UNIVERSIDAD CATOLICA DE COLOMBIA
FACULTAD DE INGENIERIA DE SISTEMAS

CURSO: JAVA BASICO
PROFESOR: EMERSON CASTAÑEDA SANABRIA

TEMA: Estructura del lenguaje.

OBJETIVOS:

- Presentar los elementos generales de Java como lenguaje de programación.

CONTENIDO:

1. Identificadores.
2. Tipos de Datos.
3. Operadores.
4. Estructuras de programación – Control de Flujo.
5. Arreglos.

DESARROLLO:

1. Identificadores

Una variable es un nombre que contiene un valor que puede cambiar a lo largo del programa. De acuerdo con el tipo de información que contienen, en Java hay dos tipos principales de variables:

Variables de tipos primitivos:
Están definidas mediante un valor único que puede ser entero, de punto flotante, carácter o booleano. Java permite distinta precisión y distintos rangos de valores para estos tipos de variables (char, byte, short, int, long, float, double, boolean). Ejemplos de variables de tipos primitivos podrían ser: 123, 3456754, 3.1415, 12e-09, 'A', true, etc.

Variables referencia:
Las variables referencia son referencias o nombres de una información más compleja: arreglos u objetos de una determinada clase.

Desde el punto de vista del papel o misión en el programa, las variables pueden ser:

Variables miembro de una clase (Atributos):
Se definen en una clase, fuera de cualquier método; pueden ser tipos primitivos o referencias.

Variables locales:
Se definen dentro de un método o más en general dentro de cualquier bloque entre llaves {}. Se crean en el interior del bloque y se destruyen al finalizar dicho bloque. Pueden ser también tipos primitivos o referencias.

Nombres de Variables

Los nombres de variables en Java se pueden crear con mucha libertad. Pueden ser cualquier conjunto de caracteres numéricos y alfanuméricos, sin algunos caracteres especiales utilizados por Java como operadores o separadores (,.+-*/ etc.).

Existe una serie de palabras reservadas las cuales tienen un significado especial para Java y por lo tanto no se pueden utilizar como nombres de variables. Dichas palabras son:

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

(*) son palabras reservadas, pero no se utilizan en la actual implementación del lenguaje Java.

2. Tipos de Datos

Tipos Primitivos de Variables

Se llaman **tipos primitivos** de variables de **Java** a aquellas variables sencillas que contienen los tipos de información más habituales: valores boolean, caracteres y valores numéricos enteros o de punto flotante.

Java dispone de ocho tipos primitivos de variables: un tipo para almacenar valores **true** y **false** (**boolean**); un tipo para almacenar caracteres (**char**), y 6 tipos para guardar valores numéricos, cuatro tipos para enteros (**byte**, **short**, **int** y **long**) y dos para valores reales de punto flotante (**float** y **double**). Los rangos y la memoria que ocupa cada uno de estos tipos se muestran a continuacion:

Tipo	Tamaño	Rango
Boolean	1 byte	Valores true y false
char	2 bytes.	Unicode. Comprende el código ASCII
byte	1 byte.	Valor entero entre -128 y 127
short	2 bytes.	Valor entero entre -32768 y 32767
int	4 bytes.	Valor entero entre -2.147.483.648 y 2.147.483.647
long	8 bytes.	Valor entre -9.223.372.036.854.775.808 y 9.223.372.036.854.775.807
float	4 bytes	(entre 6 y 7 cifras decimales equivalentes). De -3.402823E38 a -1.401298E-45 y de 1.401298E-45 a 3.402823E38
double	8 bytes	(unas 15 cifras decimales equivalentes). De -1.79769313486232E308 a -4.94065645841247E-324 y de 4.94065645841247E-324 a 1.79769313486232E308

Los tipos primitivos de **Java** tienen algunas características importantes que se resumen a continuación:

El tipo **boolean** no es un valor numérico: sólo admite los valores **true** o **false**. El tipo **boolean** no se identifica con el igual o distinto de cero, como en C/C++. El resultado de la expresión lógica que aparece como condición en un bucle o en una bifurcación debe ser **boolean**.

El tipo **char** contiene caracteres en código UNICODE (que incluye el código ASCII), y ocupan 16 bits por carácter. Comprende los caracteres de prácticamente todos los idiomas.

Los tipos **byte**, **short**, **int** y **long** son números enteros que pueden ser positivos o negativos, con distintos valores máximos y mínimos. A diferencia de C/C++, en **Java** no hay enteros **unsigned**.

Los tipos **float** y **double** son valores de punto flotante (números reales) con 6-7 y 15 cifras decimales equivalentes, respectivamente. Se utiliza la palabra **void** para indicar la ausencia de un tipo de variable determinado.

A diferencia de C/C++, los tipos de variables en **Java** están perfectamente definidos en todas y cada una de las posibles plataformas. Por ejemplo, un **int** ocupa siempre la misma memoria y tiene el mismo rango de valores, en cualquier tipo de ordenador.

Cómo se definen e inicializan las variables

Una variable se define especificando el **tipo** y el **nombre** de dicha variable. Estas variables pueden ser tanto de tipos **primitivos** como **referencias** a objetos de alguna clase perteneciente al **API** de **Java** o generada por el usuario. Si no se especifica un valor en su declaración, las variable **primitivas** se inicializan a cero (salvo boolean y char, que se inicializan a **false** y '\0'). Análogamente las variables de tipo **referencia** son inicializadas por defecto a un valor especial: **null**.

Es importante distinguir entre la referencia a un objeto y el objeto mismo. Una referencia es una variable que indica dónde está guardado un objeto en la memoria del ordenador (a diferencia de C/C++, Java no permite acceder al valor de la dirección, pues en este lenguaje se han eliminado los punteros). Al declarar una referencia todavía no se encuentra "apuntando" a ningún objeto en particular (salvo que se cree explícitamente un nuevo objeto en la declaración), y por eso se le asigna el valor null. Si se desea que esta referencia apunte a un nuevo objeto es necesario crear el objeto utilizando el operador new. Este operador reserva en la memoria del ordenador espacio para ese objeto (variables y funciones). También es posible igualar la referencia declarada a otra referencia a un objeto existente previamente.

3. Operadores

Java es un lenguaje rico en operadores, que son casi idénticos a los de C/C++. Estos operadores se describen brevemente en los apartados siguientes.

Operadores aritméticos
Son operadores binarios (requieren siempre dos operandos) que realizan las operaciones aritméticas habituales: **suma** (+), **resta** (-), **multiplicación** (*), **división** (/) y **resto de la división** (%).

Operadores de asignación
Los operadores de asignación permiten asignar un valor a una variable. El operador de asignación por excelencia es el **operador igual** (=). La forma general de las sentencias de asignación con este operador es: variable = expresión; **Java** dispone de otros operadores de asignación. Se trata de versiones abreviadas del operador (=) que realizan operaciones "acumulativas" sobre una variable. La Tabla muestra estos operadores y su equivalencia con el uso del **operador igual** (=).

Operador	Utilización	Expresión equivalente
+=	op1 += op2	op1 = op1 + op2
-=	op1 -= op2	op1 = op1 - op2
*=	op1 *= op2	op1 = op1 * op2
/=	op1 /= op2	op1 = op1 / op2
%=	op1 %= op2	op1 = op1 % op2

Operadores unarios

Los operadores **más** (+) y **menos** (-) unarios sirven para mantener o cambiar el signo de una variable, constante o expresión numérica. Su uso en **Java** es el estándar de estos operadores.

Operador instanceof
El operador **instanceof** permite saber si un objeto pertenece o no a una determinada clase. Es un operador binario cuya forma general es, objectName instanceof ClassName y que devuelve **true** o **false** según el objeto pertenezca o no a la clase.

Operador condicional ?:
Este operador, tomado de C/C++, permite realizar bifurcaciones condicionales sencillas. Su forma general es la siguiente:

booleanExpression ? res1 : res2;

donde se evalúa **booleanExpression** y se devuelve **res1** si el resultado es **true** y **res2** si el resultado es **false**. Es el único operador ternario (tres argumentos) de **Java**. Como todo operador que devuelve un valor puede ser utilizado en una expresión. Por ejemplo las sentencias:

```
x = 1;
y = 10;
z = (x<y)?x+3:y+8;
```

asignarían a **z** el valor 4, es decir x+3.

Operadores incrementales
Java dispone del operador **incremento** (++) y **decremento** (--). El operador (++) incrementa en una unidad la variable a la que se aplica, mientras que (--) la reduce en una unidad. Estos operadores se pueden utilizar de dos formas:

1. Precediendo a la variable (por ejemplo: **++i**). En este caso primero se incrementa la variable y luego se utiliza (ya incrementada) en la expresión en la que aparece.
2. Siguiendo a la variable (por ejemplo: **i++**). En este caso primero se utiliza la variable en la expresión (con el valor anterior) y luego se incrementa.

En muchas ocasiones estos operadores se utilizan para incrementar una variable fuera de una expresión. En este caso ambos operadores son equivalente. Si se utilizan en una expresión más complicada, el resultado de utilizar estos operadores en una u otra de sus formas será diferente. La actualización de contadores en bucles **for** es una de las aplicaciones más frecuentes de estos operadores.

Operadores relacionales
Los **operadores relacionales** sirven para realizar comparaciones de igualdad, desigualdad y relación de menor o mayor. El resultado de estos operadores es siempre un valor **boolean** (**true** o **false**) según se cumpla o no la relación considerada. La tabla muestra los operadores relacionales de **Java**.

Operador	Utilización	El resultado es true
>	op1 > op2	si op1 es mayor que op2
>=	op1 >= op2	si op1 es mayor o igual que op2
<	op1 < op2	si op1 es menor que op2
<=	op1 <= op2	si op1 es menor o igual que op2
==	op1 == op2	si op1 y op2 son iguales
!=	op1 != op2	si op1 y op2 son diferentes

Estos operadores se utilizan con mucha frecuencia en las **bifurcaciones** y en los **bucles.**

Operadores lógicos
Los operadores lógicos se utilizan para construir **expresiones lógicas**, combinando valores lógicos (**true** y/o **false**) o los resultados de los operadores **relacionales**. La Tabla muestra los operadores lógicos de **Java**. Debe notarse que en ciertos casos el segundo operando no se evalúa porque ya no es necesario (si ambos tienen que ser **true** y el primero es **false**, ya se sabe que la condición de que ambos sean **true** no se va a cumplir). Esto puede traer resultados no deseados y por eso se han añadido los operadores (&) y (|) que garantizan que los dos operandos se evalúan siempre.

Operador	Nombre	Utilización	Resultado
&&	AND	op1 && op2	true si op1 y op2 son true. Si op1 es false ya no se evalúa op2
||	OR	op1 || op2	true si op1 u op2 son true. Si op1 es true ya no se evalúa op2
!	negación	! op	true si op es false y false si op es true
&	AND	op1 & op2	true si op1 y op2 son true. Siempre se evalúa op2
|	OR	op1 | op2	true si op1 u op2 son true. Siempre se evalúa op2

Operador de concatenación de cadenas de caracteres (+)
El operador más (+) se utiliza también para concatenar cadenas de caracteres. Por ejemplo, para escribir una cantidad con un rótulo y unas unidades puede utilizarse la sentencia:

```
System.out.println("El total asciende a " + result + " unidades");
```

donde el operador de concatenación se utiliza dos veces para construir la cadena de caracteres que se desea imprimir por medio del método **println()**. La variable numérica **result** es convertida automáticamente por **Java** en cadena de caracteres para poderla concatenar. En otras ocasiones se deberá llamar explícitamente a un método para que realice esta conversión.

Operadores que actúan a nivel de bits
Java dispone también de un conjunto de operadores que actúan a nivel de bits. Las operaciones de bits se utilizan con frecuencia para definir señales o **flags**, esto es, variables de tipo entero en las que cada uno de sus bits indican si una opción está activada o no. La Tabla muestra los operadores de **Java** que actúan a nivel de bits.

Operador	Utilización	Resultado
>>	op1 >> op2	Desplaza los bits de op1 a la derecha una distancia op2
<<	op1 << op2	Desplaza los bits de op1 a la izquierda una distancia op2
>>>	op1 >>> op2	Desplaza los bits de op1 a la derecha una distancia op2 (positiva)
&	op1 & op2	Operador AND a nivel de bits
|	op1 | op2	Operador OR a nivel de bits
^	op1 ^ op2	Operador XOR a nivel de bits (1 si sólo uno de los operandos es 1)
~	~op2	Operador complemento (invierte el valor de cada bit)

En binario, las potencias de dos se representan con un único bit activado. Por ejemplo, los números (1, 2, 4, 8, 16, 32, 64, 128) se representan respectivamente de modo binario en la forma (00000001, 00000010, 00000100, 00001000, 00010000, 00100000, 01000000, 10000000), utilizando sólo 8 bits. La suma de estos números permite construir una variable **flags** con los bits activados que se deseen. Por ejemplo, para construir una variable **flags** que sea 00010010

bastaría hacer **flags**=2+16. Para saber si el segundo bit por la derecha está o no activado bastaría utilizar la sentencia,

```
if (flags & 2 == 2) {...}
```

La Tabla muestra los operadores de asignación a nivel de bits.

Operador	Utilización	Equivalente a
&=	op1 &= op2	op1 = op1 & op2
|=	op1 |= op2	op1 = op1 | op2
^=	op1 ^= op2	op1 = op1 ^ op2
<<=	op1 <<= op2	op1 = op1 << op2
>>=	op1 >>= op2	op1 = op1 >> op2
>>>=	op1 >>>= op2	op1 = op1 >>> op2

Precedencia de operadores
El orden en que se realizan las operaciones es fundamental para determinar el resultado de una expresión. Por ejemplo, el resultado de x/y*z depende de qué operación (la división o el producto) se realice primero. La siguiente lista muestra el orden en que se ejecutan los distintos operadores en un sentencia, de **mayor a menor** precedencia:

Parentesis, incremeto y decremento	[] . (params) expr++ expr--
Operadores unarios	++expr --expr +expr -expr ~ !
De creación o cast	new (type)expr
Multiplicativos	* / %
Aditivos	+ -
De desplazamiento	<< >> >>>
Relacionales	< > <= >= instanceof
Comparación	== !=
AND a nivel de bits	&
OR exclusivo a nivel de bits	^
OR inclusivo a nivel de bits	|
AND Lógico	&&
OR Lógico	||
Condicional	? :
Asignación	= += -= *= /= %= &= ^= |= <<= >>= >>>=

En **Java**, todos los operadores binarios, excepto los operadores de asignación, se evalúan de **izquierda a derecha**. Los operadores de asignación se evalúan de derecha a izquierda, lo que significa que el valor de la derecha se copia sobre la variable de la izquierda.

4. Estructuras de programación – Control de Flujo.

Las **estructuras de programación** o **estructuras de control** permiten tomar decisiones y realizar un proceso repetidas veces. Son los denominados **bifurcaciones** y **bucles**. En la mayoría de los lenguajes de programación, este tipo de estructuras son comunes en cuanto a concepto, aunque su sintaxis varía de un lenguaje a otro. La sintaxis de **Java** coincide prácticamente con la utilizada en C/C++, lo que hace que para un programador de C/C++ no suponga ninguna dificultad adicional.

Sentencias o expresiones
Una **expresión** es un conjunto variables unidos por **operadores**. Son órdenes que se le dan al computador para que realice una tarea determinada. Una **sentencia** es una **expresión** que acaba

en **punto y coma** (;). Se permite incluir varias sentencias en una línea, aunque lo habitual es utilizar una línea para cada sentencia. Por ejemplo:

```
i = 0; j = 5; x = i + j;// Línea compuesta de tres sentencias
```

Comentarios
Existen dos formas diferentes de introducir comentarios entre el código de **Java** (en realidad son tres, como pronto se verá). Son similares a la forma de realizar comentarios en el lenguaje C++. Los comentarios son tremendamente útiles para poder entender el código utilizado, facilitando de ese modo futuras revisiones y correcciones. Además permite que cualquier persona distinta al programador original pueda comprender el código escrito de una forma más rápida. Se recomienda acostumbrarse a comentar el código desarrollado. De esta forma se simplifica también la tarea de estudio y revisión posteriores.

Java interpreta que todo lo que aparece a la derecha de dos barras "***//***" en una línea cualquiera del código es un comentario del programador y no lo tiene en cuenta. El comentario puede empezar al comienzo de la línea o a continuación de una instrucción que debe ser ejecutada. La segunda forma de incluir comentarios consiste en escribir el texto entre los símbolos ***/****…****/***. Este segundo método es válido para comentar más de una línea de código. Por ejemplo:

```
// Esta línea es un comentario

int a=1; // Comentario a la derecha de una sentencia

// Esta es la forma de comentar más de una línea utilizando
// las dos barras. Requiere incluir dos barras al comienzo de cada línea

/* Esta segunda forma es mucho más cómoda para comentar un número elevado
de líneas ya que sólo requiere modificar el comienzo y el final. */
```

En **Java** existe además una forma especial de introducir los comentarios (utilizando /**…*/ más algunos caracteres especiales) que permite generar automáticamente la documentación sobre las **clases** y **packages** desarrollados por el programador. Una vez introducidos los comentarios, el programa **javadoc.exe** (incluido en el **JDK**) genera de forma automática la información de forma similar a la presentada en la propia documentación del **JDK**. La sintaxis de estos comentarios y la forma de utilizar el programa **javadoc.exe** se puede encontrar en la información que viene con el **JDK**.

Bifurcaciones
Las bifurcaciones permiten ejecutar una de entre varias acciones en función del valor de una expresión lógica o relacional. Se tratan de estructuras muy importantes ya que son las encargadas de controlar el flujo de ejecución de un programa. Existen dos bifurcaciones diferentes: **if** y **switch**.

Bifurcación if
Esta estructura permite ejecutar un conjunto de sentencias en función del valor que tenga la expresión de comparación (se ejecuta si la expresión de comparación tiene valor **true**). Tiene la forma siguiente:

```
if (booleanExpression) {
      statements;
}
```

Las **llaves** {} sirven para agrupar en un **bloque** las sentencias que se han de ejecutar, y no son necesarias si sólo hay una sentencia dentro del **if**.

Bifurcación if else

Análoga a la anterior, de la cual es una ampliación. Las sentencias incluidas en el **else** se ejecutan en el caso de no cumplirse la expresión de comparación (**false**),

```
if (booleanExpression) {
      statements1;
} else {
      statements2;
}
```

Bifurcación if elseif else
Permite introducir más de una expresión de comparación. Si la primera condición no se cumple, se compara la segunda y así sucesivamente. En el caso de que no se cumpla ninguna de las comparaciones se ejecutan las sentencias correspondientes al **else**.

```
if (booleanExpression1) {
      statements1;
} else if (booleanExpression2) {
      statements2;
} else if (booleanExpression3) {
      statements3;
} else {
      statements4;
}
```

Véase a continuación el siguiente ejemplo:

```
int numero = 61; // La variable "numero" tiene dos dígitos
if(Math.abs(numero) < 10) // Math.abs() calcula el valor absoluto.
(false)
      System.out.println("Numero tiene 1 digito ");
else if (Math.abs(numero) < 100) // Si numero es 61, estamos en este caso
(true)
      System.out.println("Numero tiene 1 digito ");
else { // Resto de los casos
      System.out.println("Numero tiene mas de 3 digitos ");
      System.out.println("Se ha ejecutado la opcion por defecto ");
}
```

Sentencia switch
Se trata de una alternativa a la bifurcación **if elseif else** cuando se compara la **misma expresión** condistintos valores. Su forma general es la siguiente:

```
switch (expression) {
      case value1: statements1; break;
      case value2: statements2; break;
      case value3: statements3; break;
      case value4: statements4; break;
      case value5: statements5; break;
      case value6: statements6; break;
      [default: statements7;]
}
```

Las características más relevantes de **switch** son las siguientes:

1. Cada sentencia **case** se corresponde con un único valor de **expression**. No se pueden establecer rangos o condiciones sino que se debe comparar con valores concretos. El ejemplo del Apartado no se podría realizar utilizando **switch**.

2. Los valores no comprendidos en ninguna sentencia **case** se pueden gestionar en **default**, que es opcional.

3. En ausencia de **break**, cuando se ejecuta una sentencia **case** se ejecutan también todas las **case** que van a continuación, hasta que se llega a un **break** o hasta que se termina el **switch**. Ejemplo:

```
char c = (char)(Math.random()*26+'a'); // Generación aleatoria de letras
System.out.println("La letra " + c );
switch (c) {
      case 'a': // Se compara con la letra a
      case 'e': // Se compara con la letra e
      case 'i': // Se compara con la letra i
      case 'o': // Se compara con la letra o
      case 'u': // Se compara con la letra u
      System.out.println(" Es una vocal "); break;
      default:
      System.out.println(" Es una consonante ");
}
```

Bucles

Un **bucle** se utiliza para realizar un proceso repetidas veces. Se denomina también **lazo** o **loop**. El código incluido entre las **llaves** {} (opcionales si el proceso repetitivo consta de una sola línea), se ejecutará mientras se cumpla unas determinadas condiciones. Hay que prestar especial atención a los bucles infinitos, hecho que ocurre cuando la condición de finalizar el bucle (**booleanExpression**) no se llega a cumplir nunca. Se trata de un fallo muy típico, habitual sobre todo entre programadores poco experimentados.

Bucle while

Las sentencias **statements** se ejecutan mientras **booleanExpression** sea **true**.

```
while (booleanExpression) {
      statements;
}
```

Bucle for

La forma general del bucle **for** es la siguiente:

```
for (initialization; booleanExpression; increment) {
      statements;
}
```

que es equivalente a utilizar **while** en la siguiente forma,

```
inicializacion;
while (booleanExpression) {
      statements;
      increment;
}
```

La sentencia o sentencias **initialization** se ejecuta al comienzo del **for**, e **increment** después de **statements**. La **booleanExpression** se evalúa al comienzo de cada iteración; el bucle termina cuando la expresión de comparación toma el valor **false**. Cualquiera de las tres partes puede estar vacía. La **initialization** y el **increment** pueden tener varias expresiones separadas por comas. Por ejemplo, el código situado a la izquierda produce la salida que aparece a la derecha:

Código:

```
for(int i = 1, j = i + 10; i < 5; i++, j = 2*i) {
      System.out.println(" i = " + i + " j = " + j);
}
```

Salida:

```
i = 1 j = 11
i = 2 j = 4
i = 3 j = 6
i = 4 j = 8
```

Bucle do while

Es similar al bucle **while** pero con la particularidad de que el control está al final del bucle (lo que hace que el bucle se ejecute al menos una vez, independientemente de que la condición se cumpla o no). Una vez ejecutados los **statements**, se evalúa la condición: si resulta **true** se vuelven a ejecutar las sentencias incluidas en el bucle, mientras que si la condición se evalúa a **false** finaliza el bucle.

Este tipo de bucles se utiliza con frecuencia para controlar la satisfacción de una determinada condición de error o de convergencia.

```
do {
      statements
} while (booleanExpression);
```

Sentencias break y continue

La sentencia **break** es válida tanto para las bifurcaciones como para los bucles. Hace que se salga inmediatamente del bucle o bloque que se está ejecutando, sin sin realizar la ejecución del resto de las sentencias.

La sentencia **continue** se utiliza en los bucles (no en bifurcaciones). Finaliza la iteración “i” que en ese momento se está ejecutando (no ejecuta el resto de sentencias que hubiera hasta el final del bucle). Vuelve al comienzo del bucle y comienza la siguiente iteración (i+1).

Sentencias break y continue con etiquetas

Las **etiquetas** permiten indicar un lugar donde continuar la ejecución de un programa después de un **break** o **continue**. El único lugar donde se pueden incluir etiquetas es **justo delante de un bloque** de código entre llaves {} (if, switch, do...while, while, for) y sólo se deben utilizar cuando se tiene uno o más bucles (o bloques) dentro de otro bucle y se desea salir (break) o continuar con la siguiente iteración (continue) de un bucle que no es el actual.

Por tanto, la sentencia **break labelName** finaliza el bloque que se encuentre a continuación de **labelName**. Por ejemplo, en las sentencias,

```
bucleI: // etiqueta o label
for ( int i = 0, j = 0; i < 100; i++){
      while ( true ) {
            if( (++j) > 5) { break bucleI; } // Finaliza ambos bucles
            else { break; } // Finaliza el bucle interior (while)
      }
}
```

la expresión break bucleI; finaliza los dos bucles simultáneamente, mientras que la expresión break; sale del bucle **while** interior y seguiría con el bucle **for** en **i**. Con los valores presentados ambos bucles finalizarán con i = 5 y j = 6 (se invita al lector a comprobarlo). La sentencia **continue** (siempre dentro de al menos un bucle) permite transferir el control a un bucle con nombre o etiqueta. Por ejemplo, la sentencia, continue bucle1;

transfiere el control al bucle **for** que comienza después de la etiqueta **bucle1:** para que realice una nueva iteración, como por ejemplo:

```
bucle1:
for (int i=0; i<n; i++) {
      bucle2:
      for (int j=0; j<m; j++) {
            ...
            if (expression) continue bucle1; then continue bucle2;
            ...
      }
}
```

Sentencia return
Otra forma de salir de un bucle (y de un método) es utilizar la sentencia **return**. A diferencia de **continue** o **break**, la sentencia **return** sale también del método o función. En el caso de que la función devuelva alguna variable, este valor se deberá poner a continuación del return (returnvalue;).

Bloque try {...} catch {...} finally {...}
Java incorpora en el propio lenguaje la gestión de errores. El mejor momento para detectar los errores es durante la compilación. Sin embargo prácticamente sólo los errores de sintaxis son detectados en esta operación. El resto de problemas surgen durante la ejecución de los programas.

En el lenguaje **Java**, una **Exception** es un cierto tipo de error o una condición anormal que se ha producido durante la ejecución de un programa. Algunas **excepciones** son **fatales** y provocan que se deba finalizar la ejecución del programa. En este caso conviene terminar ordenadamente y dar un mensaje explicando el tipo de error que se ha producido. Otras **excepciones**, como por ejemplo no encontrar un fichero en el que hay que leer o escribir algo, pueden ser **recuperables**. En este caso el programa debe dar al usuario la oportunidad de corregir el error (definiendo por ejemplo un nuevo **path** del fichero no encontrado).

Los errores se representan mediante clases derivadas de la clase **Throwable**, pero los que tiene que chequear un programador derivan de **Exception** (**java.lang.Exception** que a su vez deriva de **Throwable**). Existen algunos tipos de excepciones que **Java** obliga a tener en cuenta. Esto se hace mediante el uso de bloques **try**, **catch** y **finally**.

El código dentro del bloque **try** está “vigilado”. Si se produce una situación anormal y se lanza como consecuencia una excepción, el control pasa al bloque **catch**, que se hace cargo de la situación y decide lo que hay que hacer. Se pueden incluir tantos bloques **catch** como se desee, cada uno de los cuales tratará un tipo de excepción. Finalmente, si está presente, se ejecuta el bloque **finally**, que es opcional, pero que en caso de existir se ejecuta siempre, sea cual sea el tipo de error.

5. Arreglos

Los **arreglos** de **Java** (vectores, matrices, hiper-matrices de más de dos dimensiones) se tratan como objetos de una clase predefinida. Los **arreglos** son **objetos**, pero con algunas características propias.

Los **arreglos** pueden ser asignados a objetos de la clase **Object** y los métodos de **Object** pueden ser utilizados con **arreglos**.

Algunas de sus características más importantes de los **arreglos** son las siguientes:

1. Los **arreglos** se crean con el operador **new** seguido del tipo y número de elementos.

2. Se puede acceder al número de elementos de un arreglo con la variable miembro implícita **length** (por ejemplo, **vect.length**).

3. Se accede a los elementos de un **arreglo** con los **corchetes []** y un **índice** que varía de 0 a **length-1**.

4. Se pueden crear **arreglos** de objetos de cualquier tipo. En principio un **arreglo** de objetos es un **arreglo de referencias** que hay que completar llamando al operador **new**.

5. Los elementos de un **arreglo** se inicializan al valor por defecto del tipo correspondiente (cero para valores numéricos, el carácter nulo para **char**, **false** para **boolean**, **null** para **Strings** y para referencias).

6. Como todos los objetos, los **arreglos** se pasan como argumentos a los métodos **por referencia**.

7. Se pueden crear **arreglos anónimos** (por ejemplo, crear un nuevo arreglo como argumento actual en la llamada a un método).

Inicialización de **arreglos**:

1. Los **arreglos** se pueden inicializar con valores entre llaves {...} separados por comas.
2. También los **arreglos de objetos** se pueden inicializar con varias llamadas a **new** dentro de unas llaves {...}.
3. Si se igualan dos referencias a un arreglo no se copia el arreglo, sino que se tiene un arreglo con dos nombres, apuntando al mismo y único objeto.
4. Creación de una **referencia** a un arreglo. Son posibles dos formas:

```
double[] x; // preferible
double x[];
```

5. Creación del **arreglo** con el operador **new**:

```
x = new double[100];
```

6. Las dos etapas 4 y 5 se pueden unir en una sola:

```
double[] x = new double[100];
```

A continuación se presentan algunos ejemplos de creación de arreglos:

```
// crear un arreglo de 10 enteros, que por defecto se inicializan a cero
int v[] = new int[10];

// crear arreglos inicializando con determinados valores
int v[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
String dias[] = {"lunes", "martes", "miercoles", "jueves",
"viernes", "sabado", "domingo"};

// arreglo de 5 objetos
MiClase listaObj[] = new MiClase[5]; // de momento hay 5 referencias a
null
for( i = 0 ; i < 5;i++)
```

```
      listaObj[i] = new MiClase(...);

// arreglo anónimo
obj.metodo(new String[]={"uno", "dos", "tres"});
```

Arreglos bidimensionales

Los arreglos bidimensionales de **Java** se crean de un modo muy similar al de C++ (con reserva dinámica de memoria). En **Java** una **matriz** es un **vector** de **vectores fila**, o más en concreto un vector de referencias a los vectores fila. Con este esquema, cada fila podría tener un número de elementos diferente. Una matriz se puede crear directamente en la forma,

```
int [][] mat = new int[3][4];
```

o bien se puede crear de modo dinámico dando los siguientes pasos:

1. Crear la **referencia** indicando con un doble corchete que es una **referencia a matriz**,

```
int[][] mat;
```

2. Crear el vector de referencias a las filas,

```
mat = new int[nfilas][];
```

3. Reservar memoria para los vectores correspondientes a las filas,

```
for (int i=0; i<nfilas; i++);
      mat[i] = new int[ncols];
```

A continuación se presentan algunos ejemplos de creación de arreglos bidimensionales:

```
// crear una matriz 3x3
// se inicializan a cero
double mat[][] = new double[3][3];
int [][] b = {{1, 2, 3},{4, 5, 6}, // esta coma es permitida
};
int c = new[3][]; // se crea el arreglo de referencias a arreglos
c[0] = new int[5];
c[1] = new int[4];
c[2] = new int[8];
```

En el caso de una matriz **b**, **b.length** es el número de filas y **b[0].length** es el número de columnas (de la fila 0). Por supuesto, los arreglos bidimensionales pueden contener tipos primitivos de cualquier tipo u objetos de cualquier clase.