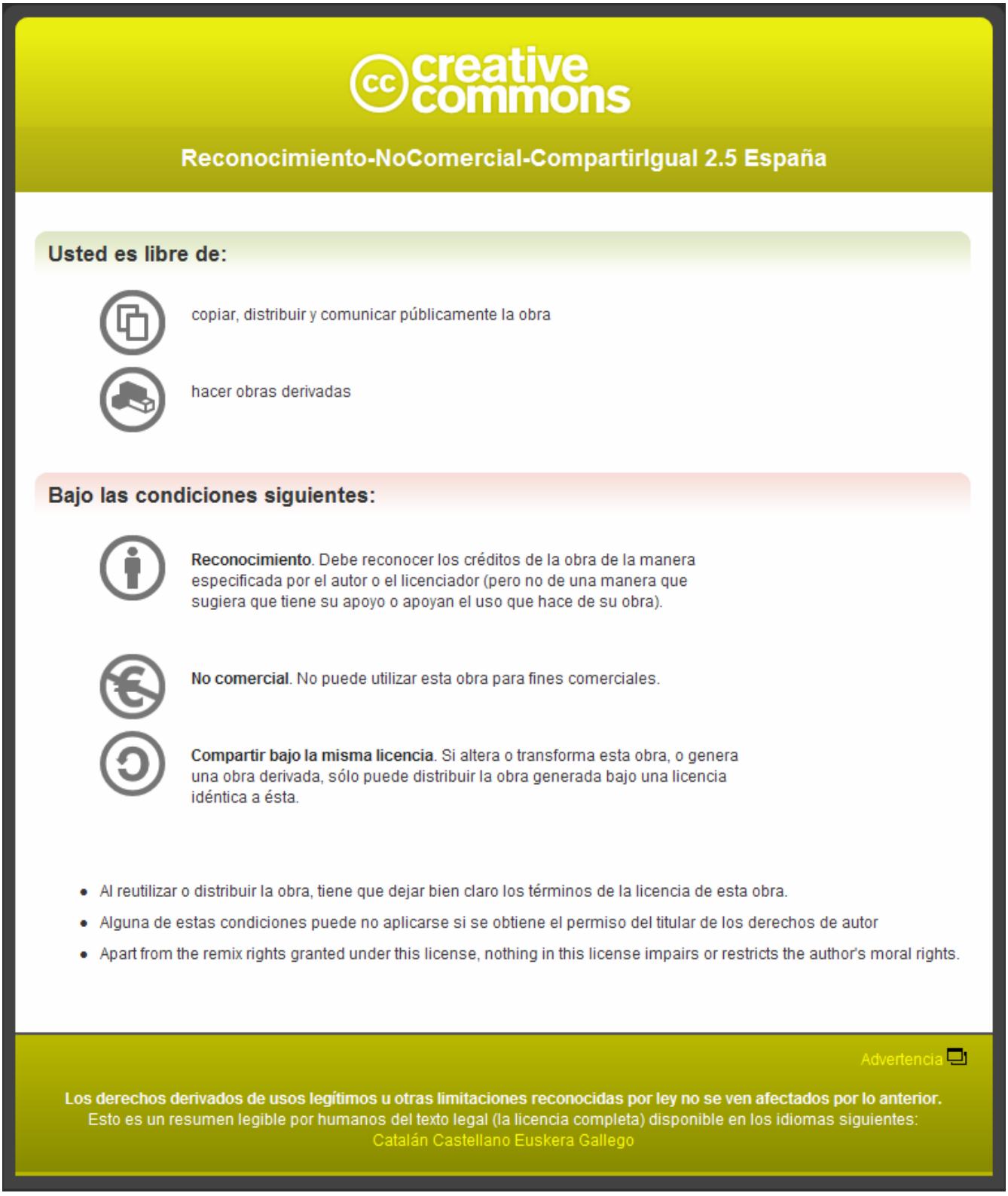
Unidad 4: Programación estructurada en lenguaje Java

Fundamentos de Programación. 1º de ASI



Para ver una copia completa de la licencia, acudir a la dirección http://creativecommons.org/licenses/by-nc-sa/2.5/es/legalcode.es

(4) programación estructurada en Java

esquema de la unidad

(4.1) introducción. expresiones lógicas

Hasta ahora las instrucciones que hemos visto, son instrucciones que se ejecutan secuencialmente; es decir, podemos saber lo que hace el programa leyendo las líneas de izquierda a derecha y de arriba abajo.

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 **(&&, | |**, **!**).

(4.2) if

(4.2.1) 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++;
}
```

La idea gráfica del funcionamiento del código anterior sería:

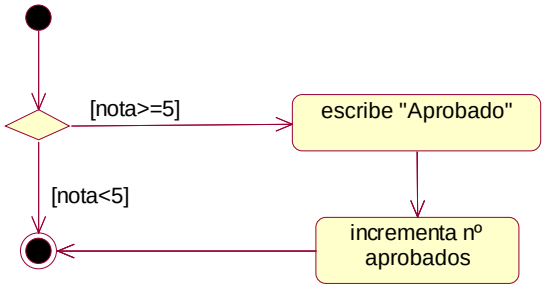


Ilustración 3-1, Diagrama de actividad de la instrucción *if* simple

(4.2.2) 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
   ....
}
else {
   instrucciones
   ...
}
```

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++;
}
```

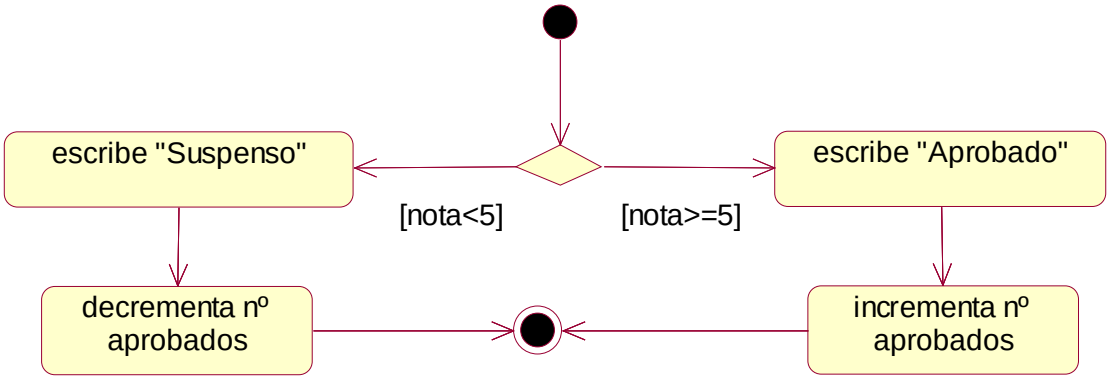


Ilustración 3-2, **Diagrama de actividad de una instrucción *if* doble**

(4.2.3) anidación

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:

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

Una forma más legible de escribir ese mismo código sería:

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

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

(4.3) 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 1
         [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.

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.**

Ejemplo 1:

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

Ejemplo 2:

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

(4.4) 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++;
}
```

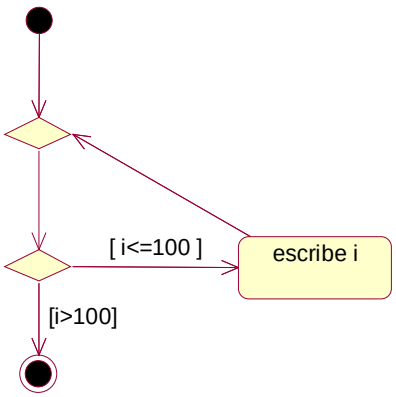


Ilustración 3-3, Diagrama UML de actividad del bucle while

bucles con contador

Se llaman así a los bucles que se repiten una serie determinada de veces. Dichos bucles 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,... o como queramos) 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 del bucle.

(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.

Ejemplo:

```
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 */

   printf("%d\n",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*/
}
/* Al final el bucle escribe:
10
20
30
...
y así hasta 200
*/
```

Bucles de 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.floor(Math.random()*500+1; // Lo que se repite en el
   System.out.println(i);  // bucle: calcular un número
                           aletorio de 1 a 500 y escribirlo */
   salir=(i%7==0); /* El centinela vale verdadero si el número es
                      múltiplo de 7
                      */
}
```

Comparando los bucles de centinela con los de contador, podemos señalar estos puntos:

- Los bucles de contador se repiten un número concreto de veces, los bucles de centinela no
- Un bucle de contador podemos considerar que es seguro que finalice, el de centinela puede no finalizar si el centinela jamás varía su valor (aunque, si está bien programado, alguna vez lo alcanzará)
- Un bucle de contador está relacionado con la programación de algoritmos basados en series.

Un bucle podría ser incluso mixto: de centinela y de contador. Por ejemplo imaginar 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.

Sería:

```
boolean salir = false; //centinela
int n;
int i=1; //contador
while (salir == false && i<=5) {
   n = (int) Math.floor(Math.random() * 500 + 1);
   System.out.println(n);
   i++;
   salir = (i % 7 == 0);
}
```

(4.5) 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)
```

Ejemplo (contar de uno a 1000):

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

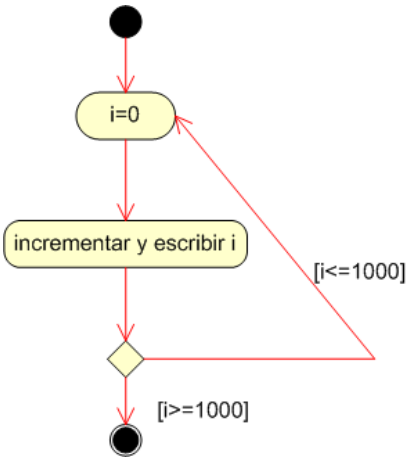


Ilustración 3-4, Diagrama de actividad del bucle do..while

Se utiliza cuando sabemos al menos que 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. El ejemplo anterior se puede escribir usando la instrucción while, así:

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

(4.6) 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

(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

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.

(4.7) sentencias de ruptura de flujo

No es aconsejable su uso ya que son instrucciones que rompen el paradigma de la programación estructurada. Cualquier programa que las use ya no es estructurado. Se comentan aquí porque puede ser útil conocerlas, especialmente para interpretar código de terceros.

(4.7.1) sentencia break

Se trata de una sentencia que hace que el flujo del programa abandone el bloque en el que se encuentra.

```
for(int i=1;i<=1000;i++){
   System.out.println(i);
   if(i==300) break;
}
```

En el listado anterior el contador no llega a 1000, en cuanto llega a 300 sale del ***for***

(4.7.2) sentencia continue

Es parecida a la anterior, sólo que en este caso en lugar de abandonar el bucle, lo que ocurre es que no se ejecutan el resto de sentencias del bucle y se vuelve a la condición del mismo:

```
for(int i=1;i<=1000;i++){
   if(i%3==0) continue;
   System.out.println(i);
}
```

En ese listado aparecen los números del 1 al 1000, menos los múltiplos de 3 (en los múltiplos de 3 se ejecuta la instrucción ***continue*** que salta el resto de instrucciones del bucle y vuelve a la siguiente iteración.

El uso de esta sentencia genera malos hábitos, siempre es mejor resolver los problemas sin usar ***continue.*** El ejemplo anterior sin usar esta instrucción quedaría:

```
for(int i=1;i<=1000;i++){
   if(i%3!=0) System.out.println(i);
}
```

La programación estructurada prohíbe utilizar las sentencias **break** y **continue**