Manual Básico Java

Recopilación de material - Jesús Arce (jesus.arce@rootssolutions.com.ar) – 27/11/2019

CONTENIDO

Tipos de Datos Primitivos 1

¿Qué son los tipos de datos primitivos en Java? 1

Valores por defecto de los tipos de datos primitivos 2

Tipos de datos Objeto 2

Variables 3

¿Qué son las variables en Java? 3

Tipos de variables en Java 3

Nombres de las variables Java 4

Conceptos básicos de Orientación a Objetos 5

Objeto 5

Clase 6

Interface 8

Paquete 8

Herencia 9

Literales 10

¿Qué son los literales Java? 10

Literales de enteros 11

Literales de decimales 11

Literales de caracteres y cadenas 11

Literales subrayados 12

Expresiones, sentencias y bloques en Java 13
Expresiones 13
Sentencias 13
Bloques 14
Operadores de Asignación y Aritméticos Java 14
Operador de Asignación 14
Operadores Aritméticos 14
Operadores Unarios en Java 15
Operadores unarios suma o resta 15
Operadores de incremento y decremento 16
Operador de complemento lógico 17
Operadores Igualdad y Relacionales en Java 17
Operadores de Igualdad 17
Operadores relacionales 18
Operadores Condicionales 19
Operadores Condicionales 19
Operador Ternario 20
Operador instanceof 20
Sentencias de Control 21
Sentencias de Decisión 21
Sentencias de Bucle 22
Sentencias de ramificación 23
Sentencias Decisión en Java 23
if-else 23
switch 24
Bucles 27
while 27
do-while 28
for 29
Sentencias DE Ramificación en Java 30
break 30
continue 33

Clase String 34
Creando una cadena 34
Crear una cadena vacía 34
Volcando una cadena de texto a la consola 35
Funciones Básicas con Cadenas 35
Información básica de la cadena 35
Comparación de Cadenas 36
Búsqueda de caracteres 37
Búsqueda de subcadenas 37
Métodos con subcadenas 38
Manejo de caracteres 39
Conversión a String: valueOf() 39
Arrays 40
¿Qué es un array en Java? 40
Tamaño del array: .length 41
Matrices o Arrays de varios subindices 41
Incialización de Arrays en Java 42
ArrayList 42

TIPOS DE DATOS PRIMITIVOS

¿Qué son los tipos de datos primitivos en Java?

Como ya hemos comentado Java es un lenguaje de tipado estático. Es decir, se define el tipo de dato de la variable a la hora de definir esta. Es por ello que todas las variables tendrán un tipo de dato asignado.

Es importante saber que estos son tipos de datos del lenguaje y que no representan objetos. Cosa que sí sucede con el resto de elementos del lenguaje Java.

	NOMBRE	TIPO	OCUPA	RANGO (aprox)
TIPOS PRIMITIVOS (sin métodos; no son objetos; no necesitan una invocación para ser creados)	byte	Entero	1 byte	-128 a 127
	short	Entero	2 bytes	-32768 a 32767
	int	Entero	4 bytes	Grande ($2*10^9$)
	long	Entero	8 bytes	Muy grande
	float	Decimal	4 bytes	Grande
	double	Decimal	8 bytes	Muy grande
	char	Carácter simple	2 bytes	---
	boolean	Valor true o false	1 bit	---

byte

Representa un tipo de dato de 8 bits con signo. De tal manera que puede almacenar los valores numéricos de -128 a 127 (ambos inclusive).

short

Representa un tipo de dato de 16 bits con signo. De esta manera almacena valores numéricos de -32.768 a 32.767.

int

Es un tipo de dato de 32 bits con signo para almacenar valores enteros grandes.

long

Es un tipo de dato entero de 64 bits con signo para almacenar valores muy grandes.

float

Es un tipo dato para almacenar números en coma flotante con precisión simple de 32 bits.

double

Es un tipo de dato para almacenar números en coma flotante con doble precisión de 64 bits.

char

Es un tipo de datos que representa a un carácter Unicode sencillo de 16 bits.

boolean

Sirve para definir tipos de datos booleanos. Es decir, aquellos que tienen un valor de true o false. Ocupa 1 bit de información.

Valores por defecto de los tipos de datos primitivos

En el caso de que definamos una variable y no le demos ningún valor, por defecto llevarán los siguientes valores:

Dato Primitivo	Valor por Defecto
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'u0000'
String (o cualquier objeto)	null
boolean	false

TIPOS DE DATOS OBJETO

Hay un tipo de dato String para el manejo de cadenas que no es en sí un tipo de dato primitivo. Con el tipo de dato String podemos manejar cadenas de caracteres separadas por dobles comillas.

El tipo de dato String es un *tipo de dato Objeto*. Así es que existen otros tipos de datos Objeto que están a disposición en el lenguaje para simplificar la manipulación de los datos y trabajar más fácilmente con ellos.

Nota: Los tipos de datos Objeto suelen presentar métodos y atributos.

TIPOS OBJETO (con métodos, necesitan una invocación para ser creados)	Tipos de la biblioteca estándar de Java	- String (cadenas de texto). - Muchos otros (ej: Scanner, TreeSet, ArrayList).
	Tipos definidos por el programador	Cualquiera que se nos ocurra, por ejemplo Persona, Vehiculo, Animal
	Arrays	Serie de elementos o formación tipo vector o matriz. Se considera un tipo objeto especial que carece de métodos.
	Tipos envoltorio o wrapper (Equivalentes a los tipos primitivos pero como objetos.)	Byte
		Short
		Integer
		Long
		Float
		Double
		Character
		Boolean

Este tema se amplía en Literales

VARIABLES

¿Qué son las variables en Java?

Las variables Java son un espacio de memoria en el que guardamos un determinado valor (o dato). Para definir una variable seguiremos la estructura:

```
[privacidad] tipo_variable identificador;
```

Java es un lenguaje de tipado estático. Por lo cual todas las variables tendrán un tipo de dato (ya sea un **tipo de dato primitivo** o una **clase**) y un nombre de identificador.

El tipo de dato se asignará a la hora de definir la variable. Además, en el caso de que las variables sean propiedades de objetos tendrán una privacidad.

Ejemplos de variables serían…

```
int numero = 2;
String cadena = "Hola";
long decimal = 2.4;
boolean flag = true;
```

Las variables son utilizadas como propiedades dentro de los objetos.

```
class Triangulo {
    private long base;
    private long altura;
}
```

No te preocupes por el concepto de objeto, ya que lo revisaremos más adelante cuando hablemos de la Programación Orientada a Objetos

Tipos de variables en Java

Dentro de Java podemos encontrar los siguientes tipos de variables:

- **Variables de instancia (campos no estáticos)**, son las variables que están definidas dentro de un objeto pero que no tienen un modificador de estáticas (static). Suelen llevar un modificador de visibilidad (public, private, protected) definiéndose.

```
class Triangulo {
    private long base;
    private long altura;
}
```

- **Variables de clase (campos estáticos)**, son aquellas variables que están precedidas del modificador static. Esto indica que solo hay una instancia de dicha variable. Es decir, aunque tengamos N objetos de la clase, la variable estática solo se instancia una vez.

```
class Triangulo {
    static long lados = 3;
}
```

Si además queremos que el valor no pueda cambiar nunca la definiremos como final.

```
class Matematicas {
    final static long PI = 3.14159;
}
```

- **Variables locales**, son variables temporales cuyo ámbito de visibilidad es el método sobre el que están definidas. No pueden ser accedidas desde otra parte del código. Se las distingue de las variables de instancia ya que estas no llevan modificadores de visibilidad delante.

```
int variable = 2;
```

- **Parámetros**, son las variables recibidas como parámetros de los métodos. Su visibilidad será el código que contenga dicho método.

```
public Triangulo(long base, long altura){...}
```

Nombres de las variables Java

Cuando vayamos a dar un nombre a una variable deberemos de tener en cuenta una serie de normas. Es decir, no podemos poner el nombre que nos dé la gana a una variable.

Los identificadores son secuencias de texto unicode, sensibles a mayúsculas cuya primer carácter solo puede ser una letra, número, símbolo dolar $ o subrayado. Si bien es verdad que el símbolo dolar no es utilizado por convención.

Es recomendable que los nombres de los identificadores sean legibles y no acrónimos que no podamos leer. De tal manera que a la hora de verlos se auto-documenten por sí mismos. Además, estos identificadores nunca podrán coincidir con las palabras reservadas.

Algunas reglas no escritas, pero que se han asumido por convención son:

- Los identificadores siempre se escriben en minúsculas. (pe. nombre). Y si son dos o más palabras, el inicio de cada siguiente palabra se escriba en mayúsculas (pe. nombrePersona)
- Si el identificador implica que sea una constante. Es decir que hayamos utilizado los modificadores final static, dicho nombre se suele escribir en mayúsculas (pe. LETRA). Y si la constante está compuesta de dos palabras, estas se separan con un subrayado (pe. LETRA_PI).

CONCEPTOS BÁSICOS DE ORIENTACIÓN A OBJETOS

Si estás empezando con el lenguaje Java hay una serie de conceptos básicos de la orientación a objetos que debes de manejar para poder desarrollar con este lenguaje.

Aquí los vamos a ver por encima y dedicaremos un capítulo entero a ellos entrando en detalle sobre todas sus características

- Objeto
- Clase
- Paquete
- Interface
- Herencia

Objeto

Es un elemento de software que intenta representar un objeto del mundo real. De esta forma un objeto tendrá sus propiedades y acciones a realizar con el objeto. Estas propiedades y acciones están encapsuladas dentro del objeto, cumpliendo así los principios de encapsulamiento.

El ***paradigma de la orientación a objetos*** aparece como contraste a la ***programación estructurada*** que se venía utilizando desde los años 60.

Un **objeto** tiene su estado (o estados) y su comportamiento. Esto se modela mediante propiedades (o variables) y métodos. Incluso un objeto puede contener a su vez a otro tipo de objeto.

Encapsulación de datos

Las interacciones con los objetos se hacen mediante los métodos. Es decir, si queremos conocer información del estado del objeto deberemos de llamar a uno de sus métodos y no directamente a las propiedades.

Esta encapsulación nos permitiría el cambiar las propiedades del objeto sin que los consumidores se vean afectados siempre y cuando les sigamos retornando el mismo resultado.

Si bien hay objetos que tienen propiedades públicas, por lo cual podremos acceder directamente a dichas propiedades sin necesidad de utilizar un método.

El uso de objetos nos proporciona los siguientes beneficios:

1. **Modularidad**, el objeto y sus propiedades puede ser pasado por diferentes estructuras del código fuente, pero el objeto es el mismo.
2. **Encapsular Datos**, ocultamos la implementación de propiedades del objeto ya que accederemos a través de los métodos del objeto.
3. **Reutilización de Código**, podemos tener diferentes instancias de un objeto de tal manera que esas diferentes instancias están compartiendo el mismo código.
4. **Reemplazo**, podemos reemplazar un objeto por otro siempre y cuando estos objetos tengan el mismo comportamiento.

Ejemplos de objetos

Cualquier concepto del mundo real se puede modelar como un objeto con su estado y comportamiento. Por ejemplo *un televisor es un objeto**, cuyos estados pueden ser: *encendida*, *apagada*, *en el canal1*, *en el canal2*, *grabando*,… y sus acciones serán *"encender televisor"*, *"apagar televisor"*, *"cambiar de canal"*, *"iniciar la grabación"*,…

Por ejemplo imaginemos una figura geométrica como podría ser un *triángulo*. Un triángulo podemos definirlo por varias propiedades como pueden ser: *base*, *altura*, *el lado* y las *coordenadas x,y del centro del triángulo*. Como métodos de un triángulo podemos *"calcular el área del triángulo"*, *"calcular el perímetro del triángulo"*.

Clase

Las clases representan los prototipos de los objetos que tenemos en el mundo real. Es decir, **es una generalización de un conjunto de objetos**. A su vez **los objetos serán instancias de una determinada clase**.

Si volvemos al ejemplo del televisor, existen múltiples tipos de televisores y cada uno con sus características. Si bien existe un esquema o prototipo que define el televisor. Este prototipo es lo que conocemos la clase.

En la clase es dónde realmente definimos las propiedades y métodos que podrán contener cada una de las instancias de los objetos.

Por ejemplo, para nuestro caso de las figuras geométricas podríamos definir un triángulo de la siguiente forma:

```
class Triangulo {
    private long base;
    private long altura;

    public Triangulo(long base, long altura) {
        this.base = base;
        this.altura = altura;
    }

    public long area() {
```

```
        return (base*altura)/2;
    }
}
```

```
Triangulo t1 = new Triangulo(2.0,3.0);
Triangulo t2 = new Triangulo(4.0,7.0);

t1.area(); // Área 3.0
t2.area(); // Área 14.0
```

De momento no te preocupes por entender el código del todo, pero verás que hemos definido una clase triángulo la cual tiene **dos propiedades base y altura**. Estas propiedades las hemos definido como *"private"* lo cual hace que no puedan ser visibles desde fuera.

```
private long base;
private long altura;
```

Luego tenemos lo que se conoce como un **método constructor**. Es el método que *tiene el mismo nombre que la clase:* *Triangulo ()* y que nos sirve para inicializar las propiedades desde el exterior.

```
public Triangulo(long base, long altura) {
    this.base = base;
    this.altura = altura;
}
```

Además hemos creado un **método que nos calcula el área de un triángulo (base x altura / 2)**. Este método ya es público y podrá ser invocado de forma externa.

```
public long area() {
  return (base*altura)/2;
}
```

Vemos cómo creamos diferentes objetos del tipo Triángulo. A estos objetos los pasamos diferentes valores.

```
Triangulo t1 = new Triangulo(2.0,3.0);
Triangulo t2 = new Triangulo(4.0,7.0);
```

Y por último hemos invocado al método que nos devuelve el área del triángulo del objeto en concreto.

```
t1.area(); // Área 3.0
t2.area(); // Área 14.0
```

Interface

Un **interface** es una forma de establecer un contrato entre dos elementos. Un **interface** indica qué acciones son las que una determinada clase nos va a ofrecer cuando vayamos a utilizarla.

Cuando implementemos un interface (cuando lo usemos) deberemos de implementar todas las acciones (métodos) que este contenga.

Por ejemplo podríamos definir un interface Figura el cual indique qué métodos son obligatorios cuando vayamos a definir una figura. El interface se define mediante la palabra interface.

```
interface Figura {
    ...
}
```

Dentro del interface definimos los métodos que serán obligatorios. Por ejemplo, que de una figura se pueda calcular su área y calcular su perímetro.

```
interface Figura {
    public long area();
    public long perimetro();
}
```

Cuando queramos que una clase implemente un determinado interface deberemos de utilizar el **operador implements** indicando el nombre del interface a implementar.

Así, si un triángulo queremos que implemente el interface Figura lo definiremos de la siguiente forma:

```
public Triangulo implements Figura {
    ...
}
```

En este momento la clase Triangulo deberá de implementar los métodos *calcular área* y *calcular perímetro*.

Paquete

Un **paquete** es una forma de *organizar elementos de software mediante un espacio de nombres*. Así podremos afrontar desarrollos grandes de software facilitando la forma de encontrar o referirnos a un elemento.

Podríamos entender el sistema de paquetes como si fuese un sistema de carpetas. De tal manera que colocaremos cada una de las clases (o ficheros) en un paquete (o directorio).

Los paquetes se definen mediante el modficiador package seguido del nombre del paquete. El paquete lo definiremos en la primera línea de cada una de las clases.

Una definición de paquete podría ser:

```
package net.manualweb.java.ejemplos;
```

El lenguaje **Java nos proporciona un conjunto de paquetes por defecto** (conocido como **API Java**) en los que se pueden encontrar múltiples utilidades del lenguaje. Por ejemplo, la clase Java que nos ayuda a manipular las cadenas de texto es la clase String. La clase String la podemos encontrar en el paquete **java.lang**.

Herencia

La **herencia** es una forma de estructurar el software. Mediante la **herencia** podemos indicar que una clase hereda de otra. Es decir la clase extiende las capacidades (propiedades y métodos) que tenga y añade nuevas propiedades y acciones.

Digamos que las nuevas clases especializan más aquellas clases de las que heredan al añadir nueva funcionalidad. Aunque también pueden reescribir el funcionamiento de dichos elementos.

En nuestro ejemplo del triángulo, este podría heredar de una clase polígono.

```
public class Triangulo extends Poligono {
   ...
}
```

De igual manera de esta clase general Poligono podrían heredar otras clases que representasen un polígono, por ejemplo las clases Cuadrado, Pentagono,...

```
public class Cuadrado extends Poligono {
   ...
}

public class Pentagono extends Poligono {
   ...
}
```

La herencia entre clases se indica mediante el operador extends.

La clase superior de la que heredan las figuras puede definir una serie de propiedades y métodos que heredarán las clases hijas y que por ende podrán utilizar.

Por ejemplo, la clase Poligono puede tener una propiedad que sean las longitudes de los lados del polígono y que utilice esas longitudes para calcular el perímetro del polígono.

```
public class Poligono {

    private long[] lados;

    public Poligono(long[] lados) {
        this.lados = lados;
    }

    public long perimetro() {
        ...
    }
}
```

Cuando ahora indiquemos que la clase Triangulo hereda de la clase Poligono.

```
public class Triangulo extends Poligono {
    ...

    public Triangulo (long base, long altura, int[] lados) {
        super(lados);
        this.base = base;
        this.altura = altura;
    }
}
```

Veremos que los objetos instanciados como triángulos tendrán acceso a los métodos del polígono.

```
Triangulo t1 = new Triangulo(2.0,3.0);
t1.perimetro();
```

En este caso accedemos al método perímetro que heredamos de la clase Polígono.

Una de las cosas que tienes que saber en la herencia es que en el constructor de la clase que hereda (o clase hija) se deberá de llamar al constructor de la clase padre. Para ello se utiliza el método especial super();

LITERALES

¿Qué son los literales Java?

Los valores literales son aquellos que podemos asignar a las variables. Dependiendo del tipo de variable podremos asignar unos valores u otros.

Literales de enteros

Los enteros que podemos utilizar serán **byte**, **short**, **int** y **long**. Los literales que les asignemos siempre será un número entero.

```
byte variableByte = 12;
short variableShort = 12;
int variableInt = 12;
long variableLong = 12;
```

Si bien para el caso del tipo **long** podemos crear literales de enteros que acaben en L (mayúscula o minúscula, aunque por legilibilidad se recomienda la primera)

```
long variableLong = 12D;
```

Hay otros valores que pueden ser manejados por los literales enteros, para cuando representemos el número en diferentes bases. Por ejemplo, cuando los manejemos como binarios o hexadecimales. Para este caso habrá que manejar literales de enteros que tengan dicho formato.

```
int variableBinaria = 011010;
int variableHexadecimal = 0x1a;
```

Literales de decimales

Los dos tipos de datos de decimales que podemos manejar son **float** y **double**. Para estos casos, la representación del literal de decimales será con separación de un punto entre la parte entera y la parte decimal.

```
float variableFloat = 12.2;
double variableDouble = 12.2;
```

De igual manera podemos utilizar las letras F o f para el tipo de dato **float** y D o d para el tipo de dato **double**. Siempre, por legilibilidad se recomienda la letra en mayúsculas.

```
float variableFloat = 12.2F;
double variableDouble = 12.2D;
```

Literales de caracteres y cadenas

Tanto los caracteres del tipo de dato **char**, como las cadenas del tipo de datos **String** contienen caracteres Unicode UTF-16.

Los caracteres UTF-16 se pueden escribir directamente en la cadena o si nuestro editor de textos no nos permite el manejo de esa codificación los podemos poner escapados en el formato.

'uCODIGOUNICODE'

Por ejemplo, la letra como la ñ se escaparía de la siguiente forma:

'u00F1'

Para utilizarla en una cadena de texto “España” podríamos poner

```
String pais = "Espau00F1a";
```

Para los caracteres utilizaremos comillas simples para delimitarlos, mientras que para las cadenas utilizaremos comillas dobles.

```
char variableChar = ‘a’;
String variableString = “cadena”;
```

Además en las cadenas podemos utilizar una serie de secuencias de escape, las cuales empiezan por una barra invertida y siguen con un modificador:

Secuencia	Significado
b	retroceso
t	tabular la cadena
n	salto de línea
f	form feed
r	retorno de carro
'	comilla simple
"	comilla doble
\	barra invertida

Literales subrayados

A partir de la versión 1.7 de Java se puede utilizar el subrayado para realizar separaciones entre números para una mejor visualización.

A todos los efectos el valor del número es como si no existiese el carácter de subrayado.

```
long tarjetaCredito = 1234_5678_9012_3456L;
long mascaraBinaria = 0b11010010_01101001_10010100_10010010;
```

No podremos utilizar el literal de subrayado al principio o final del número, alrededor de un punto decimal, ni entre el número y un literal de entero o decimal (D, F o L).

EXPRESIONES, SENTENCIAS Y BLOQUES EN JAVA

Un programa en Java se compone de un conjunto de sentencias que se ejecutan para resolver un problema. Las sentencias son el elemento básico de ejecución de los programas Java.

A parte de las sentencias, en un programa Java nos encontraremos con expresiones y bloques.

Expresiones

Una expresión es un conjunto de variables, operadores e invocaciones de métodos que se construyen para poder ser evaluadas retornando un resultado.

Ejemplos de expresiones son:

```
int valor = 1;
if (valor 1 > valor2) { ... }
```

Cuando tengamos expresiones de evaluación complejas es recomendable que utilicemos paréntesis para saber cuál es el orden de ejecución de operaciones.

Ya que si tenemos una expresión como:

```
2 + 10 / 5
```

No será la misma si ponemos:

```
(2 + 10) / 5
```

O algo como esto:

```
2 + (10 / 5)
```

En el caso de no utilizar paréntesis se ejecutará el orden de preferencia de operadores. En este caso la división tiene más preferencia que la suma.

Sentencias

Una sentencia es la unidad mínima de ejecución de un programa. Un programa se compone de conjunto de sentencias que acaban resolviendo un problema. **Al final de cada una de las sentencias encontraremos un punto y coma (;)**.

Bloques

Un bloque es un conjunto de sentencias los cuales están delimitados por llaves.

```
if (expresion) {
   // Bloque 1
} else {
   // Bloque 2
}
```

OPERADORES DE ASIGNACIÓN Y ARITMÉTICOS JAVA

Operador de Asignación

El operador Java más sencillo es el **operador de asignación**. Mediante este operador se asigna un valor a una variable. El operador de asignación es el símbolo igual.

La estructura del operador de asignación es:

```
variable = valor;
```

Así podemos asignar valores a variables de tipo entero, cadena,...

```
int numero = 3;
String cadena = "Hola Mundo";
double decimal = 4.5;
boolean verdad = true;
```

Operadores Aritméticos

Los operadores aritméticos en Java son los operadores que nos permiten realizar operaciones matemáticas: *suma, resta, multiplicación, división y resto.*

Los operadores aritméticos en Java son:

Operador	Descripción
+	Operador de Suma. Concatena cadenas para la suma de String
-	Operador de Resta
*	Operador de Multiplicación
/	Operador de División
%	Operador de Resto

Los operadores aritméticos en Java los utilizaremos entre dos literales o variables y el resultado, normalmente lo asignaremos a una variable o bien lo evaluamos.

```
variable = (valor1|variable1) operador (valor2|variable2);
```

Así podemos tener los siguientes usos en el caso de que queramos asignar su valor.

```
suma = 3 + 7;          // Retorna 10
resta = 5 - 2;         // Retorna 3
multiplicacion = 3 * 2;  // Retorna 6
division = 4 / 2;      // Retorna 2
resto = 5 % 3;         // Retorna 2
```

Ten en cuenta que pueden ser valores o variables:

```
suma = vble1 + 3;  // Sumamos 3 al valor de la variable vble1
resta = vble1 - 4;  // Restamos 4 al valor de la variable vble1
...
```

O podríamos utilizarlo en una condición

```
if (variable > suma + 3) { ... }
```

En este caso no asignamos el resultado de la suma a una variable, solo lo evaluamos.

OPERADORES UNARIOS EN JAVA

Los operadores unarios en Java son aquellos que solo requieren un operando para funcionar.

Los **operadores** unitarios que tenemos en Java son:

Operador	Descripción
+	Operador unario suma. Indica un número positivo.
-	Operador unario resta. Niega una expresión.
++	Operador de incremento. Incrementa el valor en 1.
–	Operador de decremento. Decrementa el valor en 1.
!	Operador de complemento lógico. Invierte el valor de un booleano

Operadores unarios suma o resta

Los operadores unitarios de suma o resta son muy sencillos de utilizar. En el caso del operador unitario suma su uso es redundante. Con el operador unitario resta podemos invertir un valor.

Por ejemplo podríamos tener el siguiente código:

```
int valor = 2;
```

```
System.out.println(-valor); // Imprimirá por pantalla un -2
```

Operadores de incremento y decremento

Los operadores de incremento se pueden aplicar como prefijo o como sufijo.

```
++ variable;
variable ++;
-- variable;
variable --;
```

En todos los casos el valor de la variable acabará con una unidad más (para el operador de incremento) o con una unidad menos (para el operador de decremento).

Si bien si están participando en una asignación hay que tener cuidado en si utilizamos el operador como prefijo o como sufijo.

En el caso de utilizarlo como prefijo el valor de asignación será el valor del operando más el incremento de la unidad. Y si lo utilizamos como sufijo se asignará el valor del operador y luego se incrementará la unidad sobre el operando.

Es más sencillo verlo en código:

```
suma = ++vble1;
```

Sería lo mismo que poner

```
vble1 = vble1 + 1;
suma = vble1;
```

Mientras que si escribimos:

```
suma = vble1++;
```

Sería lo mismo que poner:

```
suma = vble1;
vble1 = vble1 + 1;
```

Exactamente lo mismo le sucede al operador de decremento, pero restando una unidad.

Operador de complemento lógico

El operador de complemento lógico sirve para negar un valor lógico. Se suele utilizar delante de una operación de evaluación booleana. Normalmente en sentencias de decisión o bucles.

La estructura es:

```
! (expresion)
```

Si la expresión era un **true** la convierte en **false** y si era **false** la convierte en **true**.

Podemos verlo en el siguiente ejemplo:

```
int vble1 = 2;
int vble2 = 3;

if !(vble1 > vble2)
   System.out.println("variable 1 es más pequeña que la variable 2");
```

Como podemos observar el valor de la expresión evaluada es convertido.

OPERADORES IGUALDAD Y RELACIONALES EN JAVA

Los operadores de igualdad y relacionales en Java son aquellos que nos permiten comparar el contenido de una variable contra otra atendiendo a si son variables con un valor igual o distinto o bien si los valores son mayores o menores.

El listado de operadores de igualdad y relacionales en Java es:

Operador	Descripción
==	igual a
!=	no igual a
>	mayor que
>=	mayor o igual que
<	menor que
<=	menor o igual que

Operadores de Igualdad

Mediante los operadores de igualdad podemos comprobar si dos valores son iguales **(operador ==)** o diferentes **(operador !=)**.

La estructura de los **operadores de igualdad** es la siguiente:

```
vble1 == vble2
vble1 != vble2
```

Podemos utilizar estos operadores de igualdad en Java de la siguiente forma:

```
int vble1 = 5;
int vble2 = 3;

if (vble1 == vble2)
  System.out.println("Las variables son iguales");

if (vble1 != vble2)
  System.out.println("Las variables son distintas");
```

Operadores relacionales

Permiten comprobar si un valor es mayor que **(operador >)**, menor que **(operador <)**, mayor o igual que **(>=)** y menor o igual que **(<=)**.

Al final el operador lo valida entre dos valores o variables con la estructura:

vble1 > vble2
vble1 < vble2
vble1 >= vble2
vble1 <= vble2

De esta forma podemos tener un código fuente que nos ayude a realizar estas validaciones de relación:

```
int vble1 = 5;
int vble2 = 3;

if (vble1 > vble2)
  System.out.println("La variable 1 es mayor que la variable 2");

if (vble1 < vble2)
  System.out.println("La variable 1 es menor que la variable 2");

if (vble1 >= vble2)
  System.out.println("La variable 1 es mayor o igual que la variable 2");

if (vble1 <= vble2)
  System.out.println("La variable 1 es menor o igual que la variable 2");
```

OPERADORES CONDICIONALES

Los operadores condicionales en Java son aquellos que evalúan dos expresiones booleanas.

Dentro de los operadores condicionales en Java tenemos:

Operador	Descripción
&&	Operador condicional AND
||	Operador condicional OR
?:	Operador Ternario
instanceof	Operador instanceof

Operadores Condicionales

La estructura de los operadores condicionales en Java es:

```
(expresion_booleana1 && expresion_booleana2)
(expresion_booleana1 || expresion_booleana2)
```

En el caso del **operador condicional AND** el resultado será **true** siempre y cuando las dos expresiones evaluadas sean **true**. Si una de las expresiones es **false** el resultado de la expresión condicional AND será **false**.

Para el **operador condicional OR** el resultado será **true** siempre que alguna de las dos expresiones sea **true**.

Los operadores booleanos funcionan mediante la *evaluación por cortocircuito*. Es decir, que dependiendo del valor de la expresión 1 puede que no sea necesario evaluar la expresión 2.

Para el caso del **operador condicional AND**, si la primera expresión es **false** ya devuelve **false** sin evaluar la segunda expresión. Y en el caso del **operador condicional OR** si la primera expresión es **true** ya devuelve **true** sin evaluar la segunda expresión.

Podríamos ver el uso de los operadores condicionales en el siguiente código:

```
int vble1 = 5;
int vble2 = 3;

if ((vble1 == 5) && (vble2 ==3))
  System.out.println("Las dos variables mantienen sus valores iniciales");

if ((vble1 == 5) || (vble2 ==3))
  System.out.println("Al menos una variable mantiene su valor inicial");
```

Operador Ternario

El **operador ternario** es otro de los operadores condicionales. Es una forma reducida de escribir un **if-else**. El **operador ternario** es representado mediante el símbolo **?:**

La estructura del operador ternario es:

```
(expresion)?valor_true:valor_false;
```

En el caso de que la expresión tenga un valor de **true** se retorna el valor indicado después del cierre de interrogación (?) Y si la expresión tiene un valor de **false** se retorna el valor que esté después de los dos puntos (:).

El **operador ternario** se suele utilizar para decidir que valor asignar. Un ejemplo de código del operador ternario sería:

```
int vble1 = 5;
int vble2 = 4;
int mayor;

mayor = (vble1 > vble2)?vble1:vble2;

System.out.println("El mayor de los dos números es " + mayor);
```

Vemos que si la variable 1 es mayor que la variable 2 guardaremos el valor de la variable 1 en la variable mayor. En caso contrario se guardaría el valor de la variable 2, ya que en ese caso sería la mayor.

Operador instanceof

El **operador instanceof** es un operador especial para los objetos. Mediante el **operador instanceof** podemos comprobar si un objeto es de una clase concreta.

La estructura del **operador instanceof** es:

```
objeto instanceof clase
```

El operador instanceof devolverá true siempre y cuando el objeto sea del tipo clase o de alguna de las clases de las que herede.

Así podríamos definir una secuencia de clases:

```
class Poligono {}
interface Figura {}
class Triangulo extends Poligono implements Figura {}
```

Ahora definimos un par de objetos:

```
Poligono p = new Poligono();
Triangulo t = new Triangulo();
```

Podemos, mediante el uso del **operador instanceof**, comprobar que t es instancia de tipo Triangulo, Poligono y Figura. Mientras que p es instancia de tipo Polígono, pero no de Triangulo, ni Figura.

```
System.out.println("p es instancia de ");
if (p instanceof Poligono)
  System.out.println("Poligono");
if (p instanceof Triangulo)
   System.out.println("Triangulo");
if (p instanceof Figura)
   System.out.println("Figura");

System.out.println("t es instancia de ");
if (t instanceof Poligono)
   System.out.println("Poligono");
if (t instanceof Triangulo)
   System.out.println("Triangulo");
if (t instanceof Figura)
   System.out.println("Figura");
```

SENTENCIAS DE CONTROL

Un programa en Java se ejecuta en orden desde la primera sentencia hasta la última.

Si bien existen las **sentencias de control de flujo** las cuales permiten alterar el fujo de ejecución para tomar decisiones o repetir sentencias.

Dentro de las **sentencias de control de flujo** tenemos las siguientes:

- Sentencias de decisión
- Sentencias de bucle
- Sentencias de ramificación

Sentencias de Decisión

Son sentencias que nos permiten tomar una decisión para poder ejecutar un bloque de sentencias u otro.

Las sentencias de decisión son: if-else y switch.

Mediante if-else podremos evaluar una decisión y elegir por un bloque u otro.

```
if (expresion) {
  // Bloque then
} else {
  // Bloque else
}
```

Mientras que con switch podremos evaluar múltiples decisiones y ejecutar un bloque asociado a cada una de ellas.

```
switch (expresion) {
  case valor1:
    bloque1;
    break;
  case valor2:
    bloque2;
    break;
  case valor3:
    bloque3;
    break;
  ...
  default:
     bloque_por_defecto;
}
```

Sentencias de Bucle

Las **sentencias de bucle** nos van a permitir ejecutar un bloque de sentencias tantas veces como queramos, o tantas veces como se cumpla una condición.

En el momento que se cumpla esta condición será cuando salgamos del bucle.

Las sentencias de bucle son: while, do-while y for.

En el caso de la sentencia while tenemos un bucle que se ejecuta mientas se cumple la condición, pero puede que no se llegue a ejecutar nunca, si no se cumple la condición la primera vez.

```
while (expresion) {
  bloque_sentencias;
}
```

Por otro lado, si utilizamos do-while, lo que vamos a conseguir es que el bloque de sentencias se ejecute, al menos, una vez.

```
do {
  bloque_sentencias;
} while (expresion)
```

La sentencia for nos permite escribir toda la estructura del bucle de una forma más acotada. Si bien, su cometido es el mismo.

```
for (sentencias_inicio;expresion;incremento) {
  bloque_sentencias;
}
```

Sentencias de ramificación

Las **sentencias de ramificación** son aquellas que nos permiten romper con la ejecución lineal de un programa.

El programa se va ejecutando de forma lineal, sentencia a sentencia. Si queremos romper esta linealidad tenemos las **sentencias de ramificación**.

Las **sentencias de ramificación** son: break y continue.

En el caso de break nos sirve para salir de bloque de sentencias, mientras que continue sirve para ir directamente al siguiente bloque.

SENTENCIAS DECISIÓN EN JAVA

Las **sentencias de decisión** son sentencias que nos permiten tomar una decisión para poder ejecutar un bloque de sentencias u otro.

Las **sentencias de decisión** son: if-else y switch.

if-else

La estructura de las sentencias if-else es:

```
if (expresion) {
  // Bloque then
} else {
  // Bloque else
}
```

Se evalua la expresión indicada en la sentencia if. En el caso de que la expresión sea true se ejecutará el bloque de sentencias then y en el caso de que la expresión sea false se ejecutará el bloque de sentencias else.

La parte del else no tiene por qué existir. En este caso tendríamos una sentencia if.

```
if (expresion) {
  // Bloque then
}
```

De esta forma podríamos tener el siguiente código fuente:

```
int valor = 4;

if (valor < 10) {
  System.out.println("El número es menor de 10");
} else {
  System.out.println("El número es mayor de 10");
}
```

Las sentencias if--else pueden estar anidadas y así nos encontraríamos con una sentencia **if-elseif**, la cual tendría la siguiente estructura:

```
if (expresion) {
  // Bloque then
} else if {
  // Bloque else
} else if {
  // Bloque else
} else if {
  // Bloque else
} ...
```

De esta forma podemos tener el siguiente código:

```
int valor = 14;

if (valor < 10) {
  System.out.println("El valor es una unidad");
} else if (valor < 100) {
  System.out.println("El valor es una decena");
} else if (valor < 1000) {
  System.out.println("El valor es una centena");
} else if (valor < 10000) {
  System.out.println("El valor es un millar");
} else {
  System.out.println("Es un número grande");
}
```

switch

Para los casos en los que se tienen muchas ramas o caminos de ejecución en una sentencia if tenemos la sentencia switch. La sentencia switch evalúa una expresión y ejecutará el bloque de sentencias que coincida con el valor de la expresión.

El valor de la expresión tiene que ser numérico. Aunque a partir de *Java SE 7* ya se pueden utilizar expresiones cuya evaluación sean cadenas.

La estructura de la sentencia switch es:

```
switch (expresion) {
  case valor1:
    bloque1;
    break;
  case valor2:
    bloque2;
    break;
  case valor3:
    bloque3;
    break;
  ...
  default:
     bloque_por_defecto;
}
```

Es importante ver que se utiliza la sentencia **break**. La sentencia break hace que se salga de la sentencia switch y por lo tanto no se evalúe el resto de sentencias. Por lo tanto, su uso es obligatorio al final de cada uno de los bloques.

Un ejemplo claro en el que podemos utilizar la sentencia switch es para evaluar el valor de un mes en numérico y convertirlo a cadena. Este código quedaría de la siguiente forma:

```
int iMes = 3;
String sMes;

switch (iMes) {
   case 1:
      sMes = "Enero";
      break;
   case 2:
      sMes = "Febrero";
      break;
   case 3:
      sMes = "Marzo";
      break;
   case 4:
      sMes = "Abril";
      break;
   case 5:
      sMes = "Mayo";
      break;
   case 6:
      sMes = "Junio";
      break;
   case 7:
      sMes = "Julio";
      break;
   case 8:
```

```
        sMes = "Agosto";
        break;
    case 9:
        sMes = "Septiembre";
        break;
    case 10:
        sMes = "Octubre";
        break;
    case 11:
        sMes = "Noviembre";
        break;
    case 12:
        sMes = "Diciembre";
        break;
    default:
        sMes = "Mes incorrecto";
}

System.out.println(sMes);
```

Este mismo modelo lo podríamos haber implementado mediante una estructura if-else. Si bien, como podemos ver en el código queda más complejo

```
if (iMes == 1){
    sMes = "Enero";
} else if (iMes == 2) {
    sMes = "Febrero";
} else if (iMes == 3) {
    sMes = "Marzo";
} else if (iMes == 4) {
    sMes = "Abril";
} else if (iMes == 5) {
    sMes = "Mayo";
} else if (iMes == 6) {
    sMes = "Junio";
} else if (iMes == 7) {
    sMes = "Julio";
} else if (iMes == 8) {
    sMes = "Agosto";
} else if (iMes == 9) {
    sMes = "Septiembre";
} else if (iMes == 10) {
    sMes = "Octubre";
} else if (iMes == 11) {
    sMes = "Noviembre";
} else if (iMes == 12) {
    sMes = "Diciembre";
} else {
    sMes = "Mes incorrecto";
}

System.out.println(sMes);
```

Otra cosa que tenemos que saber de la sentencia switch es que las evaluaciones case pueden ser múltiples. La estructura en este caso sería:

```
switch (expresion) {
  case valor1: case valor2: case valor3:
    bloque1;
    break;
  case valor4: case valor5: case valor6:
    bloque2;
    break;
  ...
  default:
    bloque_por_defecto;
}
```

Esto podemos utilizarlo para saber los días del mes. El código sería el siguiente:

```
int iMes = 3;
String sDias;

switch (iMes) {
    case 1: case 3: case 5: case 7: case 8: case 10: case 12:
        sDias = "El mes tiene 31 días";
        break;
    case 4: case 6: case 9: case 11:
        sDias = "El mes tiene 30 días";
        break;
    case 2:
        sDias = "El mes tiene 28 días (o 29 días si es año bisiesto)";
        break;
    default:
        sDias = "Mes incorrecto";
}
```

Como vemos tenemos diferentes evaluaciones con la sentencia case.

BUCLES

Las **sentencias de bucle** nos van a permitir ejecutar un bloque de sentencias tantas veces como queramos, o tantas veces como se cumpla una condición.

Las **sentencias de bucle** en son: while, do-while y for.

while

La estructura repetitiva while realiza una primera evaluación antes de ejecutar el bloque. Si la expresión es true pasa a ejecutar de forma repetida el bloque de sentencias.

Cada vez que termina de ejecutar el bloque de sentencias vuelve a evaluar la expresión. Si la expresión sigue siendo true vuelve a ejecutar el bloque. En el caso de que la expresión sea false se saldrá del bucle.

Es por ello que dentro del bloque de sentencias *deberán de existir sentencias que modifiquen la evaluación de la expresión, ya que de no hacerse se podría entrar en un bucle infinito.*

La estructura de la sentencia while es la siguiente:

```
while (expresion) {
  bloque_sentencias;
}
```

Los casos de uso de una sentencia repetitiva while son variados, pero principalmente se utiliza para recorrer estructuras de datos o tener contadores.

Por ejemplo podemos realizar un contador de 1 a 10 de la siguiente forma:

```
int contador = 1;
while (contador <= 10) {
  System.out.println(contador);
  contador++;
}
```

do-while

En el caso de la estructura repetitiva do-while el funcionamiento es el mismo que el de while. Pero con una diferencia, primero se ejecuta el bloque de sentencias y luego se evalua la expresión. Por lo tanto siempre se ejecutará, al menos una vez, el bloque de sentencias.

La estructura de la sentencia do-while es:

```
do {
  bloque_sentencias;
} while (expresion)
```

Al igual que anteriormente, en el bloque de sentencias deberemos de modificar alguna de las condiciones de la expresión para poder salir del bucle.

Un ejemplo claro del bucle do-while sería el ejemplo en el que le pedimos al usuario que introduzca números por teclado, los cuales mostraremos en forma de eco por pantalla, hasta que introduzca el cero. En ese caso saldremos del bucle.

Utilizaremos la estructura do-while en vez de la while ya que al menos vamos a pedirle al usuario un número.

El código sería el siguiente:

```
Scanner reader = new Scanner(System.in);
int iNumero;

do {
   System.out.println("Introduce carácter por consola");
   iNumero = reader.nextInt();
   System.out.println(iNumero);
} while (iNumero <> 0);
```

En el caso de haberlo realizado con un bucle while tendríamos que repetir la captura y salida de datos. Veamos cómo quedaría para que puedas ver las diferencias.

```
Scanner reader = new Scanner(System.in);
int iNumero;

System.out.println("Introduce carácter por consola");
iNumero = reader.nextInt();
System.out.println(iNumero);

while (iNumero <> 0) {
  System.out.println("Introduce carácter por consola");
  iNumero = reader.nextInt();
  System.out.println(iNumero);
}
```

for

Otra de las sentencias repetitivas que tenemos, a parte de los bucles while y do-while, es la sentencia for.

La sentencia for tiene la característica de que tiene bien definido el inicio del bloque, la evaluación de la expresión, el incremento de valor y el bloque de sentencias.

La estructura del bucle for es:

```
for (sentencias_inicio;expresion;incremento) {
  bloque_sentencias;
}
```

Tanto las sentencias_inicio, expresión como incremento son opcionales y pueden estar o no. Aunque normalmente aparecerán en la estructura.

Esta estructura la podríamos reproducir mediante una sentencia while de la siguiente forma:

```
sentencias_inicio;
```

```
while (expresion) {
  bloque_sentencias;
  incremento;
}
```

Las funcionalidades en las que utilizaremos la sentencia for serán las mismas que las sentencias while y do-while, que serán contadores, recorrer estructuras,…

Si queremos definir un contador de 1 a 10 mediante una sentencia for utilizaremos el siguiente código:

```
for (int x=1;x<=10;x++=) {
  System.out.println("Valor del contador: " + x);
}
```

En pantalla obtendremos el siguiente resultado:

```
Valor del contador: 1
Valor del contador: 2
Valor del contador: 3
Valor del contador: 4
Valor del contador: 5
Valor del contador: 6
Valor del contador: 7
Valor del contador: 8
Valor del contador: 9
Valor del contador: 10
```

SENTENCIAS DE RAMIFICACIÓN EN JAVA

Las **sentencias de ramificación** son aquellas que nos permiten romper con la ejecución lineal de un programa.

break

Ya vimos que en la sentencia selectiva switch se utilizaba la sentencia break para salir de las evaluaciones y así solo ejecutar el bloque de la opción correspondiente. Si bien podemos utilizar la sentencia break con las sentencias repetitivas while, do-while y for. Esta es la que se conoce como sentencia **break sin etiquetar**.

Cuando utilicemos el break dentro de uno de estos bucles lo que se conseguirá es salirse de la ejecución del bucle hasta el siguiente bloque de sentencias. Mismo efecto que si la expresión de evaluación hubiese dado false.

Así podremos encontrarnos códigos como el siguiente:

```
while (expresion) {
```

```
  sentencia(s);
  break;
  sentencias(s);
}
```

Al ejecutar la sentencia break ya no ejecutaremos las sentencias que vayan después.

El uso del break dentro de estructuras repetitivas suele aparecer cuando estamos realizando la búsqueda de un elemento por una estructura de datos y lo hemos encontrado.

Por ejemplo, si tenemos un array y queremos buscar un número dentro del array podríamos tener el siguiente código:

```
int[] numeros = {12,3,4,5,6,7,9,10};
int posicion = 0;
boolean encontrado = false;

while (posicion < numeros.length) {
  if (numeros[posicion] == 5) {
       encontrado = true;
       break;
  }
  posicion++;
}

if (encontrado) {
    System.out.println("El número está en la posición: " + posicion);
} else {
    System.out.println("Número no encontrado");
}
```

Las sentencias break se pueden cambiar por *variables bandera*. Estas *variables bandera* actúan como cortocircuitos de las expresiones de validación y hacen que salgamos de los bucles.

En este caso podríamos haber utilizado la variable “encontrado” como *variable bandera*. Y podríamos reescribir el código de la siguiente forma:

```
int[] numeros = {12,3,4,5,6,7,9,10};
int posicion = -1;
boolean encontrado = false;

while ((!encontrado) && (posicion&lt;numeros.length)) {
  posicion++;
  if (numeros[posicion] == 5) {
       encontrado = true;
  }
}
```

```
if (encontrado) {
   System.out.println("El número está en la posición: " + posicion);
} else {
   System.out.println("Número no encontrado");
}
```

Como podéis ver el código es muy parecido y solo aparece la condición de la *variable bandera*.

Una de las cosas que tenemos que tener en cuenta a la hora de utilizar las sentencias **break sin etiquetar** es que estas generan que se rompa la secuencia de ejecución de sentencias hasta el primer bloque anidado.

Pero, ¿qué sucedería si queremos salir de un conjunto de bucles anidados? Aunque podríamos utilizar múltiples break existe la posibilidad de utilizar sentencias **break etiquetadas**.

Las sentencias **break etiquetadas** funcionan igual que las break pero al ejecutarse se salen a la siguiente sentencia después del bloque etiquetado.

La sintaxis es:

```
break nombre_etiqueta;
```

Veamos cómo podría ser una estructura de uso de las sentencias **break etiquetadas**.

```
sentencia(s) iniciales;

etiqueta:
  while (expresion) {
    sentencia(s) bloque1;
    while (expresion) {
      sentencia(s) bloque2;
      break etiqueta;
    }
  }

sentencias(s) finales;
```

Al ejecutarse se sale de todo el bloque etiquetado como etiqueta y ejecuta las sentencias finales.

Esto podemos encontrarlo si estamos recorriendo una matriz para buscar un elemento. Ya que para recorrer una matriz vamos a necesitar dos bucles anidados.

```
int[][] matriz = {
  {1,2,3,4},
  {5,6,7,8},
  {9,10,11,12}
```

```
};
int numeroBuscado = 5;

busqueda:
for (int x=0; x &lt; matriz.length; x++) {
  for (int y=0; y &lt; matriz[x].length; y++) {
    if (matriz[x][y] = numeroBuscado) {
      encontrado = true;
      break busqueda;
    }
  }
}

if (encontrado) {
  System.out.println(x + "," + y);
} else {
  System.out.println("No encontrado");
}
```

continue

Otra sentencia que podemos utilizar en los bucles es la sentencia continue. A ejecutar una La sentencia continue dejaremos de ejecutar las sentencias que quedan para acabar el bloque dentro de un bucle para volver a evaluar una expresión.

La estructura de una sentencia continue sería:

```
while (expresion) {
  sentencia(s) iniciales;
  continue;
  sentencias(s) finales;
}
```

Al ejecutarse la sentencia continue nunca se ejecutarán las sentencias finales.

De igual manera que sucedía con la sentencia break, podemos realizar **continue etiquetados**. En este caso la sentencia continue nos llevará directamente a la primera condición de evaluación del bloque.

La estructura en este caso sería la siguiente:

```
etiqueta:
  while (expresion) {
    sentencia(s) iniciales;
    while (expresion) {
      sentencia(s) iniciales;
      continue etiqueta;
      sentencia(s) finales;
    }
```

```
  sentencia(s) finales;
}
```

CLASE STRING

Una cadena de texto no deja de ser más que la sucesión de un conjunto de caracteres alfanuméricos, signos de puntuación y espacios en blanco con más o menos sentido.

Podemos encontrarnos desde la archiconocida cadena “Hola Mundo” y la no menos “Mi primera cadena de texto”, pasando por las cadenas de texto personalizadas “Víctor”, “Víctor Cuervo”, las cadenas de depuración “¿Aquí?”, “Paso 1”, “Paso 2”,… hasta las inclasificables “asdf”.

Todas ellas serán representadas en java con la clase String y StringBuffer. Aunque de momento nos centraremos en la primera.

Para encontrar la clase String dentro de las librerías de Java tendremos que ir a java.lang.String.

Creando una cadena

Para crear una cadena tenemos dos opciones:

- Instanciamos la clase String. Que sería una creación explicita de la clase

```
String sMiCadena = new String("Cadena de Texto");
```

- Crear implícitamente la cadena de texto. Es decir, simplemente le asignamos el valor al objeto.

```
String sMiCadena = "Cadena de Texto";
```

En este caso, Java, creará un objeto String para tratar esta cadena.

Crear una cadena vacía

Podemos tener la necesidad de crear una cadena vacía. Puede darse el caso que no siempre sepamos lo que vamos a poner de antemano en la cadena de texto. ¿A quién no le surgen dudas? ;-) … Fuera de bromas, muchas veces la cadena de texto nos la proporcionará el usuario, otro sistema...

Para poder crear la cadena vacía nos bastará con asignarle el valor de “”, o bien, utilizar el constructor vacío.

```
String sMiCadena = "";
String sMiCadena = new String();
```

Volcando una cadena de texto a la consola

Solo nos quedará saber cómo volcar una cadena por pantalla. Esto lo haremos con la clase System.out.println que recibirá como parámetro el objeto String.

Por ejemplo:

```
System.out.println("Mi Cadena de Texto");
```

ó

```
String sMiCadena = new String("Mi Cadena de Texto");
System.out.println(sMiCadena);
```

FUNCIONES BÁSICAS CON CADENAS

Una vez que hemos visto lo sencillo que es crear una cadena de texto vamos a echar un vistazo a los métodos que nos permiten manipular la cadena de texto. Si tuviésemos que ordenar dichos métodos podríamos llegar a la siguiente división:

- Información básica de la cadena
- Comparación de Cadenas
- Búsqueda de caracteres
- Búsqueda de subcadenas
- Manejo de subcadenas
- Manejo de caracteres
- Conversión a String: valueOf()

Información básica de la cadena

.length() Nos devuelve el tamaño que tiene la cadena.

char charAt(int index) Devuelve el carácter indicado como índice. El primer carácter de la cadena será el del índice 0. Junto con el método .length() podemos recuperar todos los caracteres de la cadena de texto. Hay que tener cuidado. Ya que si intentamos acceder a un índice de carácter que no existe nos devolverá una excepción IndexOutOfBoundsException.

Comparación de Cadenas

Los métodos de comparación nos sirven para comparar si dos cadenas de texto son iguales o no. Dentro de los métodos de comparación tenemos los siguientes:

boolean equals(Object anObject) Nos permite comparar si dos cadenas de texto son iguales. En el caso de que sean iguales devolverá como valor "true". En caso contrario devolverá "false". Este método tiene en cuenta si los caracteres van en mayúsculas o en minúsculas. Si queremos omitir esta validación tenemos dos opciones. La primera es convertir las cadenas a mayúsculas o minúsculas con los métodos .toUpperCase() y .toLowerCase() respectivamente. Métodos que veremos más adelante. La segunda opción es utilizar el método .equalsIgnoreCase() que omite si el carácter está en mayúsculas o en minúsculas.

boolean equalsIgnoreCase(String anotherString) Compara dos cadenas de caracteres omitiendo si los caracteres están en mayúsculas o en minúsculas.

int compareTo(String anotherString) Este método es un poco más avanzado que el anterior, el cual, solo nos indicaba si las cadenas eran iguales o diferentes En este caso compara a las cadenas léxicamente. Para ello se basa en el valor Unicode de los caracteres. Se devuelve un entero menor de 0 si la cadena sobre la que se parte es léxicamente menor que la cadena pasada como argumento. Si las dos cadenas son iguales léxicamente se devuelve un 0. Si la cadena es mayor que la pasada como argumento se devuelve un número entero positivo. ¿Pero, qué es esto de "mayor, menor o igual" léxicamente?. Para describirlo lo veremos con un pequeño ejemplo.

```
s1 = "Cuervo"
s2 = "Cuenca"
s1.compareTo(s2);
```

Compararíamos las dos cadenas. Los tres primeros caracteres son iguales "Cue". Cuando el método llega al 4 carácter tiene que validar entre la r minúscula y la n minúscula. Si utiliza el código Unicode llegará a la siguiente conclusión.

```
r (114) > n(110)
```

Y nos devolverá la resta de sus valores. En este caso un 4. Hay que tener cuidado, porque este método no tiene en cuenta las mayúsculas y minúsculas. Y dichos caracteres, aún siendo iguales, tienen diferentes código. Veamos la siguiente comparación

```
s1 = "CueRvo";
s2 = "Cuervo";
```

```
s1.compareTo(s2);
```

Nuevamente los tres caracteres iniciales son iguales. Pero el cuarto es distinto. Por un lado tenemos la r minúscula y por otro la r mayúscula. Así:

```
R(82) < r(114)
```

¿Qué entero nos devolverá el método compareTo()? ¿-32?

int compareToIgnoreCase(String str) Este método se comportará igual que el anterior. Pero ignorando las mayúsculas. Todo un alivio por si se nos escapa algún carácter en mayúsculas ;-) Otros métodos para la comparación de cadenas son:

```
boolean regionMatch( int thisoffset,String s2,int s2offset,int len );
boolean regionMatch( boolean ignoreCase,int thisoffset,String s2, int s2offset,int 1 );
```

Búsqueda de caracteres

Tenemos un conjunto de métodos que nos permiten buscar caracteres dentro de cadenas de texto. Y es que no nos debemos de olvidar que la cadena de caracteres no es más que eso: una suma de caracteres.

int indexOf(int ch) Nos devuelve la posición de un carácter dentro de la cadena de texto. En el caso de que el carácter buscado no exista nos devolverá un -1. Si lo encuentra nos devuelve un número entero con la posición que ocupa en la cadena.

int indexOf(int ch, int fromIndex) Realiza la misma operación que el anterior método, pero en vez de hacerlo a lo largo de toda la cadena lo hace desde el índice (fromIndex) que le indiquemos.

int lastIndexOf(int ch) Nos indica cual es la última posición que ocupa un carácter dentro de una cadena. Si el carácter no está en la cadena devuelve un -1. int lastIndexOf(int ch, int fromIndex) Lo mismo que el anterior, pero a partir de una posición indicada como argumento.

Búsqueda de subcadenas

Este conjunto de métodos es, probablemente, el más utilizado para el manejo de cadenas de caracteres. Ya que nos permiten buscar cadenas dentro de cadenas, así como saber la posición donde se encuentran en la cadena origen para poder acceder a la subcadena. Dentro de este conjunto encontramos:

int indexOf(String str) Busca una cadena dentro de la cadena origen. Devuelve un entero con el índice a partir del cual está la cadena localizada. Si no encuentra la cadena devuelve un -1.

int indexOf(String str, int fromIndex) Misma funcionalidad que indexOf(String str), pero a partir de un índice indicado como argumento del método.

int lastIndexOf(String str) Si la cadena que buscamos se repite varias veces en la cadena origen podemos utilizar este método que nos indicará el índice donde empieza la última repetición de la cadena buscada.

lastIndexOf(String str, int fromIndex) Lo mismo que el anterior, pero a partir de un índice pasado como argumento.

boolean startsWith(String prefix) Probablemente mucha gente se haya encontrado con este problema. El de saber si una cadena de texto empieza con un texto específico. La verdad es que este método podía obviarse y utilizarse el indexOf(), con el cual, en el caso de que nos devolviese un 0, sabríamos que es el inicio de la cadena.

boolean startsWith(String prefix, int toffset) Más elaborado que el anterior, y quizás, y a mi entender con un poco menos de significado que el anterior.

boolean endsWith(String suffix) Y si alguien se ha visto con la necesidad de saber si una cadena empieza por un determinado texto, no va a ser menos el que se haya preguntado si la cadena de texto acaba con otra. De igual manera que sucedía con el método .startsWith() podríamos utilizar una mezcla entre los métodos .indexOf() y .length() para reproducir el comportamiento de .endsWith(). Pero las cosas, cuanto más sencillas, doblemente mejores.

Métodos con subcadenas

Ahora que sabemos cómo localizar una cadena dentro de otra seguro que nos acucia la necesidad de saber cómo substraerla de donde está. Si es que no nos podemos estar quietos…

String substring(int beginIndex) Este método nos devolverá la cadena que se encuentra entre el índice pasado como argumento (beginIndex) hasta el final de la cadena origen. Así, si tenemos la siguiente cadena:

```
String s = "Víctor Cuervo";
```

El método:

```
s.substring(7);
```

Nos devolverá “Cuervo”.

String substring(int beginIndex, int endIndex) Si se da el caso que la cadena que queramos recuperar no llega hasta el final de la cadena origen, que será lo normal, podemos utilizar este método indicando el índice inicial y final del cual queremos obtener la cadena. Así, si partimos de la cadena…

```
String s = "En un lugar de la mancha....";
```

El método:

```
s.substring(6,11);
```

Nos devolverá la palabra “lugar”.

Hay que tener especial cuidado ya que es un error muy común el poner como índice final el índice del carácter último de la palabra a extraer. Cuando realmente es el índice + 1 de lo que queramos obtener.

Manejo de caracteres

Otro conjunto de métodos que nos permite jugar con los caracteres de la cadena de texto. Para ponerles en mayúsculas, minúsculas, quitarles los espacios en blanco, reemplazar caracteres...

String toLowerCase(); Convierte todos los caracteres en minúsculas.

String toUpperCase(); Convierte todos los caracteres a mayúsculas.

String trim(); Elimina los espacios en blanco de la cadena.

String replace(char oldChar, char newChar) Este método lo utilizaremos cuando lo que queramos hacer sea el remplazar un carácter por otro. Se reemplazarán todos los caracteres encontrados.

Conversión a String: valueOf()

Un potente conjunto de métodos de la clase String nos permite convertir a cadena cualquier tipo de dato básico: int, float, double,… Esto es especialmente útil cuando hablamos de números. Ya que en múltiples ocasiones querremos mostrarles como cadenas de texto y no en su representación normal de número. Así podemos utilizar los siguientes métodos:

- String valueOf(boolean b);
- String valueOf(int i);

- String valueOf(long l);
- String valueOf(float f);
- String valueOf(double d);
- String valueOf(Object obj);

ARRAYS

¿Qué es un array en Java?

Un array Java es una estructura de datos que nos permite almacenar una ristra de datos de un mismo tipo. El tamaño de los arrays se declara en un primer momento y no puede cambiar en tiempo de ejecución como puede producirse en otros lenguajes. La declaración de un array en Java y su inicialización se realiza de la siguiente manera:

```
tipo_dato nombre_array[];
nombre_array = new tipo_dato[tamanio];
```

Por ejemplo, podríamos declarar un array de caracteres e inicializarlo de la siguiente manera:

```
char arrayCaracteres[];
arrayCaracteres = new char[10];
```

Los arrays Java se numeran desde el elemento cero, que sería el primer elemento, hasta el tamaño-1 que sería el último elemento. Es decir, si tenemos un array de diez elementos, el primer elemento sería el cero y el último elemento sería el nueve. Para acceder a un elemento especifico utilizaremos los corchetes de la siguiente forma. Entendemos por acceso, tanto el intentar leer el elemento, como asignarle un valor.

```
arrayCaracteres[numero_elemento];
```

Por ejemplo, para acceder al tercer elemento lo haríamos de la siguiente forma:

```
// Lectura de su valor.
char x = arrayCaracteres[2];

// Asignación de un valor. Como se puede comprobar se pone el  número dos, que coincide con el tercer elemento. Ya que como  dijimos anteriormente el primer elemento es el cero.
arrayCaracteres[2] = 'b';
```

El objeto array, aunque podríamos decir que no existe como tal, posee una variable, la cual podremos utilizar para facilitar su manejo.

Tamaño del array: .length

Este atributo nos devuelve el número de elementos que posee el array. Hay que tener en cuenta que es una variable de solo lectura, es por ello que no podremos realizar una asignación a dicha variable. Por ejemplo esto nos serviría a la hora de mostrar el contenido de los elementos de un array:

```
char array[];
array = new char[10];

for (int x=0;x<array.length;x++)
  System.out.printnln(array[x]);
```

Una de las características de la orientación a objetos es la ocultación, es decir, que no podemos acceder a una variable declarada dentro de una clase a no ser que lo hagamos a través de un método de la clase. Aquí estamos accediendo a una variable. ¿Quizás sea porque no consideran a los arrays como objetos?.

Matrices o Arrays de varios subindices

Podremos declarar arrays de varios subíndices, pudiendo tener arrays de dos niveles (que serían similares a las matrices), arrays de tres niveles (que serían como cubos) y así sucesivamente. Consideremos que partir del tercer nivel se pierde la perspectiva geométrica.

Para declarar e inicializar un array de varios subíndices lo haremos de la siguiente manera:

```
tipo_dato nombre_array[][];
nombre_array = new tipo_dato[tamanio][tamanio];
```

De esta forma podemos declarar una matriz Java de 2x2 de la siguiente forma:

```
int matriz[][];
matriz = new int[2][2];
```

El acceso se realiza de la misma forma que antes:

```
int x = matriz[1][1]; // Para leer el contenido de un elemento
matriz[1][1] = x;     // Para asignar un valor.
```

Hay que tener en cuenta que para mostrar su contenido tendremos que utilizar dos bucles. Para saber el número de columnas lo haremos igual que antes mediante la variable .length, pero para saber el numero de filas que contiene cada columna lo tendremos que realizar de la siguiente manera:

```
matriz[numero_elemento].lenght;
```

Nuestra lectura de los elementos de una matriz quedaría de la siguiente forma:

```
int matriz[][];
matriz = new int[4][4];
for (int x=0; x < matrix.length; x++) {
  for (int y=0; y < matriz[x].length; y++) {
     System.out.println (matriz[x][y]);
  }
}
```

Incialización de Arrays en Java

Existe una forma de inicializar un array con el contenido, amoldándose su tamaño al número de elementos a los que le inicialicemos. Para inicializar un array utilizaremos las llaves de la siguiente forma:

```
tipo_dato array[] = {elemento1,elemento2,...,elementoN};
```

Así, por ejemplo, podríamos inicializar un array o una matriz:

```
// Tenemos un array de 5 elementos.
char array[] = {'a','b','c','d','e'};

// Tenemos un array de 4x4 elementos.
int array[][] = { {1,2,3,4}, {5,6,7,8}};
```

ARRAYLIST

La clase ArrayList en Java, es una clase que permite almacenar datos en memoria de forma similar a los Arrays, con la ventaja de que el número de elementos que almacena, lo hace de forma dinámica, es decir, que no es necesario declarar su tamaño como pasa con los Arrays.

Los ArrayList nos permiten añadir, eliminar y modificar elementos (que pueden ser objetos o elementos atómicos/literales) de forma trasparente para el programador.

Antes de pasar a explicar el manejo de los ArrayList, dejamos un enlace al JavaDoc, pulsando AQUI.

Los principales métodos para trabajar con los ArrayList son los siguientes:

```
// Declaración de un ArrayList de "String". Puede ser de cualquier otro Elemento u Objeto (float, Boolean, Object, ...)
ArrayList<String> nombreArrayList = new ArrayList<String>();
```

```
// Añade el elemento al ArrayList
nombreArrayList.add("Elemento");
```

```
// Añade el elemento al ArrayList en la posición 'n'
nombreArrayList.add(n, "Elemento 2");
```

```
// Devuelve el numero de elementos del ArrayList
nombreArrayList.size();
```

```
// Devuelve el elemento que esta en la posición '2' del ArrayList
nombreArrayList.get(2);
```

```
// Comprueba se existe del elemento ('Elemento') que se le pasa como parametro
nombreArrayList.contains("Elemento");
```

```
// Devuelve la posición de la primera ocurrencia ('Elemento') en el ArrayList
nombreArrayList.indexOf("Elemento");
```

```
// Devuelve la posición de la última ocurrencia ('Elemento') en el ArrayList
nombreArrayList.lastIndexOf("Elemento");
```

```
// Borra el elemento de la posición '5' del ArrayList
nombreArrayList.remove(5);
```

```
// Borra la primera ocurrencia del 'Elemento' que se le pasa como parametro.
nombreArrayList.remove("Elemento");
```

```
//Borra todos los elementos de ArrayList
nombreArrayList.clear();
```

```
// Devuelve True si el ArrayList esta vacio. Sino Devuelve False
nombreArrayList.isEmpty();
```

```
// Copiar un ArrayList
ArrayList arrayListCopia = (ArrayList) nombreArrayList.clone();
```

```
// Pasa el ArrayList a un Array
Object[] array = nombreArrayList.toArray();
```

Otra cosa muy importante a la hora de trabajar con los ArrayList son los "Iteradores" (Iterator). Los Iteradores sirven para recorrer los ArrayList y poder trabajar con ellos. Los Iteradores solo tienen tres métodos que son el *"hasNext()"* para comprobar que siguen quedando elementos en el iterador, el *"next()"* para que nos de el siguiente elemento del iterador; y el *"remove()"* que sirve para eliminar el elemento del Iterador.

Bueno, si esto no te ha quedado muy claro, pasamos a poner el primer ejemplo. En el siguiente fragmento de código, declaramos un ArrayList de Strings y lo rellenamos con 10 Strings (Elemento i). Esto lo hacemos con el método *"add()"*. Después añadimos un nuevo elemento al ArrayList en la posición '2' (con el metodo *"add(posición,elemento)"*) que le llamaremos "Elemento 3" y posteriormente imprimiremos el contenido del ArrayList, recorriendolo con un Iterador. El fragmento de este código es el siguiente:

```
// Declaración el ArrayList
ArrayList<String> nombreArrayList = new ArrayList<String>();
```

```
// Añadimos 10 Elementos en el ArrayList
for (int i=1; i<=10; i++){
        nombreArrayList.add("Elemento "+i);
}
```

```
// Añadimos un nuevo elemento al ArrayList en la posición 2
nombreArrayList.add(2, "Elemento 3");
```

```
// Declaramos el Iterador e imprimimos los Elementos del ArrayList
Iterator<String> nombreIterator = nombreArrayList.iterator();
while(nombreIterator.hasNext()){
		String elemento = nombreIterator.next();
		System.out.print(elemento+" / ");
}
```

Ejecutando esta código obtenemos por pantalla lo siguiente:

```
Elemento 1 / Elemento 2 / Elemento 3 / Elemento 3 / Elemento 4 / Elemento 5 /
Elemento 6 / Elemento 7 / Elemento 8 / Elemento 9 / Elemento 10 /
```

Como se observa en el resultado, tenemos repetido el elemento *"Elemento 3"* dos veces y esto lo hemos puesto a propósito para mostrar el siguiente ejemplo. Ahora para seguir trabajando con los ArrayList, lo que vamos a hacer es mostrar el número de elementos que tiene el ArrayList y después eliminaremos el primer elemento del ArrayList y los elementos del ArrayList que sean iguales a *"Elemento 3"*, que por eso lo hemos puesto repetido. El *"Elemento 3"* lo eliminaremos con el método *"remove()"* del iterador. A continuación, mostramos el código que realiza lo descrito:

```
// Recordar que previamente ya hemos declarado el ArrayList y el Iterator de la
siguiente forma:
// ArrayList<String> nombreArrayList = new ArrayList<String>();
// Iterator<String> nombreIterator = nombreArrayList.iterator();

// Obtenemos el número de elementos del ArrayList
int numElementos = nombreArrayList.size();
System.out.println("nnEl ArrayList tiene "+numElementos+" elementos");
```

```
// Eliminamos el primer elemento del ArrayList, es decir el que ocupa la posición '0'
System.out.println("n... Eliminamos el primer elemento del ArrayList
("+nombreArrayList.get(0)+")...");
nombreArrayList.remove(0);
```

```
// Eliminamos los elementos de ArrayList que sean iguales a "Elemento 3"
System.out.println("n... Eliminamos los elementos de ArrayList que sean iguales a
"Elemento 3" ...");
nombreIterator = nombreArrayList.iterator();
while(nombreIterator.hasNext()){
		String elemento = nombreIterator.next();
		if(elemento.equals("Elemento 3"))
				nombreIterator.remove();	// Eliminamos el Elemento que
hemos obtenido del Iterator
}
```

```
// Imprimimos el ArrayList despues de eliminar los elementos iguales a "Elemento 3"
System.out.println("nImprimimos los elementos del ArrayList tras realizar las
eliminaciones: ");
nombreIterator = nombreArrayList.iterator();
while(nombreIterator.hasNext()){
        String elemento = nombreIterator.next();
        System.out.print(elemento+" / ");
}
```

```
// Mostramos el numero de elementos que tiene el ArrayList tras las eliminaciones:
numElementos = nombreArrayList.size();
System.out.println("nNumero de elementos del ArrayList tras las eliminaciones =
"+numElementos);
```

Como salida a este código tenemos lo siguiente:

```
El ArrayList tiene 11 elementos

... Eliminamos el primer elemento del ArrayList (Elemento 1)...

... Eliminamos los elementos de ArrayList que sean iguales a "Elemento 3" ...

Imprimimos los elementos del ArrayList tras realizar las eliminaciones:
Elemento 2 / Elemento 4 / Elemento 5 / Elemento 6 / Elemento 7 / Elemento 8 /
Elemento 9 / Elemento 10 /

Numero de elementos del ArrayList tras las eliminaciones = 8
```

Hemos eliminado 3 elementos del ArrayList de dos formas distintas, preguntando por la posición que ocupa un elemento en el ArrayList y preguntando por el contenido de algún elemento del ArrayList. Como se observa es muy importante saber manejar los Iteradores ya que con ellos podemos tratar los elementos del ArrayList.