Sumario

Control del Flujo y Arrays

Sentencias if – else

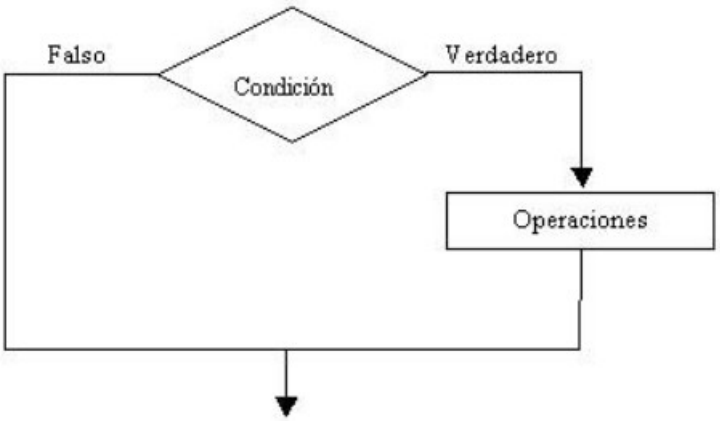

Diagrama estructura if

Proporciona a los programas la posibilidad de ejecutar selectivamente otras sentencias basándose en algún criterio booleano. Generalmente, la forma sencilla es

```
if (expresión booleana) sentencia;
```

Este formato es válido cuando se quiere ejecutar una sola sentencia dado que la condición es verdadera. Sin embargo, si se quiere usar más de una sentencia, se debe utilizar un bloque asociado al if para agruparlas. Su formato es:

```
if (expresión booleana) {
      [sentencias;]
}
```

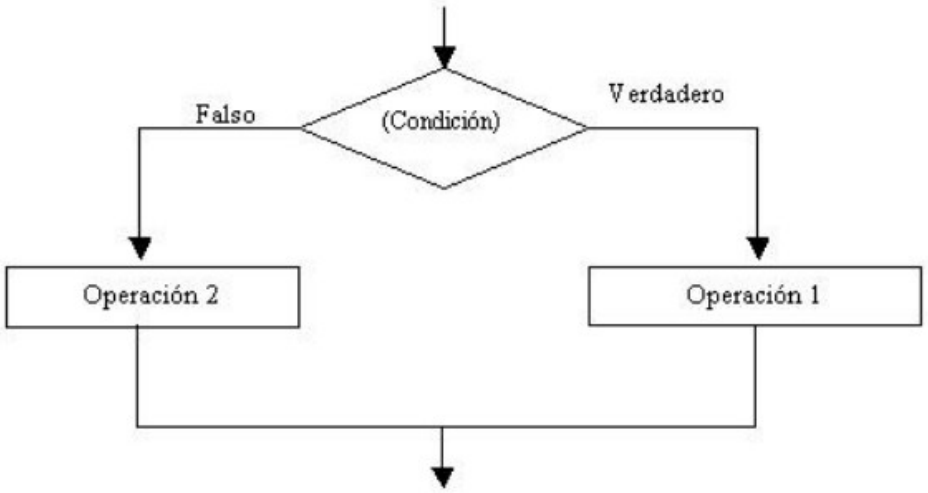

Diagrama estructura if – else

Cuando se quiere analizar también el caso en el cual la expresión booleana es falsa se puede utilizar la sentencia **else** . Por ejemplo:

```
if (expresión booleana) {
      [sentencias;]
}
else sentencia;
```

Nuevamente, esto es útil si se ejecutan varias sentencias si la condición es verdadera y tan solo una en caso de ser falsa.
Para el caso de ejecutar varias sentencias cuando la condición es falsa y, por ejemplo, una sentencia sola si es verdadera, se puede volver a utilizar la técnica de asociar un bloque de sentencias, por ejemplo:

```
if (expresión booleana) sentencia;
else {
      [sentencias;]
}
```

El último caso a analizar es cuando se ejecutan muchas sentencias en ambas situaciones, el formato es:

```
if (expresión booleana) {
      [sentencias;]
} else {
      [sentencias;]
}
```

Ejemplo de uso del condicional con bloque de sentencias en ambas instrucciones

```
if (res == 1) {
      . . .
      // Código para la acción en caso que res sea igual a 1
      . . .
} else {
      . . .
      // código para la acción en caso que res no sea igual a 1
      . .
}
```

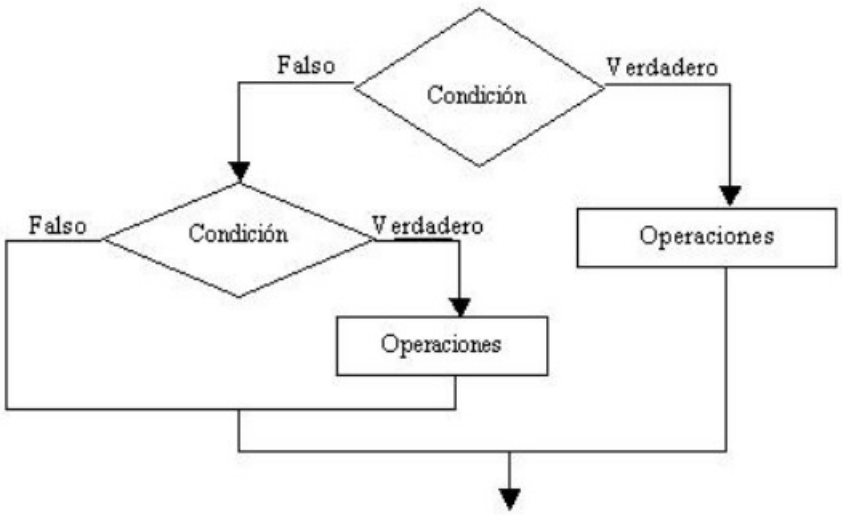

Diagrama if anidado

Existe otra forma de la sentencia else , else if que ejecuta una sentencia basada en otra expresión. No es más que una forma particular de combinar un if con un else (no es una sentencia diferente) y, aunque no es recomendable porque puede inducir a errores, muchos programadores la utilizan.

```
Uso de else if
int puntuacion;
String nota;
if (puntuacion >= 90) {
      nota = "Sobresaliente";
} else if (puntuacion >= 80) {
      nota = "Notable";
} else if (puntuacion >= 70) {
      nota = "Bien";
} else if (puntuacion >= 60) {
      nota = "Suficiente";
} else {
      nota = "Insuficiente";
}
```

Otro caso a tener en cuenta es el del anidamiento. Cuando se anidan sentencia if – else es importante tener en cuenta qué if se asocia con cuál else .
Para esto, se debe seguir la siguiente regla:
El else del anidamiento más interno se asocia con el primer if que encuentra, siguiendo esta asociación desde adentro hacia afuera.

```
if (expresión booleana) {
      [sentencias;]
      if (expresión booleana) sentencia;
      if (expresión booleana) {
            [sentencias;]
      } else {
            [sentencias;]
      }
} else {
[sentencias;]
}
```

Por convención la llave abierta '{' se coloca al final de la misma línea donde se encuentra la sentencia if - else y la llave cerrada '}' empieza una nueva línea con sangría (indentación) en la línea en la que se encuentra el if - else .

Sentencia switch

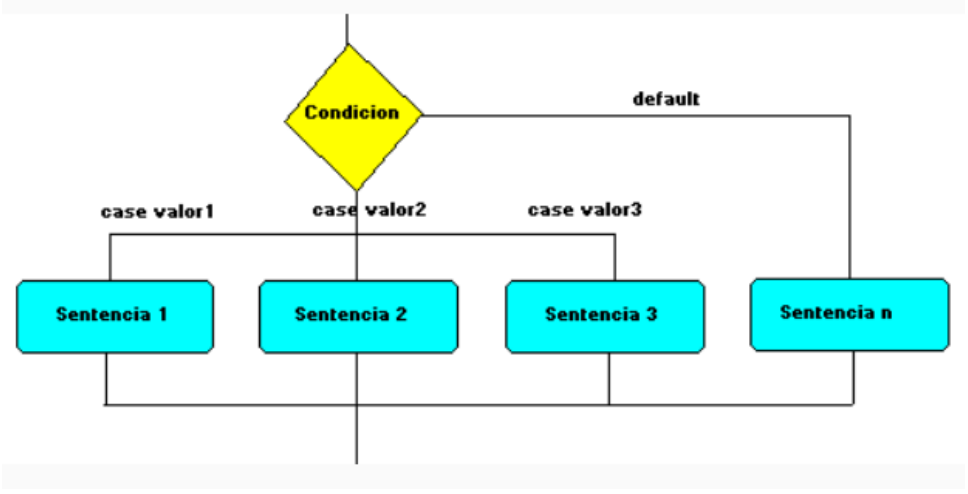

Diagrama estructura switch

La sintaxis de la sentencia switch es el siguiente:

```
switch ( expr1) {
      case constante2:
            [sentencias;]
      break;
      case constante3:
            [sentencias;]
      break;
      default:
            [sentencias;]
      break;
}
```

La sentencia switch se utiliza para ejecutar sentencias en base al valor que arroja una expresión. Por ejemplo, si un programa contiene un entero llamado mes cuyo valor indica el mes en alguna fecha y se quiere mostrar el nombre del mes basándose en su número entero equivalente, se podría utilizar la sentencia switch de Java para realizar esta tarea.

Uso del caso por defecto en el seleccionador de casos

```
int mes;
. . .
switch (mes) {
      case 1: System.out.println("Enero");     break;
      case 2: System.out.println("Febrero");   break;
      case 3: System.out.println("Marzo");     break;
      case 4: System.out.println("Abril");     break;
      case 5: System.out.println("Mayo");      break;
      case 6: System.out.println("Junio");     break;
      case 7: System.out.println("Julio");     break;
      case 8: System.out.println("Agosto");    break;
      case 9: System.out.println("Septiembre");break;
      case 10: System.out.println("Octubre");  break;
      case 11: System.out.println("Noviembre");break;
      case 12: System.out.println("Diciembre");break;
      default: System.out.println("Ese no es un mes válido!");
}
```

Otro punto interesante son las sentencias break después de cada case . La sentencia break hace que el control salga de la sentencia switch y continúe con la siguiente línea. La sentencia break es necesaria porque las sentencias case se siguen ejecutando hacia abajo. Esto es, sin un break explícito, el flujo de control seguiría secuencialmente a través de las sentencias case siguientes. En el ejemplo anterior, no se quiere que el flujo vaya de una sentencia case a otra, por eso se han tenido que poner las sentencias break .
Finalmente, puede utilizar la sentencia default al final de la sentencia switch para manejar los valores que no se han manejado explícitamente por una de las sentencias case .

Hay ciertos escenarios en los que es preferible que el control proceda secuencialmente a través de las sentencias case. Como las sentencias case no dependen de ningún orden en particular respecto de las constantes que evalúan para ejecutar las sentencias asociadas al case , se puede aprovechar la falta de orden y la no interrupción de la ejecución del código por un break.

```
int mes;
int numeroDias;
. . .
switch (mes) {
      case 1: System.out.println(“Caso 1”);
      case 3: case 5: case 7: case 8: case 10: case 12:
            numeroDias = 31; break;
      case 4: case 6: case 9: case 11:
            numeroDias = 30; break;
      case 2:
            if(((ano % 4 == 0) && !(ano % 100==0)) || ano % 400 == 0))
                  numeroDias = 29;
            else
                  numeroDias = 28;
            break;
}
```

El uso adecuado de cada break puede reducir notablemente la complejidad del código.

Expresiones switch JDK 12

Las expresiones *switch* permiten quitar varias sentencias *if else* encadenadas. Cada rama de la sentencia *siwtch* devuelve un valor y no hace falta usar la sentencia *break*, se pueden utilizar varios casos para cada rama.

```
switch (day) {
    case MONDAY, FRIDAY, SUNDAY -> System.out.println(6);
    case TUESDAY                -> System.out.println(7);
    case THURSDAY, SATURDAY     -> System.out.println(8);
    case WEDNESDAY              -> System.out.println(9);
}
```

```
String numericString = switch (integer) {
    case 0 -> "zero";
    case 1, 3, 5, 7, 9 -> "odd";
    case 2, 4, 6, 8, 10 -> "even";
    default -> "N/A";
};
```

La sentencia for

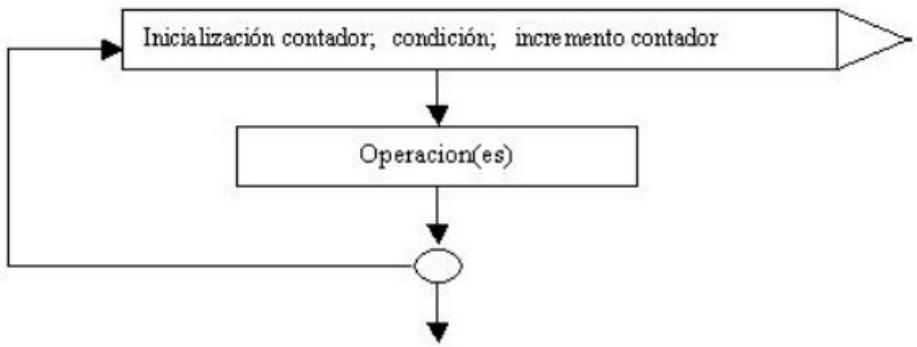

Diagrama de la sentencia for

Al igual que otras sentencias explicadas previamente, el for puede ejecutarse sobre una única sentencia o sobre un bloque asociado. La sintaxis del ciclo for es:

```
for (expresión inicial; expresión booleana; expresión) sentencia;

for (expresión inicial; expresión booleana; expresión){
      [sentencias;]
}
```

Donde:

expresión inicial: es la sentencia que se ejecuta una vez al iniciar el ciclo. Se suele utilizar para inicializar variables

expresión booleana: es una sentencia que determina cuando se termina el ciclo. Esta expresión se evalúa al principio de cada iteración en el ciclo. Cuando la expresión se evalúa a false el ciclo se termina.

expresión: se invoca en cada interacción del ciclo a partir del segundo ciclo (inclusive) en adelante.

Cualquiera (o todos) de estos componentes pueden ser una sentencia vacía (un punto y coma)

```
for (int i = 0; i < 10; i++) {
      System.out.println("Iterando i="+i);
}
//System.out.println("Valor de I: " + i); //Error
//No se puede imprimir i por que no existe fuera del ámbito del for
```

Por convención la llave abierta ' { ' se coloca al final de la misma línea donde se encuentra la sentencia for y la llave cerrada ' } ' empieza una nueva línea con sangría (indentación) en la línea en la que se encuentra el for .

Sentencia while

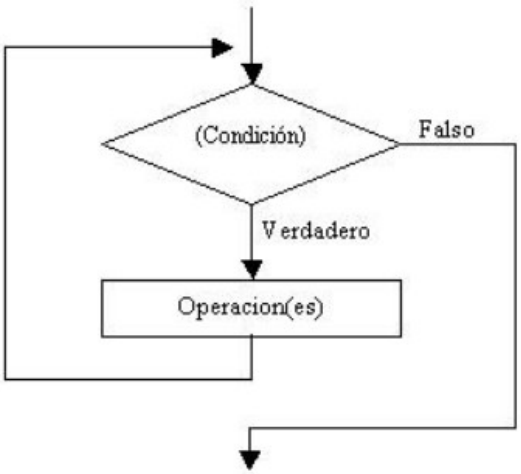

Diagrama estructura while

Una sentencia while realiza un ciclo mientras se cumpla una cierta condición (que la expresión que evalúa a un boolean sea verdadera). La sintaxis general de la sentencia while es:

```
while (expresión booleana) sentencia;

while (expresión booleana) {
      [sentencias;]
}

//Iterando con el ciclo while
int i = 0;
while (i < 10) {
      System.out.println("Iterando i:"+i);
      i++;
}
System.out.println("Fin");
System.out.println("i="+i);
//Es esta estructura la variable de control i es de ámbito global y puede
//imprimirse fuera de la estructura.
```

Sentencia do-while

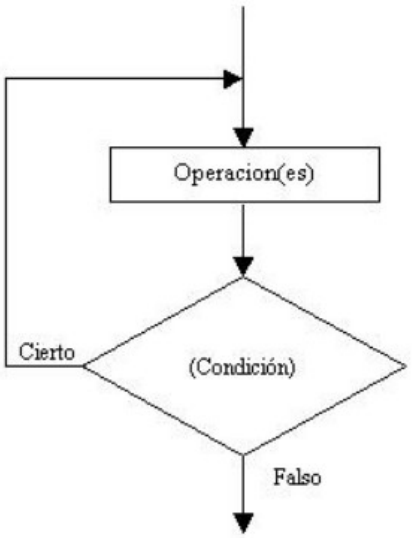

Diagrama Estructura do - while

Es conveniente utilizar la sentencia do – while cuando el ciclo debe ejecutarse al menos una vez. La sintaxis del ciclo do – while es:

```
do {
      [sentencias;]
} while (expresión booleana) ;     //Es la única sentencia que termina en ;

//Ciclo cuyas sentencias se ejecutan al menos una vez
int i = 0;
do {
      System.out.println("Iterando...");
      i++;
} while (i < 10);
System.out.println("Fin");
```

Casos especiales de control de flujo

Etiquetas en Sentencias if

Java admite la declaración de etiquetas a las cuales derivar el control de flujo de un programa siempre y cuando las mismas están asociadas a una sentencia o a un bloque de estas.
Las etiquetas tienen un beneficio dudoso y fomentan un estilo de programación de difícil mantenimiento. Sin embargo, como son cortes de control del flujo de los programas extremadamente rápidos, pueden ser utilizadas con sumo criterio cuando la velocidad de procesamiento sea crítica.
El Formato de declaración de una etiqueta es el siguiente:

```
Etiqueta: sentencia

O

Etiqueta:{
}
```

La forma más simple de utilizar etiquetas es asociándolas a sentencias if que rompe el control del flujo de programa mediante una sentencia break .

```
int contador = 0;
// Verificar break en una etiqueta asociada a un bloque
etiqueta1: {
      System.out.println("antes de etiqueta1");
      if (contador == 0) { break etiqueta1; }
      System.out.println("después de etiqueta1");
}
// Verificar break con una etiqueta asociada a una sentencia if
System.out.println("antes de etiqueta2");
etiqueta2: if (contador == 0) {
                  break etiqueta2;
            } else {
                  System.out.println("cláusulaelse;break no salta a etiqueta2");
            }
System.out.println("después de etiqueta2");

//Salida del programa por consola
//antes de etiqueta1
//antes de etiqueta2
//después de etiqueta2
```

Sentencias break y continue en los Ciclos

Como se mostró en la sentencia switch , la sentencia break hace que el control del flujo salte a la sentencia siguiente respecto de la actual en ejecución.
El caso más simple de break interrumpe la sentencia en ejecución. Si se ejecuta dentro de un ciclo que tiene asociado un bloque, como por ejemplo

```
do {
      sentencia;
      if (expresión booleana) {
            break;
      }
      sentencia;
} while (expresión booleana);
Sentencia_fuera_de_bloque;
```

La sentencia continue es similar pero a diferencia de break , hace que se ejecute el próximo ciclo de una iteración.
Ambas sentencias interrumpen el flujo normal de la secuencia de sentencias, pero además, se las puede asociar a una etiqueta de manera que dicha interrupción derive el flujo del programa a la sentencia asociada a la etiqueta
El caso más simple de continue es forzar a un nuevo ciclo cuando se ejecuta ignorando el resto de las sentencias que se encuentran en ese ciclo posteriores a él. Si por ejemplo se tiene el siguiente formato:

```
do {
      sentencia;
      if (expresión booleana) {
            continue;
      }
      sentencia;
} while (expresión booleana);
```

Vectores

Declaración

En Java, como en cualquier otro lenguaje, un vector agrupa datos del mismo tipo bajo un mismo nombre. La salvedad en este lenguaje es que los datos pueden ser tipos primitivos o referencias. Por otra parte, los vectores en sí mismos son referenciados. Esto quiere decir que cuando se declara un vector en realidad primero se declara una referencia a éste que debe ser inicializada posteriormente para poder obtener el vector en sí mismo. La forma de declarar referencias a vectores en Java es:

```
      char s[];
      Point p[];
      char[] s;
      Point[] p;
```

Se debe tener en cuenta que declaraciones como las anteriores sólo crea espacio para una referencia.
Para obtener el espacio de almacenamiento de los elementos del vector, este deberá ser solicitado en tiempo de ejecución, ya que un vector es un objeto y debe ser creado con new .

Creación

Cuando se crea un vector a partir de la referencia a este, se ejecuta el operador new . Por ejemplo, para un vector de un tipo primitivo (char)

```
char[] s;
s = new char[26];
for ( int i=0; i<26; i++) {
      s[i] = (char) ('A' + i);
}
```

```
char[] s;
```

Java reserva un espacio en memoria para la referencia s. Para ser más preciso, reserva un espacio en el stack para la referencia, pero todavía no existe un espacio de almacenamiento para los elementos del vector.

```
s = new char[26];
```

Java reserva dinámicamente en memoria el espacio de almacenamiento para los 26 elementos del vector.

Inicialización

Existen dos formas de inicializar los vectores: elemento a elemento (como se mostró anteriormente) o por un bloque de inicialización.
Cuando se inicializa elemento a elemento se necesita crear primero el espacio para almacenar los elementos del vector para luego asignar cada uno de ellos, tanto para los tipos primitivos como para los tipos referenciados. Por ejemplo, para crear e inicializar un vector de elementos primitivos se puede codificar lo siguiente

```
char[] s;
s = new char[4];
s[0] = 'A';
s[1] = 'B';
s[2] = 'C';
s[3] = 'D';
```

Sin embargo, los elementos de un vector de referencia necesitan un poco más de trabajo, ya que se debe crear en cada elemento el objeto acerca del cual se debe almacenar su referencia como un elemento del vector

```
Point[] p = new Point[4];
p[0] = new Point(0, 1);
p[1] = new Point(1, 2);
p[2] = new Point(2, 3);
p[3] = new Point(3, 4);
```

Cuando se inicializa un vector con un bloque de asignación, no se ve claramente que el espacio de almacenamiento se asigna como se lo hace con un objeto. Sin embargo el proceso es el mismo pero en lugar de declararlo explícitamente, el lenguaje lo hace automáticamente. Para el caso de tipos primitivos el código es el que se muestra a continuación. Notar el punto y coma luego del bloque. Esto es lo que diferencia un bloque de asignación de uno de declaración.

```
char[] s = { 'F', 'G', 'H', 'I' };
```

Los tipos referenciados exigen (Objetos) el esfuerzo adicional de hacer un new para cada elemento de manera de crear así el objeto que será referenciado.

```
Point[] p = {
      new Point(4, 5),
      new Point(5, 6),
      new Point(6, 7),
      new Point(7, 8)
};
```

Matrices o Vectores Bidimensionales

Java es un lenguaje que no fue pensado para realizar grandes operaciones con vectores multidimensionales. En realidad el lenguaje permite manejar dos dimensiones de cualquier tipo de datos, primitivos y referenciados.
El concepto de la declaración de la referencia se mantiene en estos casos, con la salvedad que en lugar de un par de corchetes se utilizan dos, de manera que el primer par especifica la primera dimensión y el otro la segunda.
Los vectores de dos dimensiones o matrices deben pensarse como un vector de vectores, de manera que cada elemento del primer vector mantiene una referencia al primer elemento del vector de la segunda dimensión, por lo tanto, cualquier intento de inicializar la segunda dimensión antes que la primera será un error en tiempo de compilación. Un ejemplo de las posibles declaraciones válidas y el error descripto se muestra a continuación

```
int dosDim[][] = new int [4][];
dosDim[0] = new int[5];
dosDim[1] = new int[5];
int dosDim[][] = new int [][4]; //ilegal
```

Las matrices no necesitan ser cuadradas en su creación. Recordando el concepto de considerarlas vectores de vectores, cada vector de la segunda dimensión se puede crear con los elementos que se necesiten. Por ejemplo:

```
dosDim[0] = new int[2];
dosDim[1] = new int[4];
dosDim[2] = new int[6];
dosDim[3] = new int[8];
```

También el lenguaje permite la declaración de matrices cuadradas como se hace tradicionalmente en cualquier lenguaje. Si se necesitara crear un vector de 4 vectores de 5 enteros cada uno, la declaración sería

```
int dosDim [][] = new int[4][5];
```

Límite de un Vector

Lo siguiente deseable al declarar e inicializar un vector, tenga una o dos dimensiones, es recorrerlo. Para ello los principales temas a tener en cuenta son dónde empieza y hasta dónde llega. El primer tema es fácil, ya que todos los vectores en Java comienzan por el elemento cuyo subíndice es 0. El segundo tema no es más difícil, puesto que el lenguaje brinda la posibilidad de acceder a una variable que contiene la cantidad de elementos que el vector tiene la capacidad de almacenar.
Luego, si todos los elementos del vector poseen valores iniciales, recúrrelo se limita a establecer los límites superiores e inferiores y usar un ciclo de iteración. Se debe tener especial cuidado cuando los elementos del vector sean referencias a objetos que los mismo guarden una referencia válida a un objeto antes de utilizarlo o generará un error en tiempo de ejecución.
Un ejemplo de recorrido de un vector, que sólo contiene los valores iniciales que le da el lenguaje a las variables de tipo entero en cada elemento, es el siguiente:

```
int list[] = new int [10];
for (int i = 0; i < list.length; i++) {
      System.out.println(list[i]);
}
```

Redimensionamiento de un Vector

En Java no se puede cambiar el tamaño de la dimensión establecida en la declaración y creación de un vector. Si se intentara hacer esto, simplemente se crearía un nuevo vector que se asignaría a la variable de referencia y el vector anterior pasaría a ser seleccionable para el recolector de basura (quien se encarga de liberar la memoria no utilizada), perdiendo así su contenido. Este hecho se refleja en el siguiente ejemplo, en el cual el vector de 6 elementos es seleccionable para el recolector de basura cuando se asigna a la variable de referencia el vector de 10 elementos:

```
int miVec[] = new int[6];
miVec = new int[10];
```

Copia de Vectores

La alternativa que brinda al lenguaje al redimensionamiento es hacerlo explícitamente (en lugar de automáticamente) copiando el contenido en un nuevo vector y reasignando su contenido a la referencia. Esto, si bien es un nuevo vector, tiene como resultado final un “redimensionamiento”. Para lograrlo, se provee un método que permite copiar un vector en otro a través de lo especificado en sus argumentos. Los argumentos del método arraycopy son los siguientes:

1. Vector fuente de los datos a copiar.
2. Elemento desde el cual empieza a leerse para copiar.
3. Vector destino de la copia.
4. Elemento a partir del cual se empieza a escribir lo que se lee del primer vector.
5. Cantidad de elementos a copiar.

Por ejemplo

```
// vector original
int vec1[] = { 1, 2, 3, 4, 5, 6 };
// nuevo vector más largo
int vec2[] = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };
// copiar todos los elementos del vector vec1
// en el vector vec2 comenzando por el subíndice 0
System.arraycopy(vec1, 0, vec2, 0, vec1.length);
```

Ordenamiento de vectores

La clase Arrays provee el método **.sort()** que recibe como parámetro un vector ordena. El método **.reverse()** de la misma clase Arrays invierte un vector.

```
int[] numbers = {6,9,1,8};
Arrays.sort(numbers);       // [1,6,8,9]
Arrays.reverse(numbers);    // [9,8,6,1]
```

Búsqueda de valores en un vector

La clase Arrays tiene el método .binarySearch(vector, valor) que recibe como parámetros un vector y un valor, este método devuelve un entero, indicando el indice del valor buscado, y en caso de no encontrarse ese valor devuelve un valor entero.

```
int[] numbers = {6,9,1,8};
System.out.println(Arrays.binarySearch(numbers, 6)); // 1
System.out.println(Arrays.binarySearch(numbers, 3)); // -2
```

Argumentos desde la línea de comando

Una aplicación puede recibir valores en formato de caracteres desde el sistema operativo. Estos valores se los denomina “argumentos desde la línea de comandos”, puesto que es el intérprete de comandos el encargado de pasarlos a la aplicación.
En el caso de Java, es la máquina virtual la que interacciona con el sistema operativo, sin embargo, a través de ella se pueden recibir dichos argumentos porque el espacio virtual de ejecución que genera emula esta interacción.
Los puede utilizar cualquier aplicación Java siempre y cuando respete el formato definido para pasar los mismos.
Cada argumento de la línea de comando se coloca en el vector args y es pasado a main :

```
public static void main(String[] args)
```

Los argumentos, se colocan en la línea de comando cuando se invoca al intérprete luego del nombre de la clase:

```
java TestArgs arg1 arg2 "otro arg"
```