, es decir, asignarle un valor inicial.

Ejm:

```
int contador=0;
contador=contador+1;
contador++;
contador--;
contador=contador-1;
```

Acumuladores

Un acumulador es una variable que incrementa o decrementa su valor, en cantidades variables.

Los acumuladores deben inicializarse. Cuando se trata de realizar sumas sucesivas, deben inicializarse a 0. En el caso de productos, deben inicializarse a 1.

Ejm:

```
AC = AC + VAR;
AC = AC - VAR;
AC = AC * VAR;
AC = AC / VAR;
```

Siendo VAR una cantidad variable.

Diagrama de flujo que representa el funcionamiento de este tipo de estructura repetitiva

Estructura do-while

La característica fundamental de este tipo de estructura repetitiva, es que las instrucciones que forman el bucle se ejecutan al menos una vez.

Sintaxis:

```
do
{
   sentencia/s
} while (condicion);
```

Tanto si el cuerpo del bucle consta de una instrucción como de varias, son imprescindibles las llaves.

Funcionamiento

El cuerpo del bucle se ejecuta la primera vez, a continuación se evaluará la condición, y si ésta es verdadera, el bucle volverá a repetirse, finalizando cuando la condición sea falsa. En ese momento el control del flujo del programa, pasará a la siguiente instrucción que exista justo detrás, del do-while.

Es imprescindible que en el interior del bucle se realice alguna acción que modifique la condición que controla la ejecución del mismo, en caso contrario estaríamos ante un bucle infinito.

Ejemplo:

```
package estructura_repetitiva;


public class Prueba {


	public static void main(String[] args) {
		int contador=0;
		do
		{
			System.out.println(contador);
			contador++;
		}
		while (contador<6);
	}
}
```

Diagrama de flujo que representa el funcionamiento de este tipo de estructura repetitiva.

Estructura repetitiva for

Se utiliza cuando se quiere ejecutar un bloque de instrucciones, un número determinado de veces. Utiliza una variable contador que controla las iteraciones (vueltas) del bucle.

Sintaxis:

```
 for (inicialización del contador; condición; variación del contador)
{
        sentencia/s
}
```

Si el cuerpo del for consta de una sola instrucción o sentencia, no es necesario encerrarla entre { }, aunque puede hacerse sin que sea un error.

Si el cuerpo del for consta de más de una instrucción, es obligatorio encerrarlas entre llaves, para señalar el principio y final del bloque de instrucciones.

Su funcionamiento consta de tres operaciones

1. Se inicializa el contador, que será el encargado de controlar el número de iteraciones.

2. Se evaluará la condición. El bucle for se ejecutará hasta que la condición sea falsa. Dicho de otra forma, el bucle for se ejecutará mientras la condición sea verdadera.

3. Se modifica o actualiza el valor del contador a través de incrementos o decrementos de éste en cada iteración. El contador no tiene porqué variar solo de uno en uno.

Condiciones

- La inicialización del contador solo se efectúa una vez al principio.
- Las variables definidas en la sentencia de inicialización son locales al bucle, dejando de existir una vez haya finalizado éste.
- El contador puede ser modificado tanto por la propia cabecera del for, como por instrucciones dentro del bucle.
- Es imprescindible que se realice alguna acción que modifique la condición, en caso contrario estaríamos ante un bucle infinito.

Ejemplo:

```
int suma = 0;
for (int contador = 0; contador < 3; contador++)
        suma = suma + contador;
```

La variable contador solo existe dentro del bloque. Una vez fuera del bloque, contador se destruye liberando memoria.

Variaciones del for

a) Bucles for con varias variables de control

El operador coma, permite encadenar operaciones.

Ejm:

```
for (sum1 = 0, sum2 = 0; sum1 + sum2 < 100; sum1++, sum2 += 2)
        System.out.println(sum1+sum2);
```

b) Bucles sin especificar las variables de control en la cabecera

Ejm:

```
int contador = 10;
for ( ;  contador < 12  ; )
{
        System.out.println (contador);
        contador++;
}
```

c) Bucles for sin cuerpo

Nos permiten generar retardos.

Ejm:

```
final int TOPE = 10000;
for (contador = 1; contador < TOPE ; contador ++); /* Incrementa el
valor  de contador hasta TOPE, y no hace nada más*/
```

Diagrama de flujo que representa el funcionamiento de este tipo de estructura repetitiva.

Estructura repetitiva for/in

Se le denomina también for mejorado, o bucle foreach.

Al igual que la estructura for, for/in también está controlada por un contador. Esta estructura es una mejora incorporada en la versión 5.0. de Java.

Permite recorrer arrays y colecciones, que veremos más adelante.

4.- SENTENCIAS DE RUPTURA

Java incorpora ciertas sentencias o estructuras de salto, que es necesario conocer y que pueden sernos útiles en algunas partes de nuestros programas, sin embargo no conviene abusar de ellas, ya que provocan una mala estructuración del código, y un incremento en la dificultad para el mantenimiento de los mismos.

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 el bucle más interno, el resto se ejecutará de forma normal.

Es decir, si colocamos un break dentro del código de un bucle, cuando se alcance, automáticamente se saldrá del bucle pasando a ejecutarse la siguiente instrucción inmediatamente después de él.

Ejm:

Uso de la sentencia break dentro de un bucle for

```
public class sentencia_break {
     public static void main(String[] args) {
        // Declaración de variables
        int contador;


        //Procesamiento y salida de información

        for (contador=1;contador<=10;contador++)
        {
            if (contador==7)
                break;
            System.out.println ("Valor: " + contador);
        }
         System.out.println ("Fin del programa");
        /* El bucle sólo se ejecutará en 6 ocasiones, ya que cuando
         * la variable contador sea igual a 7 encontraremos un break que
         * romperá el flujo del bucle, transfiriéndonos a la sentencia que
         * imprime el mensaje de Fin del programa.
         */
    }
```

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 detiene la ejecución del bucle, y se vuelve a evaluar la condición de éste.
- Si aparece en el interior de un bucle anidado, solo afectará al bucle más interno, el resto se ejecutará de forma normal.

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.

Ejm:

Uso de la sentencia continue en un bucle for, para imprimir por pantalla sólo los números pares.

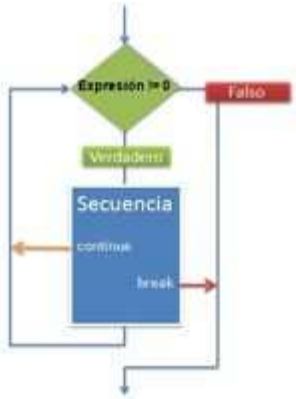
```
 * Uso de la sentencia continue
 */
public class sentencia_continue {
    public static void main(String[] args) {
        // Declaración de variables
        int contador;

        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.
         */
    }

}
```

Diagrama de flujo de las sentencias break y continue



Sentencia Return

Se utiliza para finalizar la ejecución de un método.

5.- INTRODUCCIÓN A LOS MÉTODOS

CONCEPTO

Los programas informáticos suelen ser muy complejos, como para abordarlos en su conjunto. Por ello podemos dividir el problema en partes más simples e independientes, que pueden ser construidas y probadas por separado (modularización).

Podemos definir un método, como un conjunto de instrucciones que se ejecuta para llevar a cabo una tarea concreta y bien definida. Con los métodos se evita reescribir código cada vez que esa tarea se quiera utilizar en un programa.

Los métodos pueden ser definidos por nosotros, o pueden ser proporcionados por el propio java.

Conviene poner antes de la definición de los métodos un comentario de lo que hace.

TIPOS DE MÉTODOS

1. Los que nos devuelven un solo valor

Cabecera o signatura del método: realmente la signatura de un método es su interfaz, porque informa de lo que hace, pero no de cómo lo hace.

Modo acceso otros modificadores tipo retorno nombre método ([tipo parámetro, tipo parámeto…])

```
{
        Variables locales
        Cuerpo del método
        Return resultado devuelto
  }
```

Donde:

- **Modo acceso**

a) Público: se puede invocar al método desde la propia clase donde esté, o desde cualquier otra clase, sea o no del mismo paquete.

b) Privado: solo la propia clase donde se encuentra el método, tiene acceso al mismo.

c) Protected o no escribir modo de acceso: se puede invocar al método desde cualquier clase de su mismo paquete, pero no desde otro paquete diferente.

Los métodos por lo general son públicos. Pero también puede existir la necesidad a veces, de disponer de métodos privados para

realizar operaciones intermedias o auxiliares (métodos de comprobación, de adaptación de formatos, de cálculos intermedios). Estos métodos, no son de interés (no es apropiado que sean visibles fuera del contexto del interior del objeto). Por **ejemplo**, un objeto que contenga un DNI y que es necesario calcular la letra del un DNI, o mirar si una combinación de número+ letra=DNI válido. Este tipo de cálculo pueden ser implementados como privados.

- **Otros modificadores**

Static: se les denomina métodos de clase, lo que significa, que no necesitan de la existencia de un objeto de una clase para poder ejecutarse, se pueden ejecutar directamente utilizando el nombre de la clase., o solo con el nombre, si estamos en la misma clase. **Por ejemplo**, la clase Math, tiene métodos estáticos como pow, sqrt, abs…. El acceso a los métodos se hace usando el nombre de la propia clase (Math.sqrt (valor)). Desde un método estático, solo puedo invocar a métodos estáticos.

- **Tipo retorno**: es el tipo de datos que devuelve la sentencia return. Pueden ser tipos primitivos de datos o de referencia.

- **Nombre del método**: es el nombre con el que se va a invocar o llamar al método. Puede usarse cualquier nombre, ahora bien, se ha establecido el siguiente convenio: Utilizar un verbo o una frase que indique acción en minúsculas , o bien un nombre formado por varias palabras que comience por un verbo en minúsculas, seguido por adjetivos, nombre, etc, que sí aparecerá la primera letra en mayúsculas (calcularSuperficie).

- **Parámetros:** se denominan también parámetros formales. Son variables locales, que usa el método para sus cálculos. Su ámbito de existencia es el propio método, cuando éste acaba de ejecutarse se destruyen y se libera memoria. Se puede incluir cualquier cantidad de parámetros, es una decisión del programador, pudiendo no incluir ninguno, y pueden ser de cualquier tipo (primitivos, referencias, objetos, etc). No puede haber 2 parámetros con el mismo nombre. Los métodos con parámetros, son métodos que nos piden algo (datos u objetos), y los que no tienen parámetros, no piden ningún dato u objeto para ejecutarse.

- **Variables locales**: Se declaran dentro del método, y existen solo mientras se ejecuta el mismo (solo son visibles para ese método), es decir, tienen carácter temporal, de ahí, que no tengan modo de acceso (public, prívate,

protected). Una variable local del método, no puede coincidir con el nombre de un parámetro, sin embargo, podemos usar el mismo nombre de una variable local en diferentes métodos, puesto que no interfieren entre ellos.

- **Cuerpo del método:** contiene las sentencias o sentencias de control, que implementan el método.

- **Return:** Debería ser la última línea del método, ya que si hay una línea por detrás, no llega a ejecutarse nunca.

El return, puede devolver un resultado (variable, constante o expresión algebraica), o no devolver nada. En cualquiera de los dos casos, cuando se encuentra esta instrucción, se devuelve el control al punto donde se hizo la llamada al método, y el programa continúan en la sentencia inmediatamente posterior.

Si el tipo de datos del resultado devuelto por la sentencia return, no coincide con el tipo de datos que debe devolver el método, se intentará automáticamente convertir el tipo de datos de la expresión, para que haya coincidencia. El resultado puede ser un error de compilación o pérdida de información, por tanto se debe evitar estas inconsistencias.

En un método pueden existir varias instrucciones return, cuando incluye varias bifurcaciones (uso de condicionales).

2. Los que no devuelven ningún valor:

Cabecera o signatura del método

Modo acceso otros modificadores void nombre método ([tipo parámetro, tipo parámeto…])

```
{
        Variables locales
        Cuerpo del método
  }
```

Estos métodos, no devuelven ningún valor, pero pueden presentar datos en pantalla, escribir o leer en otras estructuras de datos, etc.

Como vemos el tipo de retorno es void, y no llevan return, o se usa sin un valor asociado (return;), aunque no tiene sentido.

LLAMADA O INVOCACIÓN A LOS MÉTODOS

Se efectúa en el método main, o desde cualquier otro método, y se hace por nombre.

Sintaxis: Nombre del método (argumento, argumento…)

En los métodos que nos devuelven un valor, la llamada no puede aparecer sola, debe aparecer: a la derecha de una sentencia de asignación, formando parte de una expresión o en una salida por pantalla. Si la llamada aparece sola, su valor se pierde.

En los métodos que no devuelven ningún valor, la llamada aparece sola.

Los argumentos se llaman también parámetros actuales o reales, no llevan tipo y están separados por comas. Son valores que se establecen en el programa llamante, y que se traspasan (por valor), al método que llaman.

Los argumentos se corresponden con los parámetros por posición y tipo. Si en el paso argumento-parámetro, el tipo recibido no coincide con el esperado, hay un error. Los argumentos pueden llamarse igual que los parámetros, aunque son variables diferentes.

Como funciona

Cuando llamamos al método, se ejecutan sus instrucciones, y cuando se termina de ejecutar, se devuelve el flujo de control al programa que llama al método.

PASO DE ARGUMENTOS POR VALOR

En java los argumentos siempre se pasan por valor.

Cuando los argumentos son de tipos básicos o primitivos (byte, short, int, long, float, double, char, boolean), no se traspasan los propios datos, sino una copia de ellos, por tanto aunque los parámetros varíen su valor, los argumentos no lo varían.

Cuando los argumentos son objetos o instancias de clase, se copia la dirección de memoria del argumento en el parámetro (el parámetro pasa a apuntar a la estructura de datos donde lo está haciendo el argumento), por tanto si se modifica la estructura de datos, esa modificación se mantiene.

Cuando el método termina, se borra la copia de la referencia que se pasó como parámetro, pero la referencia original sigue apuntando al objeto sobre el que ha trabajado el método.

MÉTODO MAIN

Es un método especial que da lugar al inicio del programa. Su cometido por tanto es iniciar el programa. Es al que llamará el sistema operativo cuando pidamos ejecutar el programa.

No debe ser muy extenso en cuanto a líneas del código. Debe permanecer en un segundo plano, mientras los objetos interactúan entre sí. Si esto no ocurre, es indicador de que tiene más protagonismo del que debe. En esencia debe limitarse a crear objetos e invocar sus métodos.

```
public class test {
        public static void main (---)
        {
        }
}
```

El main es estático, ya que cuando se invoca no existen objetos creados por que la ejecución del programa no ha comenzado. Es void porque su función es arrancar el programa, no devuelve su valor. Como parámetros tiene el String[] arg que es una tabla de cadenas de caracteres (se le suele llamar args, aunque no es obligatorio), donde el sistema operativo cargará los valores que debe enviar a nuestro programa.