Grado en Ingeniería Informática
Grado en Matemáticas e Informática

Asignatura: PROGRAMACIÓN II

Elementos básicos de programación imperativa en Java

Manuel Collado

DLSIIS - E.T.S. de Ingenieros Informáticos
Universidad Politécnica de Madrid

Febrero 2014

En este documento se introducen algunos elementos básicos del lenguaje Java. En concreto los que permiten desarrollar programas siguiendo el modelo de programación imperativa, pero sin llegar a utilizar de manera explícita el mecanismo de clases y objetos.

Por lo tanto lo que se aborda aquí es una base preliminar para introducir posteriormente la programación orientada a objetos en Java.

Elementos de programación imperativa

Concepto de programación imperativa

El término ***programación imperativa*** corresponde a un modelo de programación en que los programas se expresan como colecciones de órdenes que se van obedeciendo a medida que se ejecuta el programa.

Ejemplo cotidiano: Receta de cocina.

La programación imperativa se caracteriza también por el uso de ***variables*** para almacenar datos. Se puede modificar el contenido de una variable tantas veces como se desee durante la ejecución del programa.

Programación orientada a objetos

La programación orientada a objetos es un modelo particular de programación imperativa que introduce además el concepto de ***clase*** como agrupación de un tipo de datos con las operaciones que los manejan. Los datos se plantean como ***objetos*** de la clase correspondiente.

Tipos de datos básicos

Tipos numéricos

Java maneja valores numéricos con o sin parte fraccionaria. En un programa se pueden expresar como literales numéricos en notación decimal simple o científica (mantisa + exponente de potencia de 10).

- Tipo `int`: valores enteros (`7, 29, 0, -34, 568,` ...)
- Tipo `double`: valores en coma flotante (`27.3, 0.0, -12.5, 123E4,` ...)

Observaciones

Los valores de tipo `int` se representan con 32 bits. Su rango va de -2147483648 a 2147483647.

Los valores de tipo **`double`** se representan con 64 bits. Tienen unos 16 dígitos de precisión, y su rango es aproximadamente $\pm 10^{308}$.

Existe también otro tipo entero **`long`**, que se representa con 64 bits. Admite valores hasta 9223372036854775808. Los literales numéricos deben escribirse con el sufijo L (`37L, -1L,` ...).

Existe también otro tipo **`float`** para valores en coma flotante con 32 bits. Tiene unos 7 dígitos de precisión, en el rango de $\pm 10^{38}$. Los literales numéricos deben llevar el sufijo F (`-4.5F, 3.45E3F,` ...)

Operadores numéricos

Java admite, entre otros, los siguiente ***operadores numéricos***:

- Operadores aritméticos: suma, resta, producto, cociente, resto (`+ - * / %`). Notación infija.
- Operadores de incremento y decremento (`++ --`). Prefijo o sufijo.
- Operadores de cambio de signo (`+ -`). Prefijo.

Observaciones

La división entre valores de tipo entero produce el cociente entero por defecto. Ejemplos: `3/4` → `0`, `8/3` → `2`.

Se pueden combinar operaciones con valores numéricos de los distintos tipos. El operando del tipo de menor rango se convierte al tipo de mayor rango, y el resultado se obtiene en el tipo de mayor rango. El orden de menor a mayor rango es **`int`**, **`long`**, **`float`**, **`double`**. Se puede perder algo de precisión en la conversión a tipos de coma flotante.

Los operadores de incremento y decremento deben aplicarse a variables (`a++, --b,` ...). La variable queda incrementada o decrementada. Si el operador es sufijo, el resultado es el valor anterior de la variable (se incrementa *a posteriori*). Si el operador es prefijo el resultado es el valor final de la variable (se incrementa *a priori*).

Tipo booleano

Corresponde a valores lógicos o de verdad. El nombre del tipo es **`boolean`**. Los valores posibles se expresan con los literales booleanos **`true`** y **`false`**.

Operadores booleanos

Se pueden obtener valores lógicos mediante los siguientes ***operadores booleanos***:

- Operadores de comparación: igual, distinto, mayor, menor, etc. (`== != > >= < <=`).
- Operadores lógicos: unión, intersección y negación (`&& || !`).

Observaciones

Los operadores `&&` y `||` se evalúan "*en cortocircuito*". Es decir, se evalúa primero el primer operando, y si con ello ya se puede deducir el resultado el segundo operando no se evalúa. Ejemplos: para evaluar `(false && x)` o `(true || x)` no hace falta conocer el valor de `x`.

Tipo carácter

En Java existe el tipo **`char`** cuyos valores son los posibles caracteres individuales de un texto. Estos valores se representan literalmente escribiendo el carácter entre apóstrofos (`'a', 'A', '3', '=', '@',` ...).

Algunos caracteres no imprimibles se pueden representar como una secuencia de escape, formada por la barra inclinada hacia atrás seguida de una letra de código. También deben representarse como secuencia de escape la barra inclinada hacia atrás y el apóstrofo. Ejemplos:

- Carácter de tabulación `'\t'`.
- Carácter de fin de línea `'\n'`.

- Barra inclinada hacia atrás `'\\'`.
- Apóstrofo `'\''`.

Observaciones

Los valores de tipo **`char`** se representan internamente como valores numéricos en 16 bits. El rango es de 0 a 65535. El valor numérico de un carácter es el código Unicode que le corresponde. La codificación Unicode permite representar caracteres de todos los alfabetos, símbolos matemáticos y técnicos, etc.

Operadores con caracteres

Se pueden aplicar **operadores de comparación** a valores de caracteres. El resultado corresponde a la comparación de sus códigos numéricos. Esto no siempre corresponde al orden alfabético de un idioma determinado.

Se pueden aplicar **operadores numéricos** a los valores de tipo **`char`**. Si es necesario se aplica la regla de conversión entre valores numéricos de distinto tipo. El rango del tipo **`char`** es menor que el de **`int`**. Ejemplos:

- `'7'-'0'` da como resultado el valor numérico 7.
- `'A'+1` da como resultado el código del carácter `'B'`.

Estas operaciones sólo tienen sentido en algunos casos. En particular interesa saber que las letras mayúsculas tienen códigos consecutivos, y también las letras minúsculas y los dígitos decimales.

Clase String

Además de los caracteres individuales, en Java se pueden manejar valores de texto que contengan cadenas de caracteres. No corresponden a un tipo de datos básico, sino a la clase predefinida **`String`**. Un *string* puede tener cero, uno o más caracteres.

Los valores de *strings* se pueden introducir como literales de texto entre comillas dobles (`"ejemplo"`, `"Felipe"`, `"hoy es lunes"`, `"123"`, `""`). Dentro del literal se pueden usar las secuencias de escape correspondientes a caracteres especiales, y por supuesto al carácter de comilla doble (`"Linea 1\nLinea 2"`, `"La \"x\" es una letra"`).

Operadores con *strings*

El operador + aplicado a *strings* tiene el significado de ***operador de concatenación***.

Ejemplo: `"uno"+"dos"` → `"unodos"`.

El operador de concatenación permite combinar *strings* con valores de otros tipos, que son convertidos automáticamente a su representación como texto.

Ejemplos: `"3"+5` → `"35"`. `"3>5="+(3>5)` → `"3>5=false"`.

Observaciones

Los *strings* no pueden compararse con los operadores de comparación (==, >, ...). En lugar de eso hay que utilizar métodos definidos en la clase `String` (`equals`, `compareTo`, ...).

No se puede modificar parte de un *string* sobre él mismo. Para cambiar parte de un *string* hay que fabricar otro *string* nuevo combinado la parte que no cambia con el valor nuevo de la parte que cambia.

Conversión de tipos

Se puede producir automáticamente una conversión de tipo de valores numéricos cuando:

- Se asigna valor a una variable.
- Se combinan valores con un operador.
- Se pasa un valor como argumento a un subprograma.

La conversión automática sólo se realiza si el tipo al que se convierte es de mayor rango que el tipo original (*widening*). Si se quiere convertir un valor de un tipo a otro de menor rango (*narrowing*) hay que utilizar un operador de conversión explícito (***casting***). Esta operación se escribe poniendo el nombre del tipo de destino entre paréntesis delante del valor a convertir.

Ejemplos: `(int) 3.0` → `3`. `(char) 97` → `'a'`.

Observaciones

La conversión a un tipo entero no redondea el valor, sino que lo trunca.

```
(int) 3.3 → 3
(int) 3.7 → 3
```

El operador de *casting* tiene mayor prioridad que los de multiplicación o división, y menor que los de incremento o decremento.

```
(char) 'a' + 1 → 98
(char) ('a' + 1) → 'b'
```

Sólo se puede aplicar el operador de *casting* entre tipos compatibles.

```
(boolean) 0 → ERROR
(char) "a" → ERROR
```

Aunque los tipos sean compatibles, la operación de *casting* puede producir error si el valor a convertir está fuera del rango del tipo de destino.

```
(char) 123456789 → VALOR ERRÓNEO
```

Comentarios

Comentarios de línea y de bloque

Los ***comentarios de línea*** se introducen con el símbolo `//` y se extienden hasta el final de la línea de código fuente. Ejemplo:

```
int DIAS_SEMANA = 7; // Número de días de una semana
```

Los ***comentarios de bloque*** se inician con `/*` y terminan con `*/`. Se pueden extender a lo largo de varias líneas, y también ocupar sólo parte de una línea. Ejemplo:

```
/*
 * Calcular el área del rectángulo
 */
area = b /*base*/ * a /*altura*/ ;
```

Comentarios Javadoc

Los ***comentarios Javadoc*** son una forma especial de comentarios de bloque, que empiezan con `/**`. Pueden contener texto con marcas especiales. Se procesan con la herramienta Javadoc, que extrae esos comentarios y genera un documento que describe los elementos contenidos en el programa. Ejemplo:

```
/**
 * Calcular el área de un rectángulo
 *
 * @param b base del rectángulo
 * @param a altura del rectángulo
```

```
 * @return el área del rectángulo
 */
static double areaRectangulo( double b, double a ) {
    return b * a;
}
```

Constantes, variables y expresiones

Identificadores

Los ***identificadores*** permiten dar nombre a distintos elementos del programa. Deben empezar por una letra y pueden contener letras y dígitos decimales (`Texto`, `valor`, `x`, `f16`, `valorMaximo`, ...). También pueden contener el guión bajo (`HAL_9000`).

Observaciones

Es habitual usar identificadores que sean nombres compuestos, iniciando cada parte intermedia con mayúscula (`nombreDeVariable`, `NombreDeClase`).

Java acepta como letras las que se designan como tales en la codificación Unicode. Esto incluye letras con acentos, la eñe, y en general letras en cualquier alfabeto (`Año`, `cálculo`, `cigüeña`, `Μετά`, ...).

Esta posibilidad se usa poco en el campo profesional por posibles problemas de compatibilidad entre sistemas con distinta codificación de ficheros de texto. E incluso si se garantiza la compatibilidad de codificación de texto, no es aconsejable utilizar identificadores que sólo se distingan por la presencia o ausencia de tildes en determinadas letras (`maximo` vs. `máximo`).

Palabras reservadas

Hay ***palabras reservadas*** que siguen las reglas de construcción de los identificadores, pero cuyo uso está reservado como elementos especiales del lenguaje Java, y no pueden usarse para otra cosa.

- Palabras clave: `static`, `for`, `return`, ...
- Nombres de los tipos básicos: `int`, `double`, `boolean`, ...
- Constantes de los tipos predefinidos: `true`, `false`, ...

Observaciones

La lista completa de palabras reservadas en Java es la siguiente:

```
abstract assert boolean break byte case catch char class const continue default double do else enum extends
false final finally float for goto if implements import instanceof int interface long native new null package
private protected public return short static strictfp super switch synchronized this throw throws transient
true try void volatile while
```

Las palabras `const` y `goto` no se usan para nada en Java. Se han reservado por razones históricas.

Variables

Una ***variable*** se declara indicando su tipo, su nombre y, opcionalmente, su valor inicial. Se pueden declarar varias variables del mismo tipo en la misma declaración. Ejemplos:

```
int ancho;
double suma = 0.0;
boolean esPrimo = true, hecho = false;
```

Observaciones

El tipo puede ser un tipo básico o un nombre de clase. El valor inicial se da como una expresión.

El nombre de la variable es directamente visible desde el punto de su declaración hasta el final del bloque de código en que se declara.

Java dispone de mecanismos para poder nombrar una variable desde fuera del ámbito de su declaración (modificadores de visibilidad, nombres cualificados, etc.).

Constantes con nombre

Las ***constantes con nombre*** se declaran como las variables, pero anteponiendo el calificador `final` a la declaración. Ejemplo:

```
final double PI = 3.1415926535897932;
```

El calificador `final` indica que una vez que se asigna un valor inicial, ya no podrá ser modificado.

Observaciones

Es frecuente escribir los nombres de constantes todo en letras mayúsculas. Si es un nombre compuesto se usa el guión bajo para separar cada parte del nombre (`VALOR_MAXIMO`).

Expresiones

Las ***expresiones*** representan un cálculo a realizar combinando operadores y operandos. Los operandos pueden ser literales, constantes con nombre, variables o invocación de funciones. Se pueden usar paréntesis para indicar el orden de evaluación. Ejemplos:

```
3.5 + a * b - 7
2 * (alto + ancho)
letra != 'x'
final >= inicial && final <= limite
```

En ausencia de paréntesis los operadores se evalúan según un orden de prioridad preestablecido. Las operaciones consecutivas de la misma prioridad se evalúan de izquierda a derecha (si son prefijos se evalúan de derecha a izquierda). Algunos grupos de prioridad, de mayor a menor, son:

```
! - (operadores prefijo)
* / %
+ -
< <= > >=
== !=
&&
||
```

Observaciones

Es aconsejable que la evaluación de una expresión no modifique las variables que intervienen en ella (eso se denomina efectos laterales). Por lo tanto no es aconsejable usar los operadores de incremento o decremento como parte de una expresión con más términos (`++x-32`). La excepción son expresiones simples de incremento o decremento de un contador.

Tipos de datos compuestos

Tipos "array"

Un ***array*** (o vector) es un dato compuesto por una colección de elementos todos del mismo tipo. Una variable *array* se declara normalmente como:

```
tipoElemento[] variable = valorInicial;
```

Donde
- *tipoElemento* es el tipo de cada uno de los elementos
- *variable* es el nombre del *array*
- *valorInicial* es un constructor que fabrica el *array* con sus elementos inicializados

Ejemplos:

```
double[] punto = new double[2];
char[] letras = {'a', 'b', 'c', 'd', 'e'};
int[] valores = new int[] {1, 2, 3};
```

En el primer ejemplo se declara el número de elementos pero no sus valores. Por defecto toman valor cero. En el segundo ejemplo se indican los valores de los elementos, y no hace falta decir explícitamente cuántos son. El tercer ejemplo combina de alguna manera la sintaxis de los dos anteriores.

Observaciones

Los corchetes que designan un tipo *array* se pueden poner también detrás del nombre de la variable en lugar de detrás del tipo (`double punto[]` en lugar de `double[] punto`).

El valor inicial se puede omitir. En este caso el *array* realmente no existe hasta que se construya más adelante.

Operaciones con "arrays"

En general, para operar con un *array* hay que operar con sus elementos individuales. Los elementos se numeran correlativamente empezando por el índice 0 hasta N-1 (si hay N elementos). La manera de hacer referencia a un elemento en concreto es *variable*`[`*índice*`]`. El índice puede ser una expresión. Ejemplos:

```
punto[0] = 3.5;
punto[1] = 6.1;
int x = 3;
letras[x] = letras[x+1];
```

El tamaño (número de elementos) de un *array* es un atributo del *array* que se designa como *variable*`.length`.

Observaciones

Si se intenta acceder a un elemento que no existe (el índice está fuera del rango de 0 a N-1) se produce un error, y el programa termina en ese momento.

Una vez construido un *array* no se puede cambiar su tamaño. Por ejemplo, si se necesita añadir elementos habrá que construir un nuevo *array* de mayor tamaño. Eso sí, una vez construido ese nuevo *array* se podrá almacenar en la misma variable *array* de antes, sustituyendo al anterior que se había quedado pequeño.

Sentencias simples

Sentencia expresión

Una ***sentencia simple*** se plantea como una única orden. Las sentencias simples en Java terminan con punto y coma. Una sentencia simple puede ser simplemente una expresión a evaluar (***sentencia-expresión***).

```
expresión;
```

La expresión se evalúa y, dependiendo de cuál sea, produce un efecto u otro. A continuación se describen algunos casos frecuentes, cada uno con un sentido particular.

Sentencia de asignación

La ***asignación*** de valor a una variable se consigue escribiendo:

```
variable = expresión;
```

Si lo que se quiere es asignar a una variable el valor siguiente o anterior al que tiene en ese momento, basta escribir:

```
variable++;
variable--;
```

en lugar de

```
variable = variable + 1;
variable = variable - 1;
```

Observaciones

Esta sentencia es un caso particular de sentencia-expresión. En Java no existe realmente una **sentencia** de asignación, sino un **operador** de asignación. Una expresión puede contener asignaciones en medio de ella. El resultado de la asignación es el valor asignado. Ejemplo:

`a = b = 3;` equivale a `b = 3; a = 3;`
`a = (b = 3 + 2) + 1;` equivale a `b = 3 + 2; a = 5 + 1;`.

Existen operadores que combinan el realizar una operación con una variable y asignar el resultado a la misma variable. Ejemplos:

`a += 3;` equivale a `a = a + 3;`
`a *= 7;` equivale a `a = a * 7;`.

Sentencia de llamada a subprograma

Para ***invocar un subprograma*** se escribe su nombre seguido de los valores de los argumentos entre paréntesis.

```
nombre ( expresión, expresión, ... );
```

Observaciones

Esta sentencia es un caso particular de sentencia-expresión.

Se debe evitar que las expresiones de los argumentos tengan efectos laterales, es decir, provoquen cambios en el estado del programa en medio de la invocación del subprograma. Si realmente interesa evaluar argumentos mediante expresiones con efectos laterales es preferible usar variables auxiliares para separar el cálculo del valor del argumento respecto a la invocación del subprograma. Ejemplo:

`operacion( x++ );` se puede reescribir como `aux = x; x++; operacion( aux );`.

Sentencias de escritura

En Java no hay ***sentencias de escritura*** específicas. La escritura de resultados se consigue invocando los subprogramas adecuados disponibles en la librería estándar de Java. Por ejemplo, para escribir resultados por la salida principal se pueden usar los subprogramas `print()` y `println()`. Ejemplo:

```
System.out.print( 3 );
System.out.println( " en la primera línea" );
System.out.print( "y " + 3*2 + " en la segunda línea" );
System.out.println();
// El resultado es:
// 3 en la primera línea
// y 6 en la segunda línea
```

Observaciones

El argumento de `print()` o `println()` puede ser un valor cualquiera, que se convierte a representación como texto antes de enviarlo a la salida. `println()` genera un fin de línea después de escribir el argumento. El argumento de `println()` se puede omitir, y en ese caso sólo se genera un salto de línea.

Además de generar saltos de línea con `println()`, también se pueden intercalar caracteres de salto de línea (`\n`) como parte del valor a escribir (`System.out.print("Línea 1\nLínea 2");`).

Hay dos canales de salida predefinidos, que se designan como `System.out` y `System.err`. El primero es la salida principal y el segundo una salida auxiliar para mensajes de incidencias. Por defecto ambas salidas se mezclan y se dirigen a la pantalla.

Sentencias de lectura

En Java no hay ***sentencias de lectura*** específicas. La lectura de datos se consigue invocando los subprogramas adecuados disponibles en la librería estándar de Java. La lectura de datos suele ser complicada porque exige analizar el texto de entrada y convertir cada fragmento al tipo de dato Java adecuado.

El siguiente ejemplo utiliza un *scanner* (analizador) asociado a la entrada principal del programa.

```
import java.util.Scanner;
. . . .
Scanner entrada = new Scanner(System.in);
. . . .
// Datos:
// 17 enero 2014 Moreno Blanco, Albino
int dia = entrada.nextInt();  // --> 17
String mes = entrada.next();  // --> "enero"
int año = entrada.nextInt();  // --> 2014
String cliente = entrada.nextLine(); // --> " Moreno Blanco, Albino"
```

Observaciones

Hay un canal principal de entrada predefinido, que se designa como `System.in`. Por defecto está asociado a la entrada por teclado.

El `Scanner` de Java descompone la entrada en fragmentos de texto separados por espacio en blanco o saltos de línea. Cada fragmento se puede leer por separado y convertirlo a un valor del tipo deseado.

La orden `nextLine()` lee el resto de la línea (o una línea siguiente), incluyendo los espacios en blanco que contenga.

La lectura de valores numéricos con parte fraccionaria requiere ciertas precauciones. La interpretación del texto de entrada sigue por defecto las convenciones culturales de cada país. En España habría que usar la coma como separador de la parte fraccionaria en los datos de entrada, a diferencia de la conversión para escritura, en que se usa por defecto el punto decimal.

Sentencias compuestas

Sentencia bloque

Un ***sentencia compuesta*** o ***sentencia de control*** es una orden para ejecutar de manera controlada otras sentencias. La forma más sencilla de sentencia compuesta es la ***sentencia bloque***, consistente en plantear como una acción global la ejecución una tras otra de varias sentencias. La notación Java consiste en escribir las sentencias componentes en su orden, encerrando el conjunto entre llaves `{}`.

```
{
    sentencia
    sentencia
    . . . .
}
```

Observaciones

Dentro de un bloque se pueden declarar variables u otros elementos. Los elementos así declarados sólo existen dentro del bloque, y no pueden ser usados desde fuera.

```
int a, b
. . . .
// intercambiar a y b
{
    int aux = a;
    a = b;
    b = a;
}
// aux ya no es visible en este punto
```

Sentencia if

La sintaxis de la ***sentencia if*** es:

```
if (condición) {
    acción
}
```

La *condición* es una expresión booleana. La *acción* es una secuencia de sentencias. El efecto es que si se cumple la condición se ejecuta la acción, y si la condición no se cumple entonces no se hace nada.

Observaciones

La acción con sus llaves es una sentencia bloque. Si la acción es una única sentencia, entonces se pueden omitir las llaves, pero es aconsejable usarlas en todos los casos para hacer más clara la estructura del programa.

Sentencia if-else

La ***sentencia if-else*** ordena ejecutar una u otra de dos acciones posibles dependiendo de una condición. Se escribe:

```
if (condición) {
    acción_1
} else {
    acción_2
}
```

Si se cumple la condición se ejecuta la *acción_1*, y si no se cumple entonces se ejecuta la *acción_2*.

Se pueden encadenar varias sentencias if-else para ordenar la ejecución de una acción entre varias posibles dependiendo de ciertas condiciones. Una forma recomendable de escribirlo es:

```
if (condición_1) {
    acción_1
} else if (condición_2) {
    acción_2
} . . . .
    . . . .
} else {
    acción_por_defecto
}
```

Cada acción se ejecuta si se cumple su condición pero no las anteriores. La acción por defecto se ejecuta cuando no se cumple ninguna de las condiciones.

Observaciones

Como ya se ha indicado, Java permite omitir las llaves correspondientes a acciones que sean una sola sentencia. Pero en este caso es aún más recomendable no hacerlo, porque resultaría mucho más confusa la estructura del código.

Sentencia switch

Otra manera de seleccionar una acción entre varias posibles es utilizando un ***discriminante***. La ***sentencia switch*** permite usar como discriminante un valor entero, booleano, carácter o *string*. La notación Java es:

```
switch (expresión) {
case a:
    acción_a
    break;
case b:
    acción_b
    break;
default:
    acción_por_defecto
}
```

El valor de la *expresión* sirve como discriminante. Cada caso (*a, b* ...) representa una acción a ejecutar si el valor del discriminante en exactamente el que se indica. La acción por defecto se ejecuta si el valor del discriminante no es ninguno de los que se mencionan.

Observaciones

Las acciones pueden ser secuencias de sentencias, y no es necesario encerrarlas entre llaves.

Si el valor del discriminante no es ninguno de los mencionados y no existe alternativa **`default`**, entonces se produce un error y el programa termina en ese momento.

Si se omite alguna sentencia **`break`** el efecto es que tras realizar la acción seleccionada se continúa ejecutando también la acción siguiente. Esto debe ser evitado, ya que entonces el significado del código resulta bastante confuso.

Se pueden asociar varios valores distintos del discriminante a una misma acción. Para ello se escriben varias cláusulas **`case`** seguidas, sin acción ni **`break`**, delante de la acción asociada:

```
. . . .
case a_1:
case a_2:
case a_3:
    acción_a
    break;
. . . .
```

Bucle while

La ***sentencia while*** ordena repetir una acción mientras se cumpla cierta condición.

```
while (condición) {
    acción
}
```

La *condición* es una expresión booleana. La *acción* es una secuencia de sentencias.

Si la condición no se cumple al comienzo, la acción no se ejecuta nunca. Si la condición se cumple al comienzo y la acción no modifica en algún momento los términos de la condición para que deje de cumplirse, el bucle se repite indefinidamente y el programa nunca termina.

Observaciones

La acción con sus llaves es una sentencia bloque. Si la acción es una única sentencia, entonces se pueden omitir las llaves, pero es aconsejable usarlas en todos los casos para hacer más clara la estructura del programa.

Bucle for

La ***sentencia for*** equivale a un bucle *while* con sus elementos reorganizados de cierta manera:

```
for (inicio; condición; siguiente) {
    acción
}
```

- La *condición* y la *acción* son idénticas a las del bucle *while*.
- Como *inicio* se puede usar una declaración de variable, o bien sentencias simples separadas por coma. Se ejecutan antes de la primera repetición del bucle.
- El elemento *siguiente* son sentencias simples separadas por coma. Se ejecutan tras cada repetición del bucle, para preparar la repetición siguiente, si la hubiera.

El uso más frecuente de la sentencia *for* es para controlar las repeticiones del bucle mediante un contador. Ejemplos:

```
for (int k=1; k<=limite; k++) {
    // k varía de 1 a limite
    acción_que_usa_k
}
for (int k=0; k<limite; k++) {
    // k varía de 0 a limite-1
    acción_que_usa_k
}
```

Observaciones

La acción con sus llaves es una sentencia bloque. Si la acción es una única sentencia, entonces se pueden omitir las llaves, pero es aconsejable usarlas en todos los casos para hacer más clara la estructura del programa.

Tanto *inicio* como *siguiente* se pueden omitir. En el caso de que se omitieran ambas el bucle *for* se reduciría estrictamente a un bucle *while*.

Si *inicio* es una declaración de variable, dicha variable sólo existe durante la ejecución del bucle:

```
for (int k=0; condición; siguiente) {
    acción
}
// k ya no es visible aquí
```

Bucle for iterador

Existe otra variante de la sentencia **for**, la ***sentencia for iterador***, que permite recorrer automáticamente los elementos de un *array* sin tener que usar explícitamente sus índices.

```
for (Tipo variable : array) { acción }
```

Tipo variable es una declaración de variable. Se asume que los elementos del *array* son también de ese tipo. La variable va tomando el valor de los elementos del *array*, uno tras otro, y con cada uno se ejecuta la *acción*. Ejemplo de código para sumar los valores de un array de números:

```
double[] lista = ...;
double suma = 0.0;
for (double x : lista) {
    suma = suma + x;
}
```

Observaciones

La variable de control sólo existe durante la ejecución del bucle:

```
for (double x : array) {
    acción
}
// x ya no es visible aquí
```

Subprogramas

Funciones

Una ***función*** representa el cálculo de un cierto valor a partir de ciertos datos de partida denominados ***argumentos***. Cada función tiene un nombre que permite hacer referencia a ella. El código para realizar el cálculo se escribe de manera independiente respecto al código que aprovecha esos cálculos. En Java se puede definir una función de la siguiente manera:

```
static Tipo nombre( Tipo_1 argumento_1, Tipo_2 argumento_2, ... ) {
    cálculo_del_resultado
    return resultado;
}
```

- *Tipo* es el tipo del resultado de la función.
- *nombre* es el nombre de la función
- Cada argumento se designa mediante un nombre, precedido del tipo de su valor.
- El cálculo del resultado puede incluir declaraciones de variables y sentencias de cualquier clase.
- La sentencia **return** termina el cálculo de la función y devuelve el *resultado*, que puede darse como una expresión.

Ejemplo:

```
// Cálculo del factorial de un número
static int factorial( int n ) {
    int valor = 1;
    for (int k=2; k<=n; k++) {
        valor = valor * k;
    }
    return valor;
}
```

Una función se usa como un término más en una expresión aritmética, invocando la función por su nombre e indicando los valores concretos de los argumentos, en forma de expresiones.

```
int numElementos, permutaciones;
. . . .
permutaciones = factorial( numElementos );
```

Observaciones

Java es un lenguaje orientado a objetos. El modificador **static** se ha usado aquí para indicar que la función puede ser utilizada en sí misma, sin necesidad de que exista algún objeto sobre el que operar.

En algunos casos el valor del resultado está disponible ya en algún punto intermedio del cálculo de función, y no necesariamente al final. En ese caso conviene resistir la tentación de usar un **return** para terminar la función anticipadamente, ya que el código resulta más propenso a errores si posteriormente hay que modificarlo. Siempre se puede almacenar el resultado anticipado en una variable auxiliar, y esperar al final para devolver ese valor.

En particular es especialmente conveniente no interrumpir un bucle con un **return** en medio de la acción que se repite.

Una excepción a lo anterior es cuando al final de la función hay una elección entre varias alternativas distintas para evaluar el resultado. En ese caso es aceptable poner un **`return`** diferente al final de cada alternativa:

```
static int mayor( int a, int b ) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}
```

Procedimientos

Un ***procedimiento*** es como una función pero que no calcula ningún valor como resultado, sino que simplemente realiza una acción. Se declara igual que una función pero usando **`void`** como tipo del resultado.

```
static void nombre( Tipo_1 argumento_1, Tipo_2 argumento_2, ... ) {
    acción
}
```

- El nombre y los argumentos se indican como en las funciones.
- El código de la *acción* puede incluir declaraciones de variables y sentencias de cualquier clase.
- No es necesaria una sentencia **`return`**, ya que no hay que devolver ningún valor.

Ejemplo:

```
// Imprimir una fecha
static void imprimirFecha( int dia, int mes, int año ) {
    System.out.print( dia + "/" + mes + "/" + año );
}
```

Un procedimiento se invoca con una sentencia de llamada a subprograma.

```
imprimirFecha( 17, 1, 2014 );
```

Observaciones

No se necesita usar **`return`** para indicar el final de la acción del procedimiento, pero puede usarse si se considera conveniente. La sentencia *return* en un procedimiento no va acompañada de ningún valor a devolver (**`return;`**).

El uso de *return* para terminar la acción de manera anticipada debe hacerse con cuidado. En general sólo resulta aceptable cuando la acción se realiza eligiendo una entre varias alternativas, usando sentencias *if* consecutivas en lugar de *if-else* anidadas:

```
static void nombre( ... ) {
    if (condición_1) {
        acción_1
        return;
    }
    if (condición_2) {
        acción_2
        return;
    }
    acción_por_defecto
}
```

Subprogramas recursivos

Un subprograma puede invocarse a sí mismo, si se considera conveniente. Ejemplo:

```
// Cálculo recursivo del factorial de un número
static int factorial( int n ) {
    if (n <= 1) {
```

```
        return 1;
    } else {
        return n * factorial( n-1 );
    }
}
```

Observaciones

La invocación recursiva debe ser una operación condicional. Si un subprograma se llama a sí mismo incondicionalmente, entonces se acumularían las llamadas indefinidamente y el programa Java terminaría con indicación de memoria agotada (cada llamada a subprograma requiere un cierto espacio en memoria mientras se está ejecutando).

Programas

Programa principal

Por defecto, la ejecución de un ***programa principal*** comienza por un subprograma que tenga el nombre **`main`** y como argumento un *array* de *strings*. Este subprograma debe estar dentro de una clase cuyo nombre se usa como nombre del programa.

```
public class Ejemplo {

    public static void main (String[] args) {
        System.out.println( "Soy un programa Java" );
    }

}
```

El programa se invocaría con la orden:

```
java Ejemplo
```

Índice alfabético