1. JAVA

1.1 ¿QUÉ ES JAVA?

Sun describió a Java de la siguiente forma:

Java: Es un lenguaje simple, orientado a objetos, distribuido, interpretado, robusto, seguro, de arquitectura neutra, portable, de alto desempeño, de hilos múltiples y dinámico.

- Java es un lenguaje de programación orientado a objetos.
  A diferencia de C++, Java se diseñó desde un principio para estar orientado a objetos. La mayoría de las cosas en Java son objetos; los primitivos tipos numéricos, carácter y booleano son la única excepción. En Java, las cadenas se representan con objetos, los hilos, etc.

- Como los programas de Java se compilan en un formato de **bytecode** (código de bytes) de arquitectura neutral, una aplicación de Java se puede ejecutar en cualquier sistema, siempre y cuando dicho sistema instrumente la máquina virtual de Java. Esto resulta muy importante para las aplicaciones distribuidas en Internet u otras redes heterogéneas.

Para asegurar que los programas de Java sean realmente independientes, de la plataforma, hay una arquitectura única sobre la que se compilan todos los programas en Java. Es decir, cuando se compila para una plataforma en Windows/Intel x86 se obtiene la misma salida que la compilada en un sistema Macintosh o Unix. El compilador no compila para la plataforma de origen, sino para una plataforma abstracta llamada ***máquina virtual de Java***, o ***JVM-Java Virtual Machine***.

- A Java también se le denomina lenguaje *distribuido*. Esto significa, simplemente, que proporciona un soporte de alto nivel para redes. Por ejemplo, la clase URL y las clases relacionadas en el paquete java.net hacen que la lectura de un archivo o una fuente remota sea tan fácil como leer un archivo local.

- Java se ha diseñado para escribir software robusto y muy confiable.

- Una de las cosas que hace simple a Java es la carencia de punteros y aritmética de puntero. Todos los accesos a arreglos y cadenas se verifican al tiempo de ejecución. Las formas de los objetos de un tipo a otro también se verifican al momento de la ejecución. La recolección automática de basura en Java evita que la memoria tenga fugas, así como la presencia de otros defectos perniciosos relacionados con la asignación y desasignación de memoria.

- El manejo de excepciones es otra característica de Java que contribuye a formar programas más robustos. Una excepción es una señal de que ha ocurrido una condición excepcional, por ejemplo, un error de "no se puede encontrar el archivo". Con esto puede simplificar la tarea del manejo y recuperación de errores.

- Uno de los aspectos más resonados de Java es que se trata de un lenguaje seguro. Proporciona varias capas de controles de seguridad que protegen contra código malicioso; estas capas permiten a los usuarios ejecutar con comodidad programas desconocidos, como los applet.

- Java es un lenguaje de hilos múltiples; proporciona soporte para varios hilos de ejecución que pueden manejar diferentes tareas. Un beneficio importante de los hilos múltiples es que se mejora el desempeño interactivo de las aplicaciones gráficas para el usuario.

1.2 JAVA VS C++

Java se asemeja mucho a C y C++. Esta similitud, evidentemente intencionada, es la mejor herramienta para los programadores, ya que facilita en gran manera su transición a Java. Desafortunadamente, tantas similitudes hacen que no nos paremos en algunas diferencias que son vitales. La terminología utilizada en estos lenguajes, a veces es la misma, pero hay grandes diferencias subyacentes en su significado.

C tiene tipos de datos básicos y punteros. C++ modifica un poco este panorama y le añade los tipos *referencia*. Java también especifica sus tipos primitivos, elimina cualquier tipo de punteros y tiene tipos referencia mucho más claros.

Conocemos ya ampliamente todos los tipos básicos de datos: datos base, integrados, primitivos e internos; que son muy semejantes en C, C++ y Java; aunque Java simplifica un poco su uso a los desarrolladores haciendo que el chequeo de tipos sea bastante más rígido. Además, Java añade los tipos boolean y hace imprescindible el uso de este tipo booleano en sentencias condicionales.

1.3 LA SIMPLICIDAD DE JAVA

Java ha sido diseñado de modo de eliminar las complejidades de otros lenguajes como C y C++.

Si bien Java posee una sintaxis similar a C, con el objeto de facilitar la migración de C hacia a Java, Java es semánticamente muy distinto a C:

- Java no posee aritmética de punteros: La aritmética de punteros es el origen de muchos errores de programación que no se manifiestan durante la depuración y que una vez que el usuario los detecta son difíciles de resolver.

- No se necesita hacer *delete*: Determinar el momento en que se debe liberar el espacio ocupado por un objeto es un problema difícil de resolver correctamente. Esto también es el origen a errores difíciles de detectar y solucionar.

- No hay herencia múltiple: En C++ esta característica da origen a muchas situaciones de borde en donde es difícil predecir cuál será el resultado. Por esta razón en Java se opta por herencia simple que es mucho más simple de aprender y dominar.

1.4 JAVA ES INDEPENDIENTE DE LA PLATAFORMA

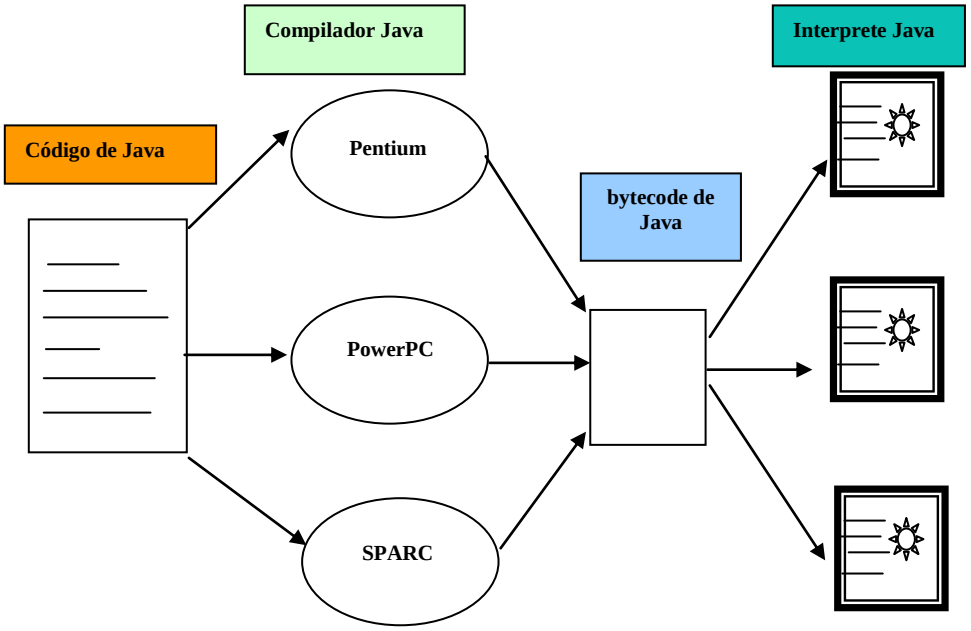
La independencia de la plataforma es la capacidad del programa de trasladarse con facilidad de un sistema computacional a otro.

Esta independencia de la plataforma es una de las principales ventajas que tiene Java sobre otros lenguajes de programación, en particular para los sistemas que necesitan trabajar en varias plataformas.

A nivel de código fuente, los tipos primitivos de datos Java, tiene tamaños consistentes, en todas las plataformas de desarrollo. Los fundamentos de bibliotecas de Java facilitan la escritura del código, el cual puede desplazarse de plataforma a plataforma sin necesidad de volver a escribirlo.

Los archivos binarios Java, también son independientes de la plataforma, y pueden compilarse en múltiples plataformas sin necesidad de volver a compilar la fuente, esto se logra, ya que los archivos binario Java, se encuentran en una forma llamada *bytecode* (conjunto de instrucciones parecidas al lenguaje de máquina, pero que no son específicas para un procesador).

El ambiente de desarrollo Java tiene dos partes: un compilador y un intérprete Java. El compilador Java toma su programa Java y en lugar de generar códigos de máquina para sus archivos fuente, genera un bytecode. Para ejecutar un programa Java debe utilizar un intérprete de bytecode el cual a su vez ejecuta su programa Java. Puede ejecutar el intérprete por si mismo o en el caso de los applets puede recurrir a los visualizadores que los ejecutan.



La desventaja de utilizar bytecode se halla en la velocidad de ejecución, puesto que los programas específicos del sistema corren directamente en el hardware en que se compilaron estos, ya que los bytecodes deben ser procesados por el intérprete

Los programas en Java pueden ejecutarse en cualquiera de las siguientes plataformas, sin necesidad de hacer cambios:

- Windows/95 y /NT
- Power/Mac
- Unix (Solaris, Silicon Graphics, ...)

La compatibilidad es total:

- A nivel de fuentes: El lenguaje es exactamente el mismo en todas las plataformas.

- A nivel de bibliotecas: En todas las plataformas están presentes las mismas bibliotecas estándares.

- A nivel del código compilado: el código intermedio que genera el compilador es el mismo para todas las plataformas. Lo que cambia es el intérprete del código intermedio.

1.5 SEGURIDAD EN JAVA

La seguridad es un aspecto importante en Java, el visualizador baja el código de toda la red y lo ejecuta en el anfitrión del usuario, se incluye varias capas de seguridad tales como:

- El propio lenguaje Java incluye restricciones cerradas de acceso a memoria muy diferentes al modelo de memoria que utiliza el lenguaje C. Estas restricciones incluyen la remoción de apuntadores aritméticos y de operadores de conversión forzada ilegales.
- Una rutina de verificación de códigos de byte en el intérprete de Java verifica que los códigos de byte (Bytecodes) no violen ninguna construcción del lenguaje (lo que podría suceder si utiliza un compilador de Java alterado). Esta rutina de verificación se asegura de que el código no falsifique apuntadores, memoria de acceso restringido u objetos de acceso diferentes a los que corresponden a sus definiciones. Esta verificación también asegura que las llamadas al método incluyen el número correcto de argumentos del tipo adecuado, y que no hay desbordamiento de pilas.
- Una verificación del nombre de la clase y de las restricciones de acceso sobre la carga.
- Una interfaz de sistema de seguridad que refuerza las políticas de seguridad en varios niveles.
- Al nivel de acceso del archivo, si un código de byte intenta el acceso a un archivo para el que no tiene permiso, aparecerá una caja de diálogo para permitir que el usuario continúe o detenga la ejecución.
- Al nivel de red, se tendrán opciones para emplear codificación de clave pública y otras técnicas de encriptación para verificar la fuente del código y su integridad después de haber pasado por la red. Esta tecnología de encriptación será la clave parra transacciones financieras seguras a través de la red.
- Al momento de la ejecución, puede utilizarse la información sobre el origen del código de byte para decidir lo que el código puede hacer. El mecanismo de seguridad puede indicar si un código de byte se originó o no desde el interior de una firewall (barrera de protección). También se puede definir una política de seguridad que restrinja el código dónde no se confía.
- Java siempre checa los índices al acceder un arreglo.
- Java realiza chequeo de tipos durante la compilación (al igual que C). En una asignación entre punteros el compilador verifica que los tipos sean compatibles.
- Además, Java realiza chequeo de tipos durante la ejecución (cosa que C y C++ no hacen). Cuando un programa usa un cast para accesar un objeto como si fuese de un tipo específico, se verifica durante la ejecución que el objeto en cuestión sea compatible con el cast que se le aplica. Si el objeto no es compatible, entonces se levanta una excepción que informa al programador la línea exacta en donde está la fuente del error.

- Java posee un recolector de basuras que administra automáticamente la memoria. Es el recolector el que determina cuando se puede liberar el espacio ocupado por un objeto. El programador no puede liberar explícitamente el espacio ocupado por un objeto.
- Java no posee aritmética de punteros, porque es una propiedad que no se necesita para programar aplicaciones. En C sólo se necesita la aritmética de punteros para programa *malloc/free* o para programar el núcleo del sistema operativo.

Por lo tanto Java no es un lenguaje para hacer sistemas operativos o administradores de memoria, pero sí es un excelente lenguaje para programar aplicaciones.

1.6 JAVA ES FLEXIBLE

Pascal también es un lenguaje robusto, pero logra su robustez prohibiendo tener punteros a objetos de tipo desconocido. Lamentablemente esta prohibición es demasiado rígida. Aunque son pocos los casos en que se necesita tener punteros a objetos de tipo desconocido, las contorsiones que están obligados a realizar los programadores cuando necesitan estos punteros dan origen a programas ilegibles.

Lisp por su parte es un lenguaje flexible y robusto. Todas las variables son punteros a objetos de cualquier tipo (un arreglo, un elemento de lista, etc.). El tipo del objeto se encuentra almacenado en el mismo objeto. Durante la ejecución, en cada operación se chequea que el tipo del objeto manipulado sea del tipo apropiado. Esto da flexibilidad a los programadores sin sacrificar la robustez. Lamentablemente, esto hace que los programas en Lisp sean poco legibles debido a que al estudiar su código es difícil determinar cuál es el tipo del objeto que referencia una variable.

Java combina flexibilidad, robustez y legibilidad gracias a una mezcla de chequeo de tipos durante la compilación y durante la ejecución. En Java se pueden tener punteros a objetos de un tipo específico y también se pueden tener punteros a objetos de cualquier tipo. Estos punteros se pueden convertir a punteros de un tipo específico aplicando un cast, en cuyo caso se chequea en tiempo de ejecución de que el objeto sea de un tipo compatible.

El programador usa entonces punteros de tipo específico en la mayoría de los casos con el fin de ganar legibilidad y en unos pocos casos usa punteros a tipos desconocidos cuando necesita tener flexibilidad. Por lo tanto Java combina la robustez de Pascal con la flexibilidad de Lisp, sin que lo programas pierdan legibilidad en ningún caso.

1.7 JAVA ADMINISTRA AUTOMÁTICAMENTE LA MEMORIA

En Java los programadores no necesitan preocuparse de liberar un trozo de memoria cuando ya no lo necesitan. Es el recolector de basuras el que determina cuando se puede liberar la memoria ocupada por un objeto.

Un recolector de basuras es un gran aporte a la productividad. Se ha estudiado en casos concretos que los programadores han dedicado un 40% del tiempo de desarrollo a determinar en qué momento se puede liberar un trozo de memoria.

Además este porcentaje de tiempo aumenta a medida que aumenta la complejidad del software en desarrollo. Es relativamente sencillo liberar correctamente la memoria en un programa de 1000 líneas. Sin embargo, es difícil hacerlo en un programa de 10000 líneas. Y se puede postular que es imposible liberar correctamente la memoria en un programa de 100000 líneas.

Para entender mejor esta afirmación, supongamos que hicimos un programa de 1000 líneas hace un par de meses y ahora necesitamos hacer algunas modificaciones. Ahora hemos olvidado gran parte de los detalles de la lógica de este programa y ya no es sencillo determinar si un puntero referencia un objeto que todavía existe, o si ya fue liberado. Peor aún, suponga que el programa fue hecho por otra persona y evalúe cuan probable es cometer errores de memoria al tratar de modificar ese programa.

Ahora volvamos al caso de un programa de 100000 líneas. Este tipo de programas los desarrolla un grupo de programadores que pueden tomar años en terminarlo. Cada programador desarrolla un módulo que eventualmente utiliza objetos de otros módulos desarrollados por otros programadores. ¿Quién libera la memoria de estos objetos? ¿Cómo se ponen de acuerdo los programadores sobre cuándo y quién libera un objeto compartido? ¿Como probar el programa completo ante las infinitas condiciones de borde que pueden existir en un programa de 100000 líneas?

Es inevitable que la fase de prueba dejará pasar errores en el manejo de memoria que sólo serán detectados más tarde por el usuario final. Probablemente se incorporan otros errores en la fase de mantenimiento.

Se puede concluir:

- Todo programa de 100000 líneas que libera explícitamente la memoria tiene errores latentes.

- Sin un recolector de basuras no hay verdadera modularidad.

- Un recolector de basuras resuelve todos los problemas de manejo de memoria en forma trivial.

La pregunta es: ¿Cuál es el impacto de un recolector de basura en el desempeño de un programa?

El sobrecosto de la recolección de basuras no es superior al 100%. Es decir si se tiene un programa que libera explícitamente la memoria y que toma tiempo X, el mismo programa modificado de modo que utilice un recolector de basuras para liberar la memoria tomará un tiempo no superior a 2X.

Este sobrecosto no es importante si se considera el periódico incremento en la velocidad de los procesadores. El impacto que un recolector de basura en el tiempo de desarrollo y en la confiabilidad del software resultante es muchos más importante que la pérdida en eficiencia.

1.8 LA APLICACIÓN HELLO WORLD

Una aplicación es un programa convencional que se invoca desde el intérprete de comandos. Este programa se carga directamente desde el disco y no de la red Internet.

Ahora veremos la aplicación más simple que se puede escribir en Java: el clásico ``*Hello World*''.

- Crear un archivo llamado Hello1.java con:

```
// La aplicación Hello World!
public class Hello1 {
    public static void main (String args[]) {
        System.out.println("Hello World!");
    }
}
```

- Compilar con: javac Hello1.java

- Ejecutar con: java Hello1

Observaciones:

- La primera línea es un comentario. Todo lo que viene después de la secuencia // hasta el fin de línea es un comentario.

  Java también acepta comentarios ``a la C'': /* ... */

- Luego viene la definición de una clase llamada Hello1:

  public class Hello1 { ... }

  En Java un programa es un conjunto de definiciones de clases que están dispuestas en uno o más archivos.

- Dentro de la clase Hello1 se define el método *main*:

  public static void main (String args[]) { ... }

  En una clase se definen uno o más métodos.

  - ✓ Las palabras *public* y *static* son atributos del método que discutiremos más tarde.

  - ✓ La palabra *void* indica que el método *main* no retorna ningún valor.

  - ✓ La forma *(String args[])* es la definición de los argumentos que recibe el método *main*. En este caso se recibe un argumento. Los paréntesis [] indican que el argumentos es un arreglo y la palabra *String* es el tipo de los elementos del arreglo.

    Por lo tanto *main* recibe como argumento un arreglo de strings que corresponden a los argumentos con que se invoca el programa.

  - ✓ La instrucción *System.out.println(...)* despliega un *string* en la consola.

    Java no posee una sintaxis abreviada para desplegar *strings*.

Consideraciones importantes:

- El nombre del archivo (Hello1.java) siempre debe ser el nombre de la clase (Hello1) con la extensión ``.java''.

- Todas las aplicaciones deben definir el método *main*.

- Al invocar el intérprete de java con java Hello1, se busca y se invoca un método *main* que textualmente haya sido definido con:

  public static void main (String args[]) { ... }

No cambie el nombre de este procedimiento ni omita ninguno de sus atributos. Tampoco cambie el tipo de los argumentos o el valor retornado.

1.9 EL APPLET HELLO WORLD

Un applet es un programa que anima una porción de una página Web. Se recupera a partir de la red y corre en la máquina del usuario, pero con muchas restricciones de modo que no pueda afectar la integridad del ambiente del usuario.

A continuación veremos la versión applet del ejemplo anterior. Es decir un programa que coloca en una página Web el mensaje ``Hello World!''.

- Crear el programa fuente Hello2.java con:

```
import java.awt.Graphics;
import java.applet.Applet;

public class Hello2 extends Applet {
  public void paint(Graphics g) {
    g.drawString("Hello world!", 50, 25);
  }
}
```

- Compilar con: *javac Hello2.java*

- Crear la página Hello.html con el siguiente contenido:

```
<html>
<body>
Este es un applet:
<applet code="Hello2.class" width=150 height=25>
</applet>
</body>
</html>
```

Atención: Hello.html debe estar en el mismo directorio que Hello2.java.

- Ver el applet con: *appletviewer* Hello.html

- El mismo applet también se puede ver desde un browser Web como netscape 2.x o superior.

Observaciones:

- Dado que un applet no se invoca desde el intérprete de comandos, no tiene sentido definir el método main. El browser Web notifica al applet que debe dibujar su contenido invocando el método paint. Esto ocurre cada vez que se

muestra la porción de la página html que contiene este applet. Por lo tanto un applet debe definir el método paint (en vez de main).

- Las instrucciones:

  import java.awt.Graphics;
  import java.applet.Applet;

  indican que dentro del archivo las clases *java.awt.Graphics y java.applet.Applet* serán conocidas simplemente como *Graphics* y *Applet*.

- Luego viene la definición de la clase Hello2 con:

  public class Hello2 extends Applet { ... }

Las palabras *extends* indican que Hello2 es una extensión de la clase de biblioteca Applet. Esto significa que Hello2 es casi como Applet, solo que se modifica el comportamiento del método *paint*.

- El método *paint* recibe como argumento un objeto de tipo Graphics que corresponde a una clase de biblioteca. Este objeto se usa para dibujar en la porción de página html asignada al applet.

- La instrucción *g.drawString("Hello world!", 50, 25);* dibuja el string ``Hello World!'' en la porción asignada en las coordenadas (50, 25).

- Además de programar el applet es necesario construir la página html que va a contener el applet. Por limitaciones de espacio en este curso sólo veremos las etiquetas de html que permiten agregar un applet a la página, sin detenernos a ver en profundidad el lenguaje html.

1.10 INSTRUCCIONES Y EXPRESIONES

Una instrucción es lo más sencillo que se puede hacer en Java. Una instrucción forma una sola operación Java. Todas las siguientes son instrucciones Java sencillas:

```
int i =1;
import java.awt.Font;
System.out.println("Esta motocicleta es " + color +  " " + make);
m.engineState = true;
```

Algunas veces, las instrucciones regresan valores. Por ejemplo, cuando sumamos dos valores o una prueba para ver si un valor es igual al otro. Este tipo de enunciados se llaman *expresiones*.

Una instrucción puede ser escrita en un solo renglón o en múltiples renglones y el compilador Java lo entenderá perfectamente.

Java también cuenta con instrucciones compuestas, o bloques de instrucciones, que pueden ser ubicadas en cualquier parte en cualquier lugar donde pondría una sola instrucción. Los enunciados de bloques están rodeados por llaves({}).

1.11 LA DECLARACIÓN IMPORT (IMPORTAR)

La declaración **import** hace que las clases de Java estén disponibles para la clase actual, bajo un nombre abreviado. Las clases públicas de Java siempre estén disponibles vía sus nombres totalmente calificados, siempre y cuando el archivo de la clase en cuestión se pueda encontrar.

Hay dos formas de la declaración import:

import *package.class;*
import *package.*;*

La primera forma permite que la clase especificada en el paquete especificado se conozca por su nombre de clase simplemente. Por ejemplo:

import java.awt.Graphics;
import java.awt.Color;

La segunda forma de la declaración import logra que todas las clases de un paquete estén disponibles mediante su nombre de clase. Por ejemplo:

import java.awt.;*

1.12 REGLAS DE ESCRITURA DE UN PROGRAMA

- Los identificadores pueden iniciar con una letra, un guión de subrayado(_) o un signo de dólares ($), de ninguna manera puede empezar con un número. Después del primer carácter, los nombres pueden incluir cualquier letra o número. Los identificadores no pueden ser una palabra clave de Java.

- El lenguaje Java utiliza un conjunto de caracteres **Unicode**. Esta es la definición del conjunto de signos que no solo ofrece caracteres en el conjunto estándar ASCII, sino también varios millones de caracteres para representar la mayoría de los alfabetos internacionales. Esto significa que puede utilizar caracteres acentuados y otros símbolos, así como caracteres legales en nombre de variables, siempre que cuenten con un número de carácter Unicode sobre 00C0.

- El lenguaje Java es sensible al tamaño de las letras, la cual significa que las mayúsculas son diferentes de las minúsculas. La variable x es diferente de X, rose no es Rose ni ROSE.

Por convención las variables Java tienen nombres significativos, con frecuencia formados de varias palabras combinadas. La primera palabra está en minúsculas, pero las siguientes tienen su letra inicial en mayúsculas.

```
Button the Button
Long reallyBigNumber;
```

- Los espacios en blanco pueden hacer que el código sea más fácil de leer.

- Cada instrucción en Java termina con un punto y coma, de no tenerlo, no se compilará el programa Java.

- Java tiene tres clases de comentarios. Uno de ellos, los delimitadores /* y */, rodean comentarios en varias líneas. Estos comentarios no pueden anidarse, no puede tener comentarios dentro de los comentarios. También puede utilizar la doble diagonal (//) para una solo línea de comentario, donde todo el texto hasta el final de la línea se ignora. El último tipo de comentario empieza con /** y termina con */. El contenido de estos comentarios especiales los emplea el sistema Javadoc, pero a excepción de éste, se emplea de manera idéntica que el primer tipo de comentario. Javadoc se emplea para generar la documentación API del código.

1.13 VARIABLES Y TIPOS DE DATOS

Las variables son lugares en la memoria en donde pueden guardarse valores; tiene un nombre, un tipo y un valor. Antes de poder utilizar una variable, primero debe declararla y a partir de ahí es factible asignarles valores. De hecho Java posee tres clases de valores: de *instancia*, de *clase* y *locales*.

Las ***variables de instancia***, se utilizan para definir atributos o el estado de un objeto en particular.

Las ***variables de clase*** son similares a las variables de instancia, con la diferencia de que sus valores se aplican a todas las instancias de clase, en lugar de tener diferentes valores para el mismo objeto.

Las ***variables locales***, se declaran y utilizan dentro de la definición de método, por ejemplo, para contadores de índice dentro de un ciclo, como variables temporales, o para guardar valores que solo necesita dentro de la definición. También pueden

usarse dentro de bloques({}). Una vez que el método o bloque termina su ejecución, la definición de variable y su método dejan de existir.

Aunque las tres clases de variables se declaran en forma parecida, las variables de clase y de instancia se accesan y se asignan en formas poco diferente a las variables locales

1.13.1 DECLARACIÓN DE VARIABLES

Una variable en Java es un identificador que representa una palabra de memoria que contiene información. El tipo de información almacenado en una variable sólo puede ser del tipo con que se declaró esa variable.

Una variable se declara usando la misma sintaxis de C.

tipoVariable nombre;

Por ejemplo la siguiente tabla indica una declaración, el nombre de la variable introducida y el tipo de información que almacena la variable:

Declaración	Identificador	Tipo
int i;	i	entero
String s;	s	referencia a string
int a[];	a	referencia a arreglo de enteros
int[] b;	b	referencia a arreglo de enteros

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.

Las siguientes instrucciones Java declaran tres variables que usan los tipos int, float y long:

```
class Class1 {
public static void main (String args[ ]) {
        int test_score;
        float salary;
        long distancia_a_la_luna;}
}
```

Asignación e inicialización de variables

Una vez que se ha declarado una variable puede asignarle un valor mediante el uso del operador de asignación =

```
Size =14;
tooMuchCaffiene = true;
int goo;
goo=100;
char coal;
coal= “b”;
```

El siguiente ejemplo asigna valores a tres variables y luego muestra el valor de cada una:

```
class Class1 {
public static void main (String args[ ]) {
  int age = 35;
```

```
        double salary = 25000.75;
        long Distancia_a_la_luna=238857;
        System.out.println("Employee age: " + age);
        System.out.println("Employee salary: " + salary);
        System.out.println("Distancia_a_la_luna: " + Distancia_a_la_luna);
        }
}
```

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:

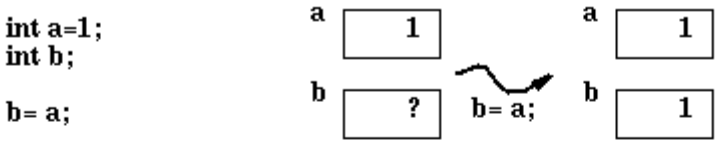
abstract	*boolean*	*break*	*byte*	*case*	*cast*	*catch*
Char	*class*	*cons*	*continue*	*default*	*do*	*double*
Else	*extends*	*final*	*finally*	*float*	*for*	*future*
generic	*goto*	*if*	*implements*	*import*	*inner*	*instanceof*
In	*interface*	*long*	*native*	*new*	*null*	*operator*
Outer	*package*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

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.

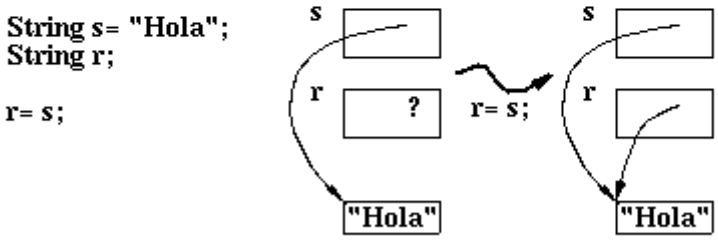
Las variables de tipo referencia a objetos almacenan direcciones y no valores directamente. Una referencia a un objeto es la dirección de un área en memoria destinada a representar ese objeto. El área de memoria se solicita con el operador new.

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.

1.13.2 NOMBRES DE VARIABLES

Un programa se refiere al valor de una variable por su nombre. Por convención, en Java, los nombres de las variables empiezan con una letra minúscula (los nombres de las clases empiezan con una letra mayúscula).
Un nombre de variable Java:

1. debe ser un identificador legal de Java comprendido en una serie de caracteres ***Unicode***. *Unicode* es un sistema de codificación que soporta texto escrito en distintos lenguajes humanos. **Unicode** permite la codificación de 34,168 caracteres. Esto le permite utilizar en sus programas Java varios alfabetos como el Japonés, el Griego, el Ruso o el Hebreo. Esto es importante para que los programadores pueden escribir código en su lenguaje nativo.
2. no puede ser el mismo que una palabra clave o el nombre de un valor booleano (true or false)

3. no deben tener el mismo nombre que otras variables cuyas declaraciones aparezcan en el mismo ámbito.

La regla número 3 implica que podría existir el mismo nombre en otra variable que aparezca en un ámbito diferente.

Por convención, los nombres de variables empiezan por un letra minúscula. Si una variable está compuesta de más de una palabra, como 'nombreDato' las palabras se ponen juntas y cada palabra después de la primera empieza con una letra mayúscula.

1.14 EXPRESIONES Y OPERADORES

Los operadores realizan algunas funciones en uno o dos operandos. Los operadores que requieren un operador se llaman operadores unarios. Por ejemplo, ++ es un operador unario que incrementa el valor su operando en uno.

Los operadores que requieren dos operandos se llaman operadores binarios. El operador = es un operador binario que asigna un valor del operando derecho al operando izquierdo.

Los operadores unarios en Java pueden utilizar la notación de prefijo o de sufijo. La notación de prefijo significa que el operador aparece antes de su operando:

operador operando

La notación de sufijo significa que el operador aparece después de su operando:

operando operador

Todos los operadores binarios de Java tienen la misma notación, es decir aparecen entre los dos operandos:

op1 operador op2

Además de realizar una operación también devuelve un valor. El valor y su tipo dependen del tipo del operador y del tipo de sus operandos. Por ejemplo, los operadores aritméticos (realizan las operaciones de aritmética básica como la suma o la resta) devuelven números, el resultado típico de las operaciones aritméticas. El tipo de datos devuelto por los operadores aritméticos depende del tipo de sus operandos: si sumas dos enteros, obtendrás un entero. Se dice que una operación evalúa su resultado.

Es muy útil dividir los operadores Java en las siguientes categorías: aritméticos, relacionales y condicionales, lógicos y de desplazamiento, y de asignación.

1.14.1 OPERADORES ARITMÉTICOS

El lenguaje Java soporta varios operadores aritméticos, incluyendo + (suma), - (resta), * (multiplicación), / (división), y % (módulo), en todos los números enteros y de punto flotante. Por ejemplo, se puede utilizar este código Java para sumar dos números:

sumaEsto + Esto

O este código para calcular el resto de una división:
divideEsto % porEsto

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 incrementa en uno su operando, y -- que decrementa en uno el valor de su operando.

El siguiente listado es un ejemplo de aritmética sencilla:

```
class Class1 {
public static void main (String args[ ]) {
short x = 6;
int y = 4;
```

```
    float a = 12.5f;
    float b = 7f;
    System.out.println(" x es " + x + ", y es " + y);
    System.out.println(" x + y = " + (x + y ));
    System.out.println(" x - y = " + (x - y ));
    System.out.println(" x * y = " + (x * y ));
    System.out.println(" x % y = " + (x % y ));
    System.out.println(" a es " + a + ", b es " + b);
    System.out.println(" a / b  = " +  (a / b ));
        }
    }
```

El resultado que obtendrá del siguiente listado es:

```
x es 6, y es 4
x + y = 10
x - y = 2
x / y = 1
x % y = 2
a es 12.5, b es 7
a / b = 1.78571
```

El siguiente ejemplo, muestra el resultado de varias operaciones matemáticas simples:

```
class Class1 {
public static void main (String args[ ]) {
        System.out.println(" 5+7 = " + (5+7));
        System.out.println(" 12-7 = " + (12-7));
        System.out.println("1.2345 * 2  = " + (1.2345 * 2 ) );
        System.out.println(" 15 / 3  = " + (15 / 3 ));

        }
}
```

1.14.2 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

Veamos los siguientes ejemplos:

Expresión	Significado
Y = X++	Y=X
	X=X+1
Y = ++X	X=X+1
	Y=X
Z=X++ +Y	Z = X +Y
	X=X+1
Z=X + --Y	Y = Y-1
	Z=X+Y

El siguiente ejemplo ilustra el uso de los operadores de incremento prefijo y sufijo:

```
class Class1 {
public static void main (String args[ ]) {
int small_count = 0;
int big_count = 1000;
System.out.println ("small_count is " + small_count);
System.out.println ("small_count is " + small_count++);
System.out.println ("El valor final de samall_count es " + small_count);
System.out.println ("big_count is " +big_count);
System.out.println ("++big_count is " +   ++ big_count);
System.out.println ("El valor final de big_count es " + big_count);
            }
}
```

El operador de sustracción (--) disminuye en 1 el valor de la variable. Al igual que el operador de incremento soporta el prefijo y el sufijo:

```
class Class1 {
public static void main (String args[ ]) {
int small_count = 0;
int big_count = 1000;
```

```
System.out.println ("small_count is " + small_count);
System.out.println ("small_count is " + small_count--);
System.out.println ("El valor final de samall_count es " + small_count);
System.out.println ("big_count is " +big_count);
System.out.println ("--big_count is " +   -- big_count);
System.out.println ("El valor final de big_count es " + big_count);
            }
}
```

1.14.3 OPERADORES RELACIÓNALES Y LOGICOS

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.

1.14.4 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.

1.14.5 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.

1.14.6 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

1.15 Expresiones

Las expresiones realizan el trabajo de un programa Java. Entre otras cosas, las expresiones se utilizan para calcular y asignar valores a las variables y para controlar el flujo de un programa Java. El trabajo de una expresión se divide en dos partes: realizar los cálculos indicados por los elementos de la expresión y devolver algún valor.

Definición: Una expresión es una serie de variables, operadores y llamadas a métodos (construida de acuerdo a la sintaxis del lenguaje) que evalúa a un valor sencillo.

El tipo del dato devuelto por una expresión depende de los elementos utilizados en la expresión. La expresión **count++** devuelve un entero porque **++** devuelve un valor del mismo tipo que su operando y **count** es un entero. Otras expresiones devuelven valores boléanos, cadenas, etc.

Una expresión de llamada a un método devuelve el valor del método; así el tipo de dato de una expresión de llamada a un método es el mismo tipo de dato que el valor de retorno del método. El método **System.in.read()** se ha declarado como un entero, por lo tanto, la expresión **System.in.read()** devuelve un entero.

La segunda expresión contenida en la sentencia **System.in.read() != -1** utiliza el operador **!=**. Recuerda que este operador comprueba si los dos operandos son distintos. En esta sentencia los operandos son **System.in.read()** y **-1**. **System.in.read()** es un operando válido para **!=** porque devuelve un entero. Así **System.in.read() != -1** compara dos enteros, el valor devuelto por **System.in.read()** y **-1**. El valor devuelto por **!=** es **true** o **false** dependiendo de la salida de la comparación.

Como has podido ver, Java te permite construir expresiones compuestas y sentencias a partir de varias expresiones pequeñas siempre que los tipos de datos requeridos por una parte de la expresión correspondan con los tipos de datos de la otra. También habrás podido concluir del ejemplo anterior, el orden en que se evalúan los componentes de una expresión compuesta.

Por ejemplo, toma la siguiente expresión compuesta:

*x * y * z*

En este ejemplo particular, no importa el orden en que se evalúe la expresión porque el resultado de la multiplicación es independiente del orden. La salida es siempre la misma sin importar el orden en que se apliquen las multiplicaciones. Sin embargo, esto no es cierto para todas las expresiones. Por ejemplo, esta expresión obtiene un resultado diferente dependiendo de si se realiza primero la suma o la división:

x + y / 100

Puedes decirle directamente al compilador de Java cómo quieres que se evalúe una expresión utilizando los paréntesis **(** y **)**. Por ejemplo, para aclarar la sentencia anterior, se podría escribir: *(x + y)/ 100*.

Si no le dices explícitamente al compilador el orden en el que quieres que se realicen las operaciones, él decide basándose en la **precedencia** asignada a los operadores y otros elementos que se utilizan dentro de una expresión. Los operadores con una precedencia más alta se evalúan primero. Por ejemplo. el operador división tiene una precedencia mayor que el operador suma, por eso, en la expresión anterior **x + y / 100**, el compilador evaluará primero **y / 100**. Así

x + y / 100

es equivalente a:

x + (y / 100)

Para hacer que tu código sea más fácil de leer y de mantener deberías explicar e indicar con paréntesis los operadores que se deben evaluar primero.
La tabla siguiente muestra la precedencia asignada a los operadores de Java. Los operadores se han listado por orden de precedencia de mayor a menor. Los operadores con mayor precedencia se evalúan antes que los operadores con un precedencia relativamente menor. Lo operadores con la misma precedencia se evalúan de izquierda a derecha.

Precedencia de Operadores en Java

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	= += -= *= /= %= ^= &= |= <<= >>= >>>=

1.16 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

1.16.1 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;
```

El siguiente ejemplo usa *if* para comparar el valor almacenado en la variable TestScore con 90. Si TestScore es mayor o igual a 90, se muestra un mensaje de felicitación indicando que obtuvo una A, de otra manera, si el valor es menor de 90 el programa termina.

```
class Class1 {
public static void main (String args[ ]) {
      int TestScore = 95;
if   (TestScore >= 90)
      System.out.println ("Felicidades obtuviste una A");
      }
}
```

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;
```

```
class Class1 {
public static void main (String args[ ]) {
      int TestScore = 95;
if (TestScore >= 90){
     System.out.println ("Felicidades obtuviste una A");
     System.out.println ("Tu puntuación es" + TestScore);
}
```

```
else
{
    System.out.println ("Deberías estudiar mas");
    System.out.println ("Perdiste de tu puntuación" + (TestScore - 10) );
     }
  }
}
```

Cuando se lee información del teclado, Java prueba si hay problemas durante la entrada (por ejemplo, no hay mas datos a introducir). Si ocurre un problema Java genera una **excepción,** la cual es un caso excepcional durante la ejecución de un programa. Para ejecutar un programa que reciba datos del teclado, puede hacerlo indicándole al compilador que está consiente de los problemas que pueden ocurrir, esto se logrará incluyendo en el programa las palabras clave throws IOException.

El siguiente programa el usuario va a teclear calificaciones de letra que se introducen en el programa para calcular el promedio en las calificaciones de un grupo:

```
import java.io.*;
class Class1 {
public static void main (String args[ ] ) throws IOException {
int counter, grade, total, average;
// fase de inicialización
total = 0;
counter = 1;
//fase de procesamiento

while (counter <= 10 ) {

        System.out.print ("Teclee calificación de letra:  " );
        System.out.flush ( );
        grade = System.in.read ( );

        if (grade == 'A' )
        total = total + 4;
         else if (grade == 'B' )
        total = total + 3;
        else if (grade == 'C' )
        total = total + 2;
        else if (grade == 'D' )
        total = total + 1;
        else if (grade == 'F' )
        total = total + 0;
        System.in.skip ( 2 );      //Saltar el carácter de nueva línea
        counter = counter + 1;
}
```

```
//fase de terminación
average = total / 10;   //divisió entera
System.out.println ("El promedio del grupo es " + average);
      }
}
```

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;
```

1.16.2 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;
}
```

En el siguiente ejemplo, se usa la instrucción switch para mostrar un mensaje, con base en la nota actual del estudiante:

```
class Class1 {
 public static void main (String args[ ] ) {
  char grade = 'B';
          switch (grade){
case 'A' :System.out.print("Felicidades obtuviste una A");
      break;
case 'B' :System.out.print("Bien, obtuviste una B");
      break;
case 'C' :System.out.print("Suficiente obtuviste una C");
      break;
case 'D' :System.out.print("No hay excusas, estudia más ");
      break;
                              }
      }
}
```

La desventaja de este tipo de pruebas, es que no pueden trabajar con valores de tipos deferentes de int, esto limita a trabajar con if anidado para la utilización de valores grandes como los que maneja (long y float).

También puede utilizar este condicional cuando quiera que más de una opción ejecuten la misma línea de código, esto lo podrá hacer al omitir la sentencia break, lo que le permitirá desplazarse hasta que encuentre el condicional de paro.

```
switch (x){
case2:
case4:
case6:
case8:
system.out.println(“x es un número par.”);
break;
default: system.out.println(“x es cualquier número.”);
}
```

1.16.3 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:

```
for (i = 4001 ; i<=10;i++);
  system.out.println("Hola");
```

En este caso como el primer punto y coma termina el ciclo con enunciado vacío, el ciclo no hace nada en general, la forma correcta para la ejecución de este ciclo sería:

```
for (i = 4001 ; i<=10;i++)
  system.out.println("Hola");
```

Ejemplos:

En el siguiente ejemplo el ciclo for muestra los números desde el 1 hasta el valor contenido en la variable EndCount:

```
public class Class1
{
  public static void main (String[] args)
  {     int count ;
        int end_count = 10;
      for (count = 1; count <=end_count ; count++)
          System.out.print("" + count);
   }
}
```

El programa siguiente efectúa repeticiones de 1 hasta 10, mostrando y sumando cada número a un gran total:

```
public class Class1
{
public static void main (String[] args)
{          int count ;
      int total = 0;
           int end_count = 10;
          for (count = 1; count <=end_count ; count++)
            {
             System.out.println("Sumando " + count + "hasta " + total);
               total = total + count ;
          }
          System.out.println("La suma total es : " + total);
}
}
```

El ciclo for no tiene la limitación de incrementar solo en 1 la variable. El siguiente programa muestra cada quinto número entre uno y cincuenta:

```
public class Class1
{
        public static void main (String[] args)
        {   int count, y ;
            for (count = 0, y = 15; count <=50 ; count += 5, y += 15) {
             System.out.println( " count: " + count);
               }
    }
}
```

Dentro de un ciclo for no es obligatorio que el conteo sea ascendente, el siguiente programa usa el ciclo for para mostrar en forma descendente los números del 1 al 10:

```
public class Class1
{
        public static void main (String[] args)
        {       int count, y ;
            for (count =1 0, y = 15; count <=50 ; count --, y += 15)
         System.out.println( " count: " + count);
         }
}
```

Los ciclos for no están restringidos a usar valores de tipo int en las variables de control de ciclos:

```
public class Class1{
      public static void main (String[] args) {
        int x ;
        char letter;
             float value;
        for (letter = 'A', x = 5;letter <= 'Z' ;letter ++)
           System.out.println(""+ letter );
            for (value = 0 , x = 5; value <1.0; value += 0.1, x  +=20)
           System.out.println(""+ value);
      }
}
```

1.16.4 CICLOS *while* y *do*

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.

1.16.4.1 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.

1.16.4.2 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);
```

El siguiente programa muestra los resultados obtenidos de un examen:

```
import java.io.*;
public class Class1{

public static void main (String args [ ] ) throws IOException
{
int passes = 0, failures = 0, student = 1, result;
while  (student <= 10 ){
System .out.print( "Teclee resultado (1= aprobó, 2= reprobó ): " );
System.out.flush ( );
result = System.in.read ( );

if (result =='1' )
   passes = passes +1;
else
  failures = failures + 1;

student = student + 1;
System.in.skip (2);
}
System.out.println ("Aprobados " + passes );
System.out.println ("Reporbados " + failures );

if ( passes > 8 )
System.out.println ("Aumentar la colegiatutra " );
}
}
```

Programa que calcula el promedio de un grupo.

```
import java.io.*;
```

```
public class Average{
public static void main (String args[ ] ) throws IOException

{
double average;
int counter, grade, total;

// fase de inicialización
total = 0;
counter = 0;

//fase de procesamiento

    System.out.print ("Teclee calificación de letra, Z para terminar:  " );
    System.out.flush ( );
    grade = System.in.read ( );

while (grade != 10 ) {

    if (grade == 'A' )
            total = total + 4;
     else if (grade == 'B' )
            total = total + 3;
    else if (grade == 'C' )
            total = total + 2;
    else if (grade == 'D' )
            total = total + 1;
    else if (grade == 'F' )
            total = total + 0;

    System.in.skip ( 1 );      //Saltar el carácter de nueva línea
    counter = counter + 1;
    System.out.print("Teclee calificación de letra, Z para terminar: " );
    System.out.flush( );
    grade = System.in.read ( );

     }


    //fase de terminación
  if  (counter != 0 ) {
            average = (double) total / counter;
             System.out.println ("El promedio del grupo es " + average);
    }
    else
```

```
            System.out.println( “No se introducieron calificaciones” );
        }
    }
```

1.16.4.3 COMO SALIR DE LOS CICLOS

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.

Ejemplo que muestra el uso de la instrucción **break**:

```
import java.io.*;
public class Class1{
public static void main (String args[ ] ) throws IOException
{
    int count, xpos = 25;
    for (count = 1; count <= 10; count++) {
            if (count ==5)
          break;       // Romper el ciclo sólo si count ==5
          System.out.println(Integer.toString (count ));
        }
System.out.print ("Me salí del ciclo con count = "  + count);
}
}
```

Ejemplo que muestra el uso de la instrucción **continue**:

```
import java.io.*;
public class Class1{
public static void main (String args[ ] ) throws IOException
{
  int xPos = 25;
  for  ( int count = 1; count <= 10; count++ ) {
  if ( count ==5 )
      continue;    //saltarse el resto del código sólo si count==5
System.out.println(Integer.toString ( count ));
}
System.out.println("Usé continue para no I mprimir 5");
```

```
}
}
```

1.17.5 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.

1.16.6 CONTROL GENERAL DEL FLUJO

break [etiqueta]
continue [etiqueta]
return expr;
etiqueta: sentencia;

En caso de que nos encontremos con bucles anidados, se permite el uso de etiquetas para poder salirse de ellos, por ejemplo:

```
uno: for( )
        {  dos: for( )
          {
          continue;         // seguiría en el bucle interno
          continue uno;     // seguiría en el bucle principal
          break uno;        // se saldría del bucle principal
          }
        }
```

Si se declara una función para que devuelva un entero, es imprescindible que se coloque un *return* final para salir de esa función, independientemente de que haya otros en medio del código que también provoquen la salida de la función.

```
int func()
{   if  ( a == 0 )
          return 1;
   return 0;   // es imprescindible porque se retorna un entero
}
```

1.17 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).

1.17.1 ARREGLOS (ARRAYS)

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 sobre cada 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);
}
```

1.17.2 Strings

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.