Unidad II - Arrays

- Arrays de Una Dimensión
  - Utilizar sólo un Inicializador
  - Utilizar sólo la Palabra Clave "new"
  - Utilizar la palabra clave "new" y un Inicializador
- Trabajar con un array uni-dimensional
- Algoritmos de búsqueda-lineal, búsqueda-binaria y ordenación de burbuja
  - Búsqueda Lineal
  - Búsqueda Binaria
  - Ordenación de Burbuja
- Arrays de Dos Dimensiones
  - Utilizar sólo un Inicializador
  - Utilizar sólo la palabra clave "new"
  - Utilizar la palabra clave "new" y un inicializador

- Java en castellano (Tutorial Estructuras de Datos y Algoritmos en Java)
  http://www.programacion.com/java/tutorial/jap_data_alg/
- Tutorial original en Inglés
  http://www.javaworld.com/

Arrays

El array es una de las estructuras de datos más ampliamente utilizada por su flexibilidad para derivar en complejas estructuras de datos y su simplicidad. Empezaremos con una definición: *un **array** es una secuencia de elementos, donde cada **elemento** (un grupo de bytes de memoria que almacenan un único ítem de datos) se asocia con al menos un **índice** (entero no-negativo).* Esta definición lanza cuatro puntos interesantes:

- Cada elemento ocupa el mismo número de bytes; el número exacto depende del tipo de datos del elemento.
- Todos los elementos son del mismo tipo.
- Tendemos a pensar que los elementos de un array ocupan localizaciones de memoria consecutivas. Cuando veamos los arrays bi-dimensionales descubrirá que no es siempre así.
- El número de índices asociados con cada elemento es la *dimensión* del array

Nota:

Esta sección se enfoca exclusivamente en arrays de una y dos dimensiones porque los arrays de más dimensiones no se utilizan de forma tan frecuente.

Arrays de Una Dimensión

El tipo de array más simple tiene una dimensión: cada elemento se asocia con un único índice. Java proporciona tres técnicas para crear un array de una dimensión: usar sólo un inicializador, usar sólo la palabra clave `new`, y utilizar la palabra clave `new` con un inicializador.

Utilizar sólo un Inicializador

Utilizando un inicializador se puede utilizar cualquiera de estas dos sintaxis:

```
type variable_name '[' ']' [ '=' initializer ] ';'
type '[' ']' variable_name [ '=' initializer ] ';'
```

Donde el inicializador tiene la siguiente sintaxis:

```
'{' initial_value1 ',' initial_value2 ',' ... '}'
```

El siguiente fragmento ilustra como crear un array de animales:

```
// Create an array of animals.
String animals [] = { "Tiger", "Zebra", "Kangaroo" };
```

Utilizar sólo la Palabra Clave "new"

Utilizando la palabra clave `new` se puede utilizar cualquiera de estas dos sintaxis:

```
type variable_name '[' ']' '=' 'new' type '[' integer_expression ']' ';'
type '[' ']' variable_name '=' 'new' type '[' integer_expression ']' ';'
```

Para ambas sintaxis:

- `variable_name` especifica el nombre de la variable del array uni-dimensional
- `type` especifica el tipo de cada elemento. Como la variable del array uni-dimensional contiene una referencia a un array uni-dimensional, el tipo es `type []`
- La palabra clave `new` seguida por `type` y seguida por `integer_expression` entre corchetes cuadrados ([]) específica el número de elementos. `new` asigna la memoria para los elementos del array uni-dimensional y pone ceros en todos los bits de los bytes de cada elemento, lo que significa que cada elemento contiene un valor por defecto que interpretamos basándonos en su tipo.
- = asigna la referencia al array uni-dimensional a la variable `variable_name`.

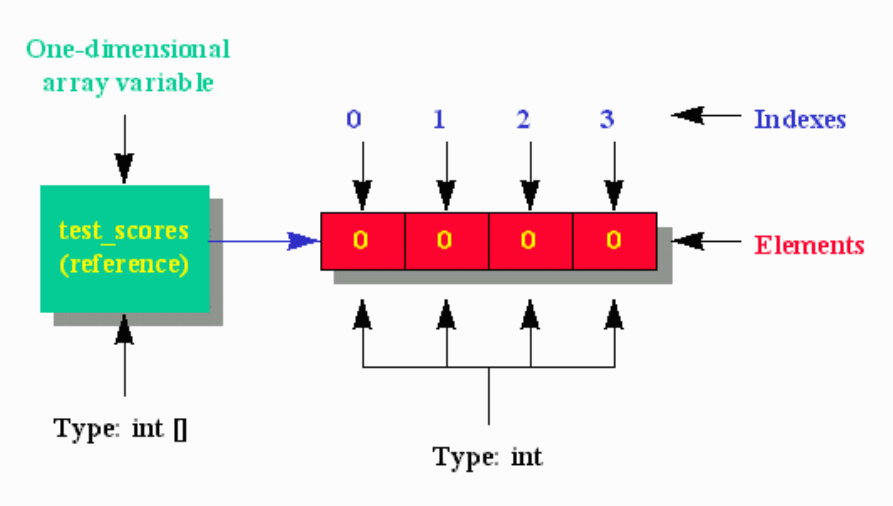
Truco:

Los desarrolladores Java normalmente sitúan los corchetes cuadrados después del tipo (`int [] test_scores`) en vez de después del nombre de la variable (`int test_scores []`) cuando declaran una variable array. Mantener toda la información del tipo en un único lugar mejora la lectura del código.

El siguiente fragmento de código utiliza sólo la palabra clave `new` para crear un array uni-dimensional que almacena datos de un tipo primitivo:

```
int [] test_scores = new int [4];
```

`int [] test_scores` declara una variable array uni-dimensional (`test_scores`) junto con su tipo de variable (`int []`). El tipo de referencia `int []` significa que cada elemento debe contener un ítem del tipo primitivo entero. `new int [4]` crea una array uni-dimensional asignando memoria para cuatro elementos enteros consecutivos. Cada elemento contiene un único entero y se inicializa a cero. El operador *igual a* (`=`) asigna la referencia del array uni-dimensional a `test_scores`. La siguiente figura ilustra los elementos y la variable array uni-dimensional resultante:



Cuidado:

Cuando se crea un array uni-dimensional basado en un tipo primitivo, el compilador requiere que aparezca la palabra clave que indica el tipo primitivo en los dos lados del operador igual-a. De otro modo, el compilador lanzará un error. Por ejemplo, `int [] test_scores = new long [20];` es ilegal porque las palabras claves `int` y `long` representan tipos primitivos incompatibles.

Los arrays uni-dimensionales de tipos primitivos almacenan datos que son valores primitivos. Por el contrario, los arrays uni-dimensiones del tipo referencia almacenan datos que son referencias a objetos. El siguiente fragmento de código utiliza la palabra clave `new` para crear una pareja de arrays uni-dimensionales que almacenan datos basados en tipo referencia:

```
Clock [] c1 = new Clock [3];
Clock [] c2 = new AlarmClock [3];
```

`Clock [] c1 = new Clock [3];` declara una variable array uni-dimensional, (`c1`) del tipo `Clock []`, asigna memoria para un array uni-dimensional `Clock` que consta de tres elementos consecutivos, y asigna la referencia del array `Clock` a `c1`. Cada elemento debe contener una referencia a un objeto `Clock` (asumiendo que `Clock` es una clase concreta) o un objeto creado desde una subclase de `Clock` y lo inicializa a `null`.

`Clock [] c2 = new AlarmClock [3];` se asemeja a la declaración anterior, excepto en que se crea un array uni-dimensional `AlarmClock`, y su referencia se asigna a la variable `Clock []` de nombre `c2`. (Asume `AlarmClock` como subclase de `Clock`.)

Utilizar la palabra clave "new" y un Inicializador

Utilizar la palabra clave `new` con un inicializador requiere la utilización de alguna de las siguientes sintaxis:

```
type variable_name> '[' ']' '='  'new' type '[' ']' initializer ';'
type '[' ']' variable_name '='  'new' type '[' ']' initializer ';'
```

Donde *`initializer`* tiene la siguiente sintaxis:

```
'{' [ initial_value [ ',' ... ] ] '}'
```

- `variable_name` especifica el nombre de la variable del array uni-dimensional
- `type` especifica el tipo de cada elemento. Como la variable del array uni-dimensional contiene una referencia a un array uni-dimensional, el tipo es `type []`
- La palabra clave `new` seguida por `type` y seguida por corchetes cuadrados ([]) vacíos, seguido por `inicializer`. No se necesita especificar el número de elementos entre los corchetes cuadrados porque el compilador cuenta el número de entradas en el inicializador. `new` asigna la

memoria para los elementos del array uni-dimensional y asigna cada una de las entradas del inicializador a un elemento en orden de izquierda a derecha.

- = asigna la referencia al array uni-dimensional a la variable `variable_name`.

Nota:

Un array uni-dimensional (o de más dimensiones) creado con la palabra clave `new` con un inicializador algunas veces es conocido como un *array anónimo*.

El siguiente fragmento de código utiliza la palabra clave `new` con un inicializador para crear un array uni-dimensional con datos basados en tipos primitivos:

```
int [] test_scores = new int [] { 70, 80, 20, 30 };
```

`int [] test_scores` declara una variable de array uni-dimensional (`test_scores`) junto con su tipo de variable (`int []`). El código `new int [] { 70, 80, 20, 30 }` crea un array uni-dimensional asignando memoria para cuatro elementos enteros consecutivos; y almacena `70` en el primer elemento, `80` en el segundo, `20` en el tercero, y `30` en el cuarto. La referencia del array uni-dimensional se asigna a `test_scores`.

Cuidado:

No especifique una expresión entera entre los corchetes cuadrados del lado derecho de la igualdad. De lo contrario, el compilador lanzará un error. Por ejemplo, `new int [3] { 70, 80, 20, 30 }` hace que el compilador lance un error porque puede determinar el número de elementos partiendo del inicializador. Además, la discrepancia está entre el número 3 que hay en los corchetes y las cuatro entradas que hay en el inicializador.

La técnica de crear arrays uni-dimensionales con la palabra clave `new` y un inicializador también soporta la creación de arrays que contienen referencias a objetos. El siguiente fragmento de código utiliza esta técnica para crear una pareja de arrays uni-dimensionales que almacenan datos del tipo referencia:

```
Clock [] c1 = new Clock [] { new Clock () };
Clock [] c2 = new AlarmClock [] { new AlarmClock () };
```

`Clock [] c1 = new Clock [3];` declara una variable de array uni-dimensional (`c1`) del tipo `Clock []`, asigna memoria para un array `Clock` que consta de un sólo elemento, crea un objeto `Clock` y asigna su referencia a este elemento, y asigna la referencia del array `Clock` a `c1`. El código `Clock [] c2 = new AlarmClock [3];` se parece a la declaración anterior, excepto en que crea un array uni-dimensional de un sólo elemento `AlarmClock` que inicializa un objeto del tipo `AlarmClock`.

Trabajar con un array uni-dimensional

Después de crear un array uni-dimensional, hay que almacenar y recuperar datos de sus elementos. Con la siguiente sintaxis se realiza esta tarea:

```
variable_name '[' integer_expression ']'
```

`integer_expression` indentifica un índice de elemento y debe evaluarse como un entero entre 0 y uno menos que la longitud del array uni-dimensional (que devuelve `variable_name.length`). Un índice menor que 0 o mayor o igual que la longitud causa que se lance una `ArrayIndexOutOfBoundsException`. El siguiente fragmento de código ilustra accesos legales e ilegales a un elemento:

```
String [] months = new String [] { "Jan", "Feb", "Mar", "Apr", "May", "Jun"
                                   "Jul", "Aug", "Sep", "Oct", "Nov", "Dec" };

System.out.println (months [0]); // Output: Jan
// The following method call results in an ArrayIndexOutOfBoundsException
// because index equals the month's length
System.out.println (months [months.length]);
```

```
System.out.println (months [months.length - 1]); // Output: Dec
// The following method call results in an ArrayIndexOutOfBoundsException
// because index < 0
System.out.println (months [-1]);
```

Ocurre una situación interesante cuando se asigna la referencia de una subclase de un array a una variable de array de la superclase, porque un subtipo de array es una clase del supertipo de array. Si intenta asignar una referencia de un objeto de la superclase a los elementos del array de la subclase, se lanza una `ArrayStoreException`. El siguiente fragmento demuestra esto:

```
AlarmClock [] ac = new AlarmClock [1];
Clock [] c = ac;
c [0] = new Clock ();
```

El compilador no dará ningún error porque todas las líneas son legales. Sin embargo, durante la ejecución, `c [0] = new Clock ();` resulta en una `ArrayStoreException`. Esta excepción ocurre porque podríamos intentar acceder a un miembro específico de `AlarmClock` mediante una referencia a un objeto `Clock`. Por ejemplo, supongamos que `AlarmClock` contiene un método `public void soundAlarm()`, `Clock` no lo tiene, y el fragmento de código anterior se ejecuta sin lanzar una `ArrayStoreException`. Un intento de ejecutar `ac [0].soundAlarm ();` bloquea la JVM *Máquina Virtual Java* porque estamos intentando ejecutar este método en el contexto de un objeto `Clock` (que no incorpora un método `soundAlarm()`).

Cuidado:

Tenga cuidado cuando acceda a los elementos de un array porque podría recibir una ArrayIndexOutOfBoundsException o una ArrayStoreException.

Algoritmos de búsqueda-lineal, búsqueda-binaria y ordenación de burbuja

Los desarrolladores normalmente escriben código para buscar datos en un array y para ordenar ese array. Hay tres algoritmos muy comunes que se utilizan para realizar estas tareas.

Búsqueda Lineal

El algoritmo de búsqueda lineal busca en un array uni-dimensional un dato específico. La búsqueda primero examina el elemento con el índice 0 y continua examinando los elementos sucesivos hasta que se encuentra el ítem o no quedan más elementos que examinar. El siguiente pseudocódigo demuestra este algoritmo:

```
DECLARE INTEGER i, srch = 72
DECLARE INTEGER x [] = [ 20, 15, 12, 30, -5, 72, 456 ]
FOR i = 0 TO LENGTH (x) - 1
    IF x [i] IS srch THEN
       PRINT "Found ", srch
       END
    END IF
NEXT i
PRINT "Did not find ", srch
END
```

El siguiente listado presenta el equivalente Java al pseudocódigo anterior:

```
// LSearchDemo.java

class LSearchDemo  {
   public static void main (String [] args) {
      int i, srch = 72;
      int [] x = { 20, 15, 12, 30, -5, 72, 456 };

      for (i = 0; i <= x.length - 1; i++)
           if (x [i] == srch)  {
               System.out.println ("Found " + srch);
               return;
```

```
            }

         System.out.println ("Did not find " + srch);
   }
}
```

`LSearchDemo` produce la siguiente salida:

```
Found 72
```

Dos de las ventajas de la búsqueda lineal son la simplicidad y la habilidad de buscar tanto arrays ordenados como desordenados. Su única desventaja es el tiempo empleado en examinar los elementos. El número medio de elementos examinados es la mitad de la longitud del array, y el máximo número de elementos a examinar es la longitud completa. Por ejemplo, un array uni-dimensional con 20 millones de elementos requiere que una búsqueda lineal examine una media de 10 millones de elementos y un máximo de 20 millones. Como este tiempo de examen podría afectar seriamente al rendimiento, utilice la búsqueda lineal para arrays uni-dimensionales con relativamente pocos elementos.

Búsqueda Binaria

Al igual que en la búsqueda lineal, el algoritmo de búsqueda binaria busca un dato determinado en un array uni-dimensional. Sin embargo, al contrario que la búsqueda lineal, la búsqueda binaria divide el array en sección inferior y superior calculando el índice central del array. Si el dato se encuentra en ese elemento, la búsqueda binaria termina. Si el dato es numéricamente menor que el dato del elemento central, la búsqueda binaria calcula el índice central de la mitad inferior del array, ignorando la sección superior y repite el proceso. La búsqueda continua hasta que se encuentre el dato o se exceda el límite de la sección (lo que indica que el dato no existe en el array). El siguiente pseudocódigo demuestra este algoritmo:

```
DECLARE INTEGER x [] = [ -5, 12, 15, 20, 30, 72, 456 ]
DECLARE INTEGER loIndex = 0
DECLARE INTEGER hiIndex = LENGTH (x) - 1
DECLARE INTEGER midIndex, srch = 72

WHILE loIndex <= hiIndex
   midIndex = (loIndex + hiIndex) / 2

   IF srch > x [midIndex] THEN
      loIndex = midIndex + 1
   ELSE
   IF srch < x [midIndex] THEN
      hiIndex = midIndex - 1
   ELSE
      EXIT WHILE
   END IF
END WHILE

IF loIndex > hiIndex THEN
   PRINT srch, " not found"
ELSE
   PRINT srch, " found"
END IF

END
```

El siguiente código representa el equivalente Java del pseudocódigo anterior:

```
// BSearchDemo.java

class BSearchDemo {
   public static void main (String [] args) {
      int [] x = { -5, 12, 15, 20, 30, 72, 456 };
      int loIndex = 0;
      int hiIndex = x.length - 1;
      int midIndex, srch = 72;

      while (loIndex <= hiIndex)  {
```

```
            midIndex = (loIndex + hiIndex) / 2;
            if (srch > x [midIndex])
                loIndex = midIndex + 1;
            else if (srch < x [midIndex])
                hiIndex = midIndex - 1;
            else
                break;
        }
        if (loIndex > hiIndex)
            System.out.println (srch + " not found");
        else
            System.out.println (srch + " found");
    }
}
```

`BSearchDemo` produce la siguiente salida:

```
72 found
```

La única ventaja de la búsqueda binaria es que reduce el tiempo empleado en examinar elementos. El número máximo de elementos a examinar es `log2n` (donde `n` es la longitud del array uni-dimensional). Por ejemplo, un array uni-dimensional con 1.048.576 elementos requiere que la búsqueda binaria examine un máximo de 20 elementos. La búsqueda binaria tiene dos inconvenientes; el incremento de complejidad y la necesidad de pre-ordenar el array.

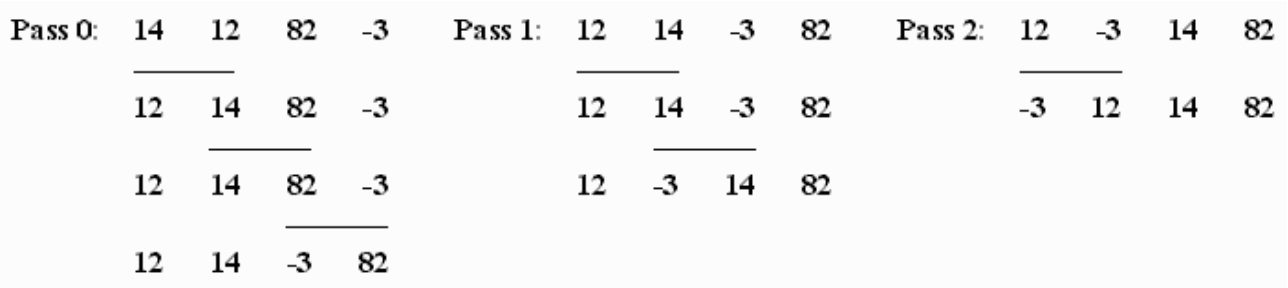
Ordenación de Burbuja

Cuando entra en juego la ordenación de datos, la ordenación de burbuja es uno de los algoritmos más simples. Este algoritmo hace varios pases sobre un array uni-dimensional. Por cada pase, el algoritmo compara datos adyacentes para determinar si numéricamente es mayor o menor. Si el dato es mayor (para ordenaciones ascendentes) o menor (para ordenaciones descendientes) los datos se intercambian y se baja de nuevo por el array. En el último pase, el dato mayor (o menor) se ha movido al final del array. Este efecto "burbuja" es el origen de su nombre. El siguiente pseudocódigo demuestra este algoritmo (en un contexto de ordenación ascendente):

```
DECLARE INTEGER i, pass
DECLARE INTEGER x [] = [ 20, 15, 12, 30, -5, 72, 456 ]

FOR pass = 0 TO LENGTH (x) - 2
    FOR i = 0 TO LENGTH (x) - pass - 2
        IF x [i] > x [i + 1] THEN
           SWAP x [i], x [i + 1]
        END IF
    NEXT i
NEXT pass
END
```

La siguiente figura muestra una ordenación de burbuja ascendente de un array uni-dimensional de cuatro elementos. Hay tres pasos, el paso 0 realiza tres comparaciones y dos intercambios, el paso 1 realiza dos comparaciones y un intercambio y el paso realiza una comparación y un intercambio.



El siguiente listado presenta el equivalente Java del pseudocódigo anterior:

```
// BSortDemo.java
```

```
class BSortDemo {

   public static void main (String [] args) {
      int i, pass;
      int [] x = { 20, 15, 12, 30, -5, 72, 456 };

       for (pass = 0; pass < = x.length - 2; pass++)
           for (i = 0; i < = x.length - pass - 2; i++)
                if (x [i] >  x [i + 1]) {
                    int temp = x [i];
                    x [i] = x [i + 1];
                    x [i + 1] = temp;
                }

      for (i = 0; i <  x.length; i++)
           System.out.println (x [i]);
   }
}
```

`BSortDemo` produce la siguiente salida:

```
-5
12
15
20
30
72
456
```

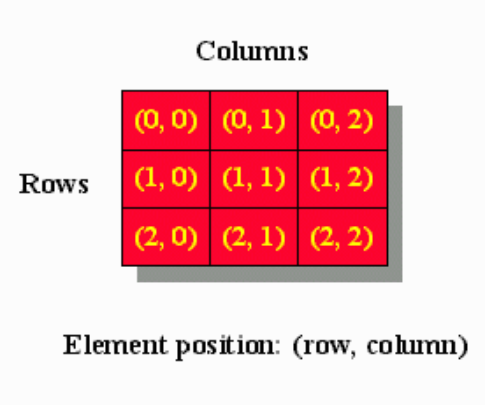
Aunque la ordenación de burbuja es uno de los algoritmos de ordenación más simples, también es uno de los más lentos. Entre los algoritmos más rápidos se incluyen la ordenación rápida y la ordenación de pila.

Truco:

Otro algoritmo muy utilizado para arrays uni-dimensionales copia los elementos de un array fuente en otro array de destino. En vez de escribir su propio código para realizar esta tarea puede utilizar el método `public static void arraycopy(Object src, int srcindex, Object dst, int dstindex, int length)` de la clase `java.lang.System`, que es la forma más rápida de realizar la copia.

Arrays de Dos Dimensiones

Un array de dos dimensiones, también conocido como *tabla* o *matriz,* donde cada elemento se asocia con una pareja de índices, es otro array simple. Conceptualizamos un array bi-dimensional como una cuadrícula rectangular de elementos divididos en filas y columnas, y utilizamos la notación `(fila, columna)` para identificar un elemento específico. La siguiente figura ilustra esta visión conceptual y la notación específica de los elementos:



Java proporciona tres técnicas para crear un array bi-dimensional:

Utilizar sólo un Inicializador

Esta técnica requiere una de estas sintaxis:

```
type variable_name '[' ']' '[' ']' '='   '{' [ rowInitializer [ ',' ... ] ] '}' ';'
type '[' ']' '[' ']' variable_name '='  '{' [ rowInitializer [ ',' ... ] ] '}' ';'
```

Donde *`rowInitializer`* tiene la siguiente sintaxis:

```
'{' [ initial_value [ ',' ... ] ] '}'
```

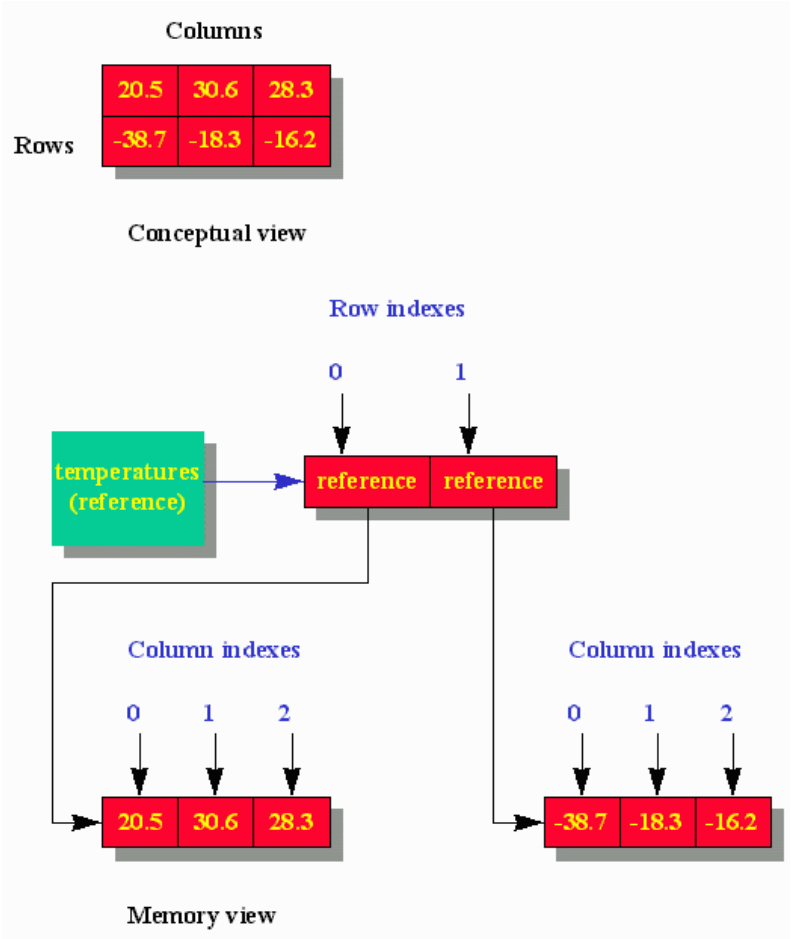
Para ambas sintaxis:

- *`variable_name`* especifica el nombre de la variable del array bi-dimensional.
- *`type`* especifica el tipo de cada elemento. Como una variable de array bi-dimensional contiene una referencia a un array bi-dimensional, su tipo es `type [ ] [ ]`.
- Especifica cero o más inicializadores de filas entre los corchetes ({ }). Si no hay inicializadores de filas, el array bi-dimensional está vacío. Cada inicializador de fila especifica cero o más valores iniciales para las entradas de las columnas de esa fila. Si no se especifican valores para esa fila, la fila está vacía.
- = se utiliza para asignar la referencia del array bi-dimensional a *`variable_name`*.

El siguiente código usa sólo un inicializador para crear un array bi-dimensional que almacena datos basados en un tipo primitivo:

```
double [][] temperatures = { { 20.5, 30.6, 28.3 },
                             { -38.7, -18.3, -16.2 } }; // Celsius temperatures
```

`double [][] temperatures` declara una variable de array bi-dimensional (`temperatures`) junto con su tipo de variable (`double [][]`). El tipo de referencia `double [][]` significa que cada elemento debe contener datos del tipo primitivo `double`. `{ { 20.5, 30.6, 28.3 }, { -38.7, -18.3, -16.2 } }` especifica un array bi-dimensional de dos filas por tres columnas, donde la primera fila contiene los datos `20.5`, `30.6`, y `28.3`, y la segunda fila contiene los datos `-38.7`, `-18.3`, y `-16.2`. Detrás de la escena, se asigna memoria y se inicializan esto datos. El operador igual-a asigna la referencia del array bi-dimensional a `temperatures`. La siguiente figura ilustra el array bi-dimensional resultante desde un punto de vista conceptual y de memoria.



Utilizar sólo la palabra clave "new"

Esta técnica requiere cualquiera de estas sintaxis:

```
type variable_name '[' ']' '[' ']' '=' 'new' type '[' integer_expression ']' '['
']' ';'

type '[' ']' '[' ']' variable_name '=' 'new' type '[' integer_expression ']' '['
']' ';'
```

En ambas sintaxis:

- *`variable_name`* especifica el nombre de la variable del array bi-dimensional.
- *`type`* especifica el tipo de cada elemento. Como es una variable de array bi-dimensional contiene una referencia a un array bi-dimensional, su tipo es `type [ ] [ ]`.
- La palabra clave `new` seguida por `type` y por una expresión entera entre corchetes cuadrados, que identifica el número de filas. `new` asigna memoria para las filas del array uni-dimensional de filas y pone a cero todos los bytes de cada elemento, lo que significa que cada elemento contiene una referencia nula. Debe crear un array uni-dimensional de columnas separado y asignarle su referencia cada elemento fila.
- = se utiliza para asignar la referencia del array bi-dimensional a *`variable_name`*.

El siguiente fragmento de código usa sólo la palabra clave `new` para crear un array bi-dimensional que almacena datos basados en un tipo primitivo:

```
double [][] temperatures = new double [2][]; // Allocate two rows.

temperatures [0] = new double [3]; // Allocate three columns for row 0
temperatures [1] = new double [3]; // Alllocate three columns for row 1
```

```
temperatures [0][0] = 20.5; // Populate row 0
temperatures [0][1] = 30.6;
temperatures [0][2] = 28.3;

temperatures [1][0] = -38.7; // Populate row 1
temperatures [1][1] = -18.3;
temperatures [1][2] = -16.2;
```

Utilizar la palabra clave "new" y un inicializador

Esta técnica requiere una de estas sintaxis:

```
type variable_name '[' ']' '[' ']' '='
    'new' type '[' ']' '[' ']' '{' [ rowInitializer [ ',' ... ] ] '}' ';'

type '[' ']' '[' ']' variable_name '='
    'new' type '[' ']' '[' ']' '{' [ rowInitializer [ ',' ... ] ] '}' ';'
```

donde *`rowInitializer`* tiene la siguiente sintaxis:

```
'{' [ initial_value [ ',' ... ] ] '}'
```

En ambas sintaxis:

- *`variable_name`* especifica el nombre de la variable del array bi-dimensional.
- *`type`* especifica el tipo de cada elemento. Como es una variable de array bi-dimensional contiene una referencia a un array bi-dimensional, su tipo es `type [ ] [ ]`.
- La palabra clave `new` seguida por `type` y por dos parejas de corchetes cuadrados vacíos, y cero o más inicializadores de filas entre un par de corchetes cuadrados. Si no se especifica ningún inicializador de fila, el array bi-dimensional está vacío. Cada inicializador de fila especifica cero o más valores iniciales para las columnas de esa fila.
- = se utiliza para asignar la referencia del array bi-dimensional a *`variable_name`*.

El siguiente fragmento de código usa la palabra clave `new` y un inicializador para crear un array bi-dimensional que almacena datos basados en un tipo primitivo:

```
double [][] temperatures = new double [][] { { 20.5, 30.6, 28.3 },
                                             { -38.7, -18.3, -16.2 } };
```

El fragmento de código de arriba se comporta igual que el anterior `double [][] temperatures = new double [][] { { 20.5, 30.6, 28.3 }, { -38.7, -18.3, -16.2 } };`.

trabajar con Arrays bi-dimensionales

Después de crear un array bi-dimensional, querrá almacenar y recuperar datos de sus elementos. Puede realizar estas tareas con la siguiente sintaxis:

```
variable_name '[' integer_expression1 ']' '[' integer_expression2 ']'
```

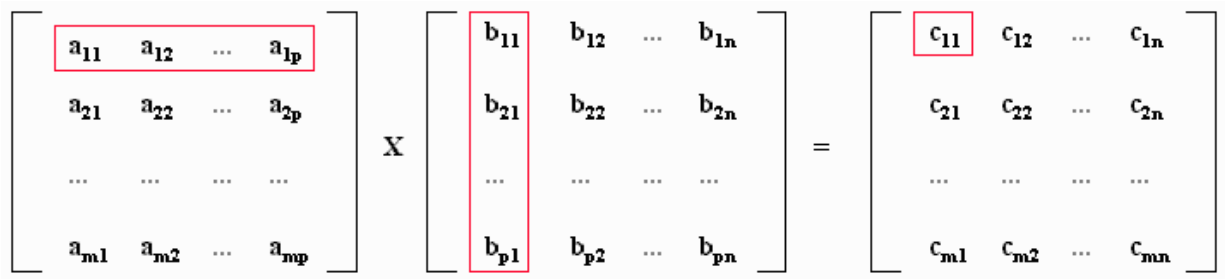
`integer_expression1` identifica el índice de fila del elemento y va de cero hasta la longitud del array menos uno. Igual ocurre con `integer_expression2` pero para el índice de columna. como todas las filas tienen la misma longitud, podría encontrar conveniente especificar *`variable_name`* `[0].length` para especificar el número de columnas de cualquier fila. El siguiente fragmento de código almacena y recupera datos de un elemento de un array bi-dimensional:

```
double [][] temperatures = { { 20.5, 30.6, 28.3 },
                             { -38.7, -18.3, -16.2 } };

temperatures [0][1] = 18.3;               // Replace 30.6 with 18.3
System.out.println (temperatures [1][2]);  // Output: -16.2
```

Algoritmo de Multiplicación de Matrices

Multiplicar una matriz por otra es una operación común en el trabajo con gráficos, con datos económicos, o con datos industriales. Los desarrolladores normalmente utilizan el algoritmo de multiplicación de matrices para completar esa multiplicación. ¿Cómo funciona ese algoritmo? Dejemos que 'A' represente una matriz con 'm' filas y 'n' columnas. De forma similar, 'B' representa un matriz con 'p' filas y 'n' columnas. Multiplicar A por B produce una matriz C obtenida de multiplicar todas las entradas de A por su correspondencia en B. La siguiente figura ilustra estas operaciones.



$$c_{11} = a_{11}b_{11} + a_{12}b_{21} + \ldots + a_{1p}b_{p1}$$

Cuidado:

La multiplicación de matrices requiere que el número de columnas de la matriz de la izquierda (A) sea igual al de la matriz de la derecha (B). Por ejemplo, para multiplicar una matriz A de cuatro columnas por fila por una matriz B (como en A x B), B debe contener exactamente cinco filas.

El siguiente pseudocódigo demuestra el algoritmo de multiplicación de matrices:

```
//  __________    _____   ____________________________
// | 10   30 |   | 5 |   | 10 x 5 + 30 x 7 (260) |
// |         | X |   | = |                       |
// | 20   40 |   | 7 |   | 20 x 5 + 40 * 7 (380) |
//  __________    _____   ____________________________

DECLARE INTEGER a [][] = [ 10, 30 ] [ 20, 40 ]
DECLARE INTEGER b [][] =  [ 5, 7 ]

DECLARE INTEGER m = 2 // Number of rows in left matrix (a)
DECLARE INTEGER p = 2 // Number of columns in left matrix (a)
                      // Number of rows in right matrix (b)
DECLARE INTEGER n = 1 // Number of columns in right matrix (b)
                      // Note: cs1 must equal rs2
DECLARE INTEGER c [m][n] // c holds 2 rows by 1 columns

// All elements initialize to 0
FOR i = 0 TO m - 1
    FOR j = 0 TO n - 1
        FOR k = 0 TO p - 1
            c [i][j] = c [i][j] + a [i][k] * b [k][j]
        NEXT k
    NEXT j
NEXT i
```

El pseudocódigo de arriba requiere tres bucles `FOR` para realizar la multiplicación. El bucle más interno multiplica una sola fila de la matriz a por una sola columna de la matriz B y añade el resultado a una sola entrada de la matriz C. El siguiente listado presenta el equivalente Java del pseudocódigo anterior:

```
// MatMultDemo.java

class MatMultDemo {
```

```
   public static void main (String [] args) {
      int [][] a =       int [][] b =
      dump (a);
      System.out.println ();
      dump (b);
      System.out.println ();
      int [][] c = multiply (a, b);
      dump (c);
   }

   static void dump (int [][] x) {
      if (x == null) {
          System.err.println ("array is null");
          return;
      }
      // Dump the matrix's element values to the standard output device
      // in a tabular order
      for (int i = 0; i < x.length; i++) {
           for (int j = 0; j < x [0].length; j++)
                System.out.print(x [i][j] + " ");
                System.out.println ();
      }
   }

   static int [][] multiply (int [][] a, int [][] b) {
      // ============================================================
      // 1. a.length contains a's row count
      //
      // 2. a [0].length (or any other a [x].length for a valid x)
      //    contains a's column count
      //
      // 3. b.length contains b's row count
      //
      // 4. b [0].length (or any other b [x].length for a valid x)
      //    contains b's column count
      // ============================================================

       // If a's column count != b's row count, bail out
      if (a [0].length != b.length) {
          System.err.println ("a's column count != b's row count");
          return null;
      }

      // Allocate result matrix with a size equal to a's row count x b's
      // column count
      int [][] result = new int [a.length][];
      for (int i = 0; i < result.length; i++)
           result [i] = new int [b [0].length];

      // Perform the multiplication and addition
      for (int i = 0; i < a.length; i++)
           for (int j = 0; j < b [0].length; j++)
                for(int k = 0; k < a [0].length; k++) // Or k < b.length
                    result [i][j] += a [i][k] * b [k][j];

      // Return the result matrix
      return result;
   }
}
```

`MatMultDemo` produce esta salida:

```
10 30
20 40
5
7
260
380
```

Exploremos un problema donde se necesita la multiplicación de matrices para obtener una solución. Un frutero de Florida carga una pareja de semitrailers con 1250 cajas de naranjas, 400 cajas de melocotones y 250 cajas de uvas. En la siguiente figura aparece la tabla de precios de mercado, por cada tipo de fruta en cinco ciudades diferentes.

	Oranges	Peaches	Grapefruit
New York	$10	$8	$12
Los Angeles	$11	$8.50	$11.55
Miami	$8.75	$6.90	$10
Chicago	$10.50	$8.25	$11.75

¿A qué ciudades debería enviar los semitrailers para obtener el máximo ingreso? Para resolver este problema, primero consideremos la tabla de la imagen anterior como una matriz de precios de cuatro filas por tres columnas. Luego construimos una matriz de tres filas por una columna con las cantidades, que aparece abajo:

```
==    ==
| 1250 |
|      |
|  400 |
|      |
|  250 |
==    ==
```

Ahora que tenemos las dos matrices, simplemente multiplicamos la matriz de precios por la matriz de cantidades para producir una matriz de ingresos:

```
==                  ==             ==        ==
| 10.00  8.00  12.00 |  ==    ==    | 18700.00 | New York
|                    |  | 1250 |    |          |
| 11.00  8.50  11.55 |  |      |    | 20037.50 | Los Ángeles
|                    |X |  400 | =  |          |
|  8.75  6.90  10.00 |  |      |    | 16197.50 | Miami
|                    |  |  250 |    |          |
| 10.50  8.25  11.75 |  ==    ==    | 19362.50 | Chicago
==                  ==             ==        ==
```

Enviar los dos semitrailers a Los Ángeles produce el mayor ingreso. Pero cuando se consideren la distancia y el consumo de gasoil, quizás New York sea la mejor apuesta

Arrays Desiguales

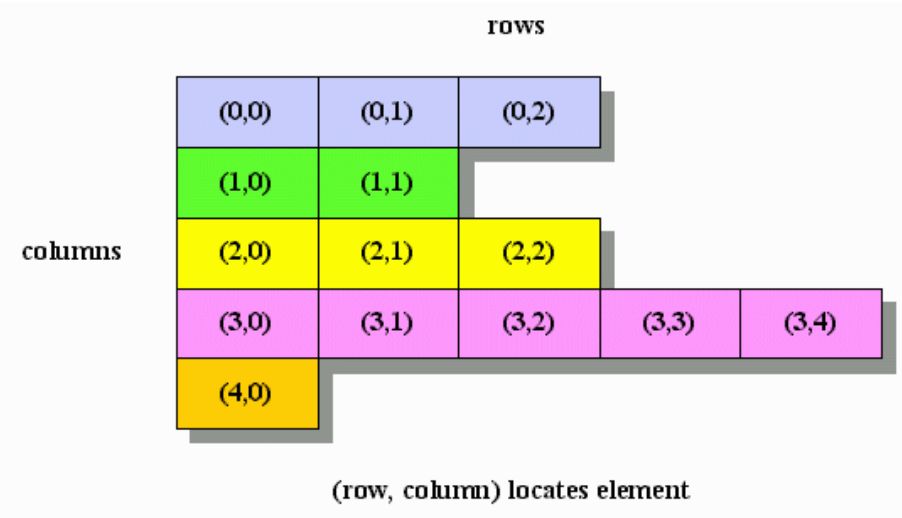
Suponga que su código fuente contiene la siguiente declaración de matriz: `int [][] x = new int [5][];`. Esto declara una matriz de enteros que contiene cinco filas, y `x.length` devuelve ese número de filas. Normalmente, completa la creación de la matriz especificando el mismo número de columnas para cada fila. Por ejemplo, especificando 10 columnas para cada fila utilizando el siguiente código:

```
for (int i = 0; i < x.length; i++)
     x [i] = new int [10];
```

Al contrario que otros lenguajes, Java no le fuerza a especificar el mismo número de columnas por cada fila. Utilizando el fragmento de código anterior, asigne tres columnas a la fila cero, dos a la fila 1, tres a la fila 2, cinco a la fila 3 y una a la fila 4, lo que demuestra el siguiente fragmento de código:

```
x [0] = new int [3];
x [1] = new int [2];
x [2] = new int [3];
x [3] = new int [5];
x [4] = new int [1];
```

Después de ejecutar este código, usted tendrá una matriz degenerada conocida como un *array desigual*. La siguiente imagen ilustra este tipo de arrays:



Los arrays desiguales son estructuras de datos útiles debido a su capacidad de ahorro de memoria. Por ejemplo, considere una hoja de cálculo con el potencial de 100.000 filas por 20.000 columnas. Si intentamos utilizar una matriz que contenga toda la hoja de cálculo, requeriremos una enorme cantidad de memoria. Pero supongamos que la mayoría de las celdas contienen valores por defecto, como un 0 para valores numéricos y `null` para celdas no numéricas. Si utilizamos un array desigual en lugar de una matriz, almacenaremos sólo las celdas que contienen datos numéricos. (Por supuesto, necesitamos algún tipo de mecanismo de mapeo que busque las coordenadas de la hoja de cálculo [filas, columnas] a las coordenadas del array desigual [filas], [columnas]).

Los Arrays Java son Objetos

La primera sentencia del capítulo 10 de la *Especificación del Lenguaje Java* dice lo siguiente: **En el lenguaje Java, los arrays son objetos.** Detrás de la escena, cada array es un ejemplar de una clase oculta que hereda 11 métodos de la clase `Object` y sobrescribe el método `protected Object clone() throws CloneNotSupportedException` de la misma clase para que un array pueda ser clonado en la sombra. Además, esta clase oculta proporciona un campo `length`. El siguiente listado demuestra la asociación entre arrays y objetos:

```
// ArrayIsObject.java

class ArrayIsObject {

   public static void main (String [] args) {

      double [] a = { 100.5, 200.5, 300.5 };
      double [] b = { 100.5, 200.5, 300.5 };
      double [] c = b;

      System.out.println ("a's class is " + a.getClass ());
      System.out.println ("a and b are " + ((a.equals (b)) ? "" : "not ") +
"equal");
      System.out.println ("b and c are " + ((b.equals (c)) ? "" : "not ") +
"equal");
```

```
      double [] d = (double []) c.clone ();
      System.out.println ("c and d are " + ((c.equals (d)) ? "" : "not ") +
"equal");

      for (int i = 0; i < d.length; i++)
           System.out.println (d [i]);
   }
}
```

Cuando se ejecuta `ArrayIsObject` produce la siguiente salida:

```
a's class is class [D
a and b are not equal
b and c are equal
c and d are not equal
100.5
200.5
300.5
```

`ArrayIsObject` crea las referencias a los arrays `a`- y `b` con la misma precisión y los mismos contenidos y la misma longitud. Para el array `a`, `a.getClass ()` devuelve `class [D`, donde `[D` es el nombre de la clase oculta. A pesar de que ambos arrays tienen los mismos contenidos, `a.equals (b)` devuelve `false` porque `equals()` compara referencias (no contenidos), y `a` y `b` contienen diferentes referencias. La referencia de `b` se asigna a `c`, y `b.equals (c)` devuelve `true` porque `b` y `c` referencian al mismo array. `c.clone()` crea un clon de `c`, y una referencia de ese array se asigna a `d`. Para probar que la referencia `d` contiene los mismos contenidos que la referencia del array `c`, el bucle `for` itera sobre todos los elementos e imprime su contenido. El bucle lee los contenidos y el campo de sólo lectura `length` para determinar sobre cuantos elementos iterar.

Truco:

En el código fuente, especifique siempre `.length` (como `d.length`) en vez de la longitud real del array. De esta forma, eliminará el riesgo de introducir bugs relacionados con la longitud, si después decide modificar la longitud del array en su código de creación.