Apéndice A: Características básicas del lenguaje de programación JAVA

A.1. Tipos de Datos Primitivos en Java

Todas las variables en el lenguaje Java deben tener un tipo de dato. El tipo de la variable determina los valores que la variable puede contener y las operaciones que se pueden realizar con ella.

Existen dos categorías de datos principales en el lenguaje Java: los tipos primitivos y los tipos referenciados.

Tipos Primitivos	Referencias a Objetos
int, short, byte, long	Strings
char, boolean	Arreglos
float, double	otros objetos

Los tipos primitivos contienen un sólo valor e incluyen los tipos como los enteros, coma flotante, los caracteres, etc... La tabla siguiente muestra todos los tipos primitivos soportados por el lenguaje Java, su formato, su tamaño y una breve descripción de cada uno:

Tipo	Tamaño/Formato	Descripción
(***Números enteros***)		
byte	8-bit complemento a 2	Entero de un Byte
short	16-bit complemento a 2	Entero corto
int	32-bit complemento a 2	Entero
long	64-bit complemento a 2	Entero largo
(***Números reales***)		
float	32-bit IEEE 754	Coma flotante de precisión simple
double	64-bit IEEE 754	Coma flotante de precisión doble
(***otros tipos***)		
char	16-bit Caracter	Un sólo carácter
boolean	true o false	Un valor booleano (verdadero o falso)

Los ocho tipos de datos primitivos manejan tipos comunes para ***enteros***, números de ***punto flotante***, ***caracteres*** y valores ***booléanos***. Se llaman primitivos, porque están integrados en el sistema y no son objetos en realidad, lo cual hace su uso más eficiente.

Observe que estos tipos de datos son independientes de la computadora, puede confiar que su tamaño y características son consistentes en los programas Java.

Enteros

Existen cuatro tipos de enteros Java, cada uno con un rango diferente de valores, todos tienen signo.

Tipo	***Tamaño***	***Rango***
byte	*8 bits*	*-128 a 127*
short	*16 bits*	*-32, 768 a 32, 767*
int	*32 bits*	*-2, 147, 483, 648 a 2, 147, 483, 647*
long	*64 bits*	*-9,223,372,036,854,775,808 a 9,223,372,036,854,775,807*

También puede forzar a un entero más pequeño a un long al agregarle una L o l a ese número.

Los enteros también pueden representarse en sistema octal o hexadecimal: un cero indica que un número es octal (0777). Un 0x o 0X inicial, significa que se expresa en hexadecimal (0xFF, oXAF45).

Punto flotante

Los tipos de punto flotante contienen más información que los tipos enteros. Las variables de punto flotante son números fraccionarios. Existen dos subtipos de punto flotante : float *y* double*.*

Puede forzar el número al tipo float al agregarle la letra f o F a ese número (2.56f).

Tipo	***Rango***
Double	*$-1.7x10^{-308}$ a $1.7x10^{308}$*
Float	*$-3.4x10^{-38}$ a $3.4x10^{38}$*

Booleanos

El tipo booleano tiene dos valores: True y False (verdadero y falso).

Caracter

El tipo carácter representa un carácter con base en el conjunto de caracteres de Unicode. Este tipo se define con la palabra clave char. El valor correspondiente a un tipo de carácter debe estar encerrado entre comillas sencillas (‘). Se representa con un código de 16 bits, permitiendo 65,536 caracteres diferentes: una magnitud mayor que la del conjunto de caracteres ASCII/ANSI.

Los valores pueden aparecer en forma normal como a,b,c o pueden representarse con una codificación hexadecimal. Estos valores hexadecimales comienzan con el prefijo \u, a fin de que Java sepa que se trata de un valor hexadecimal.

Por ejemplo, el retorno de carro en hexadecimal es \u000d. El tipo char, al igual que el booleano, no tiene un gemelo numérico. Sin embargo, el tipo char sí puede convertirse a enteros en caso necesario.

char coal;
coal ='c';

La siguiente tabla muestra los códigos especiales que pueden representar caracteres no imprimibles, así como los del conjunto de caracteres Unicode.

Escape	*Significado*
\n	*Línea nueva*
\t	*Tabulador*
\b	*Retroceso*
\r	*Regreso de carro*
\f	*Alimentación de forma*
**	*Diagonal inversa*
\'	*Comilla sencilla*
\"	*Comilla doble*
\ddd	*Octal*
\xdd	*Hexadecimal*
\udddd	*Carácter Unicode*

Cadenas y arreglos

Con excepción de los tipos entero, de punto flotante, booleano y carácter, la mayoría de los tipos restantes en Java son un objeto. En esta regla se incluyen las cadenas y los arreglos, los cuales pueden tener su propia clase.

Las cadenas son sólo una manera de representar una secuencia de caracteres. Ya que no son tipos integrados, tendrá que declararlas mediante la clase String. Así como a los tipos char se les da un valor entre comillas sencillas (‘), a las cadenas se les dan valores contenidos entre comillas dobles (“). Es posible unir (concatenar) varias cadenas por medio del signo más (+).

Las cadenas no son simples arreglos de caracteres como lo son en C o C++, aunque sí cuentan con las características parecidas a las de los arreglos. Puesto que los objetos de cadena en Java son reales, cuenta con métodos que le permiten combinar, verificar, y modificar cadenas con gran facilidad.

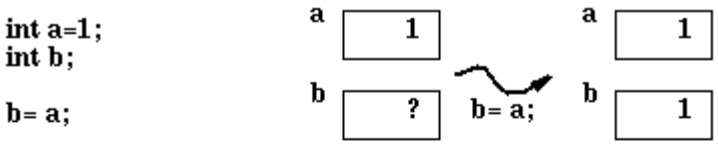
A.2. Tipos Referenciados en Java

Los ***tipos referenciados*** se llaman así porque el valor de una variable de referencia es una referencia (un puntero) hacia el valor real. En Java tenemos los arrays, las clases y los interfaces como tipos de datos referenciados.

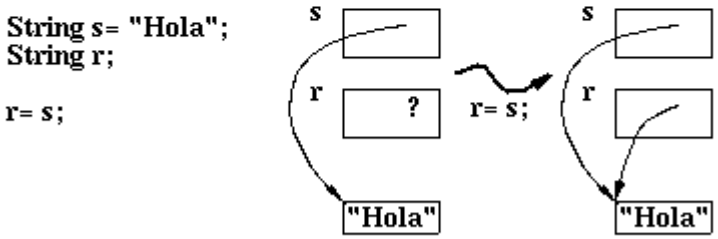
Las variables de tipo referencia a objetos almacenan direcciones y no valores directamente. Una referencia a un objeto es la dirección de un área en memoria destinada a representar ese objeto. El área de memoria se solicita con el operador *new*.

Al asignar una variable de tipo referencia a objeto a otra variable se asigna la dirección y no el objeto referenciado por esa dirección. Esto significa que ambas variables quedan referenciando el mismo objeto.

La diferencia entre ambas asignaciones se observa en la siguiente figura:



Asignacion de variables de tipo primitivo



Asignacion de variables de tipo compuesto

Esto tiene implicancias mayores ya que si se modifica el objeto referenciado por r, entonces también se modifica el objeto referenciado por s, puesto que son el mismo objeto. En Java una variable no puede almacenar directamente un objeto, como ocurre en C y C++.

Por lo tanto cuando se dice en Java que una variable es un string, lo que se quiere decir en realidad es que la variable es una referencia a un string.

A.3. Palabras Reservadas en Java

Las palabras reservadas Java no pueden usarse para nombres de variables, a continuación se muestra una tabla con las palabras reservadas que tienen un significado especial para el compilar:

abstract	*boolean*	*break*	*byte*	*case*	*cast*	*catch*
Char	*class*	*cons*	*continue*	*default*	*do*	*double*
Else	*extends*	*final*	*finally*	*float*	*for*	*future*
generic	*goto*	*if*	*implements*	*import*	*inner*	*instanceof*
In	*interface*	*long*	*native*	*new*	*null*	*operator*
Outer	*package*	*private*	*protected*	*public*	*rest*	*return*
Short	*static*	*super*	*switch*	*synchronized*	*this*	*throw*
Throws	*transient*	*try*	*var*	*unsigned*	*virtual*	*void*
Volatile	*while*					

A.4. OPERADORES ARITMETICOS

El lenguaje Java soporta varios operadores aritméticos, incluyendo + (suma), - (resta), * (multiplicación), / (división), y % (módulo), en todos los números enteros y de punto flotante. La siguiente tabla contempla todas las operaciones aritméticas binarias en Java:

Operador	Uso	Descripción
+	op1 + op2	Suma op1 y op2
-	op1 - op2	Resta op2 de op1
*	op1 * op2	Multiplica op1 y op2
/	op1 / op2	Divide op1 por op2
%	op1 % op2	Obtiene el resto de dividir op1 por op2

Nota: *El lenguaje Java extiende la definición del operador + para incluir la concatenación de cadenas.*

Los operadores + y - tienen versiones unarias que seleccionan el signo del operando:

Operador	**Uso**	**Descripción**
+	+ op	Indica un valor positivo
-	- op	Niega el operando

Además, existen dos operadores de atajos aritméticos, ++ que incrementan en uno su operando, y -- que decrementan en uno el valor de su operando.

A.5. Incrementos y Decrementos

Como en C y C++, los operadores ++ y -- se utilizan para incrementar o decrementar un valor en 1, a diferencia de C y de C++, Java permite que el valor a modificar sea de punto flotante.

Estos operadores se pueden fijar antes o después; es decir, el ++ o el -- puede aparecer antes o después del valor que incrementa o decrece. Veamos los siguientes ejemplos:

Operador	**Uso**	**Descripción**
++	op ++	Incrementa op en 1; evalúa el valor antes de incrementar
++	++ op	Incrementa op en 1; evalúa el valor después de incrementar
--	op --	Decrementa op en 1; evalúa el valor antes de decrementar
--	-- op	Decrementa op en 1; evalúa el valor después de decrementar

A.6. Operadores Relacionales y Lógicos

A.6.1. Operadores Relacionales:

Los valores relacionales comparan dos valores y determinan la relación entre ellos. Por ejemplo, != devuelve true si los dos operandos son distintos.

Esta tabla contempla los operadores relacionales de Java:

Operador	Uso	Devuelve true si
>	op1 > op2	op1 es mayor que op2
>=	op1 >= op2	op1 es mayor o igual que op2
<	op1 < op2	op1 es menor que op2
<=	op1 <= op2	op1 es menor o igual que op2
==	op1 == op2	op1 y op2 son iguales
!=	op1 != op2	op1 y op2 son distintos

Frecuentemente los operadores relacionales se utilizan con otro juego de operadores, los operadores condicionales, para construir expresiones de decisión más complejas. Uno de estos operadores es && que realiza la operación Y booleana . Por ejemplo puedes utilizar dos operadores relacionales diferentes junto con && para determinar si ambas relaciones son ciertas. La siguiente línea de código utiliza esta técnica para determinar si un índice de un array está entre dos límites, esto es, para determinar si el índice es mayor que 0 o menor que NUM_ENTRIES (que se ha definido previamente como un valor constante):

0 < index && index < NUM_ENTRIES

Observa que en algunas situaciones, el segundo operando de un operador relacional no será evaluado. Consideremos esta sentencia:

((count > NUM_ENTRIES) && (System.in.read() != -1))

Si count es menor que NUM_ENTRIES, la parte izquierda del operando de && evalúa a false. El operador && sólo devuelve true si los dos operandos son verdaderos. Por eso, en esta situación se puede deteminar el valor de && sin evaluar el operador de la derecha. En un caso como este, Java no evalúa el operando de la derecha. Así no se llamará a System.in.read() y no se leerá un carácter de la entrada estándar.

A.6.2. Operadores Lógicos:

La siguiente tabla muestra tres operadores lógicos (condicionales):

Operador	Uso	Devuelve true si
&&	op1 && op2	op1 y op2 son verdaderos
||	op1 || op2	uno de los dos es verdadero
!	! op	op es falso

El operador & se puede utilizar como un sinónimo de && si ambos operadores son boléanos. Similarmente, | es un sinónimo de || si ambos operandos son boléanos.

A.7. Operadores de Bits

Los operadores de desplazamiento permiten realizar una manipulación de los bits de los datos. Esta tabla sumariza los operadores lógicos y de desplazamiento disponibles en el lenguaje Java:

Operador	Uso	Descripción
>>	op1 >> op2	desplaza a la derecha op2 bits de op1
<<	op1 << op2	desplaza a la izquierda op2 bits de op1
>>>	op1 >>> op2	desplaza a la derecha op2 bits de op1(sin signo)
&	op1 & op2	bitwise and
|	op1 | op2	bitwise or
^	op1 ^ op2	bitwise xor
~	~ op	bitwise complemento

Los tres operadores de desplazamiento simplemente desplazan los bits del operando de la izquierda el número de posiciones indicadas por el operador de la derecha. Los desplazamientos ocurren en la dirección indicada por el propio operador. Por ejemplo:
13 >> 1;
desplaza los bits del entero 13 una posición a la derecha. La representación binaria del número 13 es 1101. El resultado de la operación de desplazamiento es 110 o el 6 decimal. Observe que el bit situado más a la derecha desaparece. Un desplazamiento a la derecha de un bit es equivalente, pero más eficiente que, dividir el operando de la izquierda por dos. Un desplazamiento a la izquierda es equivalente a multiplicar por dos.
Los otros operadores realizan las funciones lógicas para cada uno de los pares de bits de cada operando. La función "y" (and) activa el bit resultante si los dos operandos son 1.

op1	op2	resultado
0	0	0
0	1	0
1	0	0
1	1	1

Supóngase que se quiere evaluar los valores 12 "and" 13:
12 & 13
El resultado de esta operación es 12. ¿Por qué? Bien, la representación binaria de 12 es 1100 y la de 13 es 1101. La función "and" activa los bits resultantes cuando los bits de los dos operandos son 1, de otra forma el resultado es 0. Entonces si colocas en línea los dos operandos y realizas la función "and", puedes ver que los dos bits de mayor peso (los dos bits situados más a la izquierda de cada número) son 1 así el bit resultante de cada uno es 1. Los dos bits de menor peso se evalúan a 0 poque al menos uno de los dos operandos es 0:

```
   1101
& 1100
-------
1100
```

El operador | realiza la operación “or” inclusiva y el operador ^ realiza la operación “or” exclusiva. “or” inclusiva significa que si uno de los dos operandos es 1 el resultado es 1.

op1	op2	resultado
0	0	0
0	1	1
1	0	1
1	1	1

“or” exclusivo significa que si los dos operandos son diferentes el resultado es 1, de otra forma el resultado es 0:

op1	op2	resultado
0	0	0
0	1	1
1	0	1
1	1	0

Y finalmente el operador complemento invierte el valor de cada uno de los bites del operando: si el bit del operando es 1 el resultado es 0 y si el bit del operando es 0 el resultado es 1.

A.8. Operadores de Asignación

Puedes utilizar el operador de asignación =, para asignar un valor a otro. Además del operador de asignación básico, Java proporciona varios operadores de asignación que permiten realizar operaciones aritméticas, lógicas o de bits y una operación de asignación al mismo tiempo. Específicamente, supóngase que se quiere añadir un número a una variable y asignar el resultado dentro de la misma variable, como esto:

```
i = i + 2;
```

Se puede hacer lo mismo utilizando el operador +=.

i += 2;

Las dos líneas de código anteriores son equivalentes.

La siguiente tabla lista los operadores de asignación y sus equivalentes:

Operador	***Uso***	***Equivale 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
|=	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

A.9. Precedencia de los Operadores

operadores sufijo	[] . (*params*) *expr*++ *expr*--
operadores unarios	++*expr* --*expr* +*expr* -*expr* ~ !
creación o tipo	new (*type*)*expr*
multiplicadores	* / %
suma/resta	+ -
desplazamiento	<< >> >>>
relacionales	< > <= >= instanceof
igualdad	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
AND lógico	&&
OR lógico	||
condicional	? :
asignación	= += -= *= /= %= ^= &= |= <<= >>= >>>=

A.10. Sentencias de Control de Flujo

Las sentencias de control de flujo determinan el orden en que se ejecutarán las otras sentencias dentro del programa. El lenguaje Java soporta varias sentencias de control de flujo, incluyendo:

Sentencias	palabras clave
condicionales	if-else, switch-case
bucles	for, while, do-while
excepciones	try-catch-finally, throw
miscelaneas	break, continue, *label*:, return

A.10.1. Condicional if

La condicional ***if*** permite ejecutar diferentes partes del código con base en una simple prueba, contiene la palabra clave ***if***, seguida de una prueba booleana y de un enunciado a ejecutar si la prueba es verdadera:

La diferencia entre los condicionales en Java y C o C++ es que la prueba debe regresar un valor booleano (falso o verdadero). El formato de la instrucción if es la siguiente:

```
If  (condición_es_verdadera)
sentencia;
```

Para lograr que se ejecute el conjunto de instrucciones en el caso de la condición falsa, debe utilizarse ***else***. El formato de la instrucción **else** es como sigue:

```
If  (condición_es_verdadera)
Sentencia;
else
Sentencia;
```

El operador condicional

Una alternativa para utilizar las palabras claves *if* y *else* en un enunciado condicional es usar el operador condicional también conocido como el operador ternario.

El operador condicional es más útil para condicionales cortos o sencillos, y tiene esta apariencia:
test ? trueresult : falseresult

El test es una expresión que regresa true o false, al igual que en la prueba del enunciado if. En el siguiente ejemplo, este condicional prueba los valores de x y y, regresa al más pequeño de los dos y asigna ese valor a la variable smaller:

```
int smaller = x < y ? x : y;
```

A.10.2. Condicionales Switch

Este condicional nos sirve para probar alguna variable contra algún valor, y si no coincide con ese valor, probarla con otro y así sucesivamente.

```
switch (test) {

case valueOne:
        resultOne;
       break;
case valueTwo:
        resultTwo;
       break;
case valueThree:
        resultThree;
       break;
    …
default: defaultresult;
}
```

A.11. Ciclos for

Este ciclo repite una declaración o un bloque de enunciados un número de veces hasta que una condición se cumple. Estos ciclos con frecuencia se utilizan para una iteración sencilla en donde usted repite un bloque de enunciados un cierto número de veces y después se detiene; aunque también puede usarlos para cualquier clase de ciclos. El ciclo *for* en Java tiene esta apariencia:

```
for (initialization; test; increment){
staments
}
```

El inicio del ciclo for tiene tres partes:

1. ***initialization*** es una expresión que inicializa el principio del ciclo. Si tiene un índice, esta expresión puede declararla e inicializarla. Las variables que se declararán dentro del ciclo; dejan de existir después de acabar la ejecución del ciclo.
2. ***test*** es la prueba que ocurre después de cada vuelta del ciclo. La prueba puede ser una expresión booleana o una función que regresa un valor booleano. Si la prueba es verdadera el ciclo se ejecutará, de lo contrario el ciclo detiene su ejecución.

3. ***increment*** es una expresión o llamada de función. Por lo común el incremento se utiliza para cambiar el valor del índice del ciclo a fin de acercar el estado del ciclo a false y se complete.

La parte del enunciado del ciclo for son los enunciados que se ejecutan cada vez que se itera el ciclo, al igual que con if puede incluir un solo enunciado o un bloque. Cualquiera de las partes de un ciclo for pueden ser enunciados vacíos, es decir, puede incluir un solo punto y coma sin ninguna expresión o declaración, y esa parte del ciclo se ignorará.

Es importante recordar que no debe de colocar punto y coma después de la primera línea del ciclo for.

A.12. Ciclos while y do. . .while

Al igual que los ciclos for, le permiten a un código de bloque Java ejecutarse de manera repetida hasta encontrar una condición específica. Utilizar uno de estos tres ciclos solo es cuestión de estilo de programación

Los ciclos while y do son exactamente los mismos que en C y C++, a excepción de que su condición de prueba debe ser un booleano.

A.12.1. Ciclos while

Se utilizan para repetir un enunciado o bloque de enunciados, siempre que una condición en particular sea verdadera. Los ciclos while tienen esta apariencia:

```
while (condition) {
        bodyOfLoop
        }
```

La condition es una expresión booleana. Si regresa true, el ciclo while ejecutará los enunciados dentro del bodyOfLoop, y después prueba la condición de nuevo, repitiéndola hasta que sea falsa.

Si la condición es en un principio falsa la primera vez que se prueba, el cuerpo del ciclo while nunca se ejecutará. Si necesita que el ciclo por lo menos se ejecute una vez, tiene dos opciones a elegir:

1. Duplicar el cuerpo del ciclo fuera del ciclo **while**.
2. Utilizar un ciclo **do**.

El ciclo **do** se considera la mejor opción en ambas.

A.12.2. Ciclos do...while

El ciclo do es como while, excepto que do ejecuta un enunciado o bloque hasta que la condición es false. La principal diferencia es que los ciclos while prueban la condición antes de realizar el ciclo, lo cual hace posible que el cuerpo del ciclo nunca se ejecute si la condición es falsa la primera vez que se prueba. Así los ciclos do ejecutan el cuerpo del ciclo por lo menos una vez antes de probar la condición. Los ciclos do se ven así:

```
do{
bodyOfLoop
} while (condition);
```

NOTA: En todos los ciclos (for, while y do), éstos se terminan cuando la condición que prueba se cumple. Para salir de manera anticipada del ciclo, puede utilizar las palabras clave break y continue. El uso de continue es similar al de break, a excepción de que en lugar de detener por completo la ejecución del ciclo, este inicia otra vez en la siguiente iteración. Para los ciclos do y while, esto significa que la ejecución del bloque se inicia de nuevo. Para los ciclos for la expresión de incremento se evalúa y después el bloque se ejecuta .continue es útil cuando se desea tener elementos especiales de condición dentro de un ciclo.

A.13. Excepciones

Java implementa excepciones para facilitar la construcción de código robusto. Cuando ocurre un error en un programa, el código que encuentra el error lanza una excepción, que se puede capturar y recuperarse de ella. Java proporciona muchas excepciones predefinidas.

try-catch-throw

```
try {
    // Normalmente este código corre desde arriba del bloque hasta
   // el final sin problemas. Pero algunas veces puede ocasionar
  // excepciones o salir del bloque con una sentencia break,
 // continue, o return.

    sentencias;
} catch( AlgunaException e1 ) {
  // El manejo de un objeto de excepcion el de tipo AlgunaExcepcion
 // o de una subclase de ese tipo.
    sentencias;
 catch( OtraException e2 ) {
// El manejo de un objeto de excepción e2 de tipo OtraExcepcion
// o de una subclase de ese tipo.
}
```

try (tratar)

La cláusula ***try*** simplemente establece un bloque de código que habrá de manejar todas las excepciones y salidas anormales(vía break, continue o propagación de excepción). La clásula try, por sí misma, no hace nada interesante.

catch (atrapar)

Un bloque try puede ir seguido de cero o más cláusulas catch, las cuales especifican el código que manejará los distintos tipos de excepciones. Las cláusulas catch tienen una sintaxis inusual: cada una se declara con un argumento, parecido a un argumento de método. Este argumento debe ser del tipo Throwable o una subclase. Cuando ocurre una excepción, se invoca la primera cláusula catch que tenga un argumento del tipo adecuado. El tipo de argumento debe concordar con el tipo de objeto de excepción, o ser una superclase de la excepción. Este argumento catch es válido sólo dentro del bloque catch, y hacer referencia al objeto de excepción real que fue lanzado.

A.14. Arreglos y Cadenas

Al igual que otros lenguajes de programación, Java permite juntar y manejar múltiples valores a través de un objeto array (matriz). También se pueden manejar datos compuestos de múltiples caracteres utilizando el objeto String (cadena).

A.14.1. Arreglos

Esta sección te enseñará todo lo que necesitas para crear y utilizar arrays en tus programas Java.

Como otras variables, antes de poder utilizar un array primero se debe declarar. De nuevo, al igual que otras variables, la declaración de un array tiene dos componentes primarios: el tipo del array y su nombre. Un tipo de array incluye el tipo de dato de los elementos que va contener el array. Por ejemplo, el tipo de dato para un array que sólo va a contener elementos enteros es un array de enteros. No puede existir un array de tipo de datos genérico en el que el tipo de sus elementos esté indefinido cuando se declara el array. Aquí tienes la declaración de un array de enteros:

int[] arrayDeEnteros;

La parte **int[]** de la declaración indica que **arrayDeEnteros** es un array de enteros.
La declaración no asigna ninguna memoria para contener los elementos del array.

Si se intenta asignar un valor o acceder a cualquier elemento de **arrayDeEnteros** antes de haber asignado la memoria para él, el compilador dará un error como este y no compilará el programa:

testing.java:64: Variable arraydeenteros may not have been initialized.

Para asignar memoria a los elementos de un array, primero se debe ejemplarizar el array. Se puede hacer esto utilizando el operador **new** de Java. (Realmente, los pasos que se deben seguir para crear un array son similares a los se deben seguir para crear un objeto de una clase: declaración, ejemplarización e inicialización.

La siguiente sentencia asigna la suficiente memoria para que **arrayDeEnteros** pueda contener diez enteros.

```
int[] arraydeenteros = new int[10]
```

En general, cuando se crea un array, se utiliza el operador **new**, más el tipo de dato de los elementos del array, más el número de elementos deseados encerrado entre cochetes cuadrados ('[' y ']').

TipodeElemento[] NombredeArray = new TipodeElementos[tamanoArray]
Ahora que se ha asignado memoria para un array ya se pueden asignar valores a los elemetos y recuperar esos valores:

```
for (int j = 0; j < arrayDeEnteros.length; j ++) {
   arrayDeEnteros[j] = j;
   System.out.println("[j] = " + arrayDeEnteros[j]);
}
```

Como se puede ver en el ejemplo anterior, para referirse a un elemento del array, se añade corchetes cuadrados al nombre del array. Entre los corchetes caudrados se indica (bien con una variable o con una expresión) el índice del elemento al que se quiere acceder. Observa que en Java, el índice del array empieza en 0 y termina en la longitud del array menos uno.

Hay otro elemento interesante en el pequeño ejemplo anterior. El bucle **for** itera sobrecada elemento de **arrayDeEnteros** asignándole valores e imprimiendo esos valores. Observa el uso de **arrayDeEnteros.length** para obtener el tamaño real del array. **length** es una propiedad proporcionada para todos los arrays de Java.
Los arrays pueden contener cualquier tipo de dato legal en Java incluyendo los tipos de referencia como son los objetos u otros array. Por ejemplo, el siguiente ejemplo declara un array que puede contener diez objetos String.

```
String[] arrayDeStrings = new String[10];
```

Los elementos en este array son del tipo referencia, esto es, cada elemento contiene una referencia a un objeto String. En este punto, se ha asignado suficiente memoria para contener las referencias a los Strings, pero no se ha asignado memoria para los propios strings. Si se intenta acceder a uno de los elementos de **arraydeStrings** obtendrá una excepción 'NullPointerException' porque el array está vacio y no contiene ni cadenas ni objetos String. Se debe asignar memoria de forma separada para los objetos String:

```
for (int i = 0; i < arraydeStrings.length; i ++) {
   arraydeStrings[i] = new String("Hello " + i);
}
```

A.14.2. Cadenas

Una secuencia de datos del tipo carácter se llama un string (cadena) y en el entorno Java está implementada por la clase String (un miembro del paquete java.lang).

String[] args;

Este código declara explícitamente un array, llamado **args**, que contiene objetos del tipo String. Los corchetes vacios indican que la longitud del array no se conoce en el momento de la compilación, porque el array se pasa en el momento de la ejecución.

El segundo uso de String es el uso de cadenas literales (una cadena de caracteres entre comillas " y "):

"Hola mundo!"

El compilador asigna implicitamente espacio para un objeto String cuando encuentra una cadena literal.

Los objetos String son inmutables - es decir, no se pueden modificar una vez que han sido creados. El paquete java.lang proporciona una clase diferente, StringBuffer, que se podrá utilizar para crear y manipular caracteres al vuelo.

Concatenación de Cadenas

Java permite concatenar cadenas facilmente utilizando el operador **+**. El siguiente fragmento de código concatena tres cadenas para producir su salida:

"La entrada tiene " + contador + " caracteres."

Dos de las cadenas concatenadas son cadenas literales: "**La entrada tiene** " y " **caracteres.**". La tercera cadena - la del cadena y luego se concatena con las otras.